On-line kod:
www.samspublishing.com

Nesne Yönelimli C++ Programlama Kılavuzu

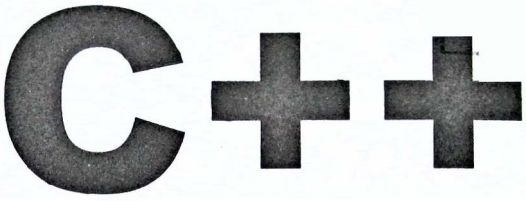
- C++ programlamanın temelleri
- Döngü ve kararlar
- Yapılar ve fonksiyonlar
- Nesneler ve sınıflar
- Diziler ve karakter katarları
- Kalıtım, işaretçiler ve sanal fonksiyonlar
- Şablonlar ve istisnalar
- Microsoft Visual C++, Borland C++ Builder ve Console Graphics Lite uygulamalarına adanmış bölümler
- Yüzlerce soru ve örnek alıştırma
- Program mimarisini Unified Modeling Language (UML) ile görselleştirin...
- CASE modelleme kullanarak sınıf keşfi ile nesne yönelimli tasarım ve analiz
- Sık karşılaşılan programlama problemlerine mükemmel bir çözüm: Standard Template Library

Robert Lafore
Editör: Selçuk Tüzel

Uzmanlar İçin!

SAMS
ALFA

Uzmanlar İçin



Nesne Yönelimli Programlama Kılavuzu

Robert Lafore

Çeviri: Duygu Arbatlı Yağcı

Editör: Selcuk Tüzel

ALFA

Alfa Yayınları: 1143
Bilgisayar Dizi No: 214

Uzmanlar İçin
C++
Nesne Yönelimli Programlama Kılavuzu

Robert Lafore

Çeviri: Duygu Arbatlı Yağcı
Çeviri Editörü: Selçuk Tüzel

Orijinal Adı: Object, Oriented Programming in C++

1. Baskı : Temmuz 2002
ISBN : 975-297-128-8

Yayıncı ve Genel Yayın Yönetmeni : M. Faruk Bayrak
Yayın Koordinatörü : Rana Gürtuna
Yayın Yönetmeni : Mehmet Çömlekçi
Kapak Tasarımı : Nevruz Kıran
Teknik Editör : Zeynep Çömlekçi
Pazarlama ve Satış Müdürü : Vedat Bayrak

ALFA Basım Yayım Dağıtım Ltd. Şti.
Ticarethane Sk. No: 41/1 34410 Cağaloğlu, İstanbul
Tel : (212) 511 5303-513 8751-512 3046 Fax : (212) 519 3300
www.alfakitap.com
E-mail : info@alfakitap.com

ALFA/AKTÜEL KİTABEVİ
Burç Sinema Pasajı No: 34
Altıparmak / BURSA
Tel : (224) 223 60 16

Baskı ve Cilt
MELİSA MATBAACILIK
Çiftehavuzlar Yolu Acar Sanayi Sitesi
No: 8 Bayrampaşa-İstanbul
Tel/Fax : (212) 501 97 57

İçindekiler

Yazar Hakkında

Robert Lafore 1982'den beri bilgisayar programlama konusunda kitaplar yazmaktadır. Yazarın eserleri arasında *Assembly Language Programming for the IBM PC* (IBM PC İçin Assembly Programlama), *C Programming Using Turbo C++* (Turbo C++ ile C Programlama), *C++ Interactive Course* (Etkileşimli C++ Dersi), ve *Data Structures and Algorithms in Java* (Java'da Veri Yapıları ve Algoritmalar) da yer almaktadır. Robert Lafore, matematik ve elektrik mühendisliği mezunu olup, 4K belleğin lüks sayıldığı PDP-5 günlerinden bu yana programcılıkla uğraşmaktadır. Yazarın ilgi alanları arasında doğa gezileri, sörf ve matematik eğlenceleri yer almaktadır.

İthaf

Bu kitap GGL'ye ve onun yılmaz şahsiyetine ve heyecanına ithaf edilmiştir.

Teşekkür

Öncelikle bütün okurlarıma teşekkür ederim. Öneri ve düzeltmelerinden dolayı şu bilgisayar bilimi profesörlerine minnettarım: Denver'deki Regis University'den Bill Blomberg; Güney Carolina'daki College of Charleston'dan Richard Daehler-Wilking; İsveç Kraliyet Teknoloji Enstitüsü'nden Frank Hoffmann ve California'daki San Jose State University'den David Blockus. Ayrıntılı fikirleri ve problemleri yakalamaktaki üstün dikkatinden dolayı California'nın Fremont kentindeki Ohlone College'dan David Topham'a özel olarak teşekkürlerimi sunarım.

Sams Publishing'de Michael Stephens yayımcılığın ayrıntıları konusunda bana uzman ve dostane bir aracı oldu. Redaktör Robin Rowe ve Teknik Editör Mark Cashman beni benden korumak için büyük itina göstermeye çabaladılar; şayet herhangi bir noktada aksayan bir şey olduysa bu tamamen benim hatamdır. Proje Yöneticisi Christina Smith, tüm parçaların inanılmayacak kadar kısa bir zaman içinde bir araya gelmesini sağladı. Angela Boley, işlerin pürüzsüz ilerlemesine yardımcı oldu. Matt Wyndala da uzman bir düzeltmen olarak görev yaptı. Hepinize minnettarım.

GİRİŞ

Bu kitap size C++ programlama diliyle program yazmayı öğretir. Ancak bunun biraz ötesine de geçer. Son birkaç yıl içinde yazılım geliştirme sahnesinde bazı büyük yenilikler meydana geldi. Bu kitap C++'ı söz konusu gelişmeler çerçevesinde öğretir. Gelin şimdi bu gelişmelerin ve yeniliklerin ne olduklarına bakalım.

Programlamadaki Yenilikler

Eski dönemde, yani yaklaşık 20 sene öncesinde, bir projeye başlayan programcılar neredeyse hemen işin başından itibaren kod yazmaya girişirlerdi. Ancak, programlama projeleri büyüdükçe ve karmaşıklaştıkça bu yaklaşımın pek iyi netice vermediği anlaşıldı. Problem, karmaşıklıktan kaynaklanıyordu.

Büyük programlar muhtemelen insanoğlunun meydana getirdiği en karmaşık varlıklardır. Bu karmaşıklıktan dolayı, programlar hata içermeye eğilimlidir. Üstelik yazılım hataları pahalıya mal olabilir, hatta (örneğin hava trafik kontrolü sahasında) hayati tehlikelere dahi yol açabilir. Karmaşıklık problemiyle baş edebilmek için programlama alanında şu üç büyük yenilik geliştirildi:

- Nesne yönelimli Programlama (Object Oriented Programming – OOP)
- Birleştirilmiş Modelleme Dili (Unified Modeling Language – UML)
- İleri yazılım geliştirme süreçleri

Bu kitap, C++ dilini yukarıda sıralanan bu gelişmeleri dikkate alarak öğretir. Kitabı okuyarak sadece bir programlama dili öğrenmekle kalmayacaksınız, aynı zamanda yazılım geliştirmeyi kavramsallaştırmanın yeni yollarını da öğreneceksiniz.

Nesne Yönelimli Programlama

Acaba nasıl oldu da nesne yönelimli programlama, yazılım projelerinin çoğunluğu için tercih edilen yaklaşım haline geldi? Nesne yönelimli programlama, karmaşıklıkla baş edebilmek için yeni ve güçlü bir yöntem sunar. Nesne yönelimli programlama, bir programı, icra edilecek işlemler dizisi olarak görmek yerine, belli özellikleri olan ve belli eylemler yapabilen nesneler olarak değerlendirir. Konuyu daha iyi öğrenene kadar bu size biraz karmaşık ve anlaşılmaz gelebilir ama nesne yönelimli programlama; daha açık, daha güvenilir ve bakımı daha kolay programlar yazılmasını sağlar.

Bu kitabın ana amaçlarından biri, nesne yönelimli programlamayı öğretmektir. Nesne yönelimli programlama konusuna mümkün olduğunca erken bir noktada başlıyoruz ve konunun tüm ana özelliklerini işliyoruz. Örnek programlarımızın çoğunluğu nesne yönelimlidir.

Birleştirilmiş Modelleme Dili (UML)

Birleştirilmiş Modelleme Dili (UML - Unified Modeling Language), çok çeşitli şemalardan oluşan bir grafik dilidir. UML, bir programın ne yapması gerektiğini belirleme konusunda program analistlerine; programların çalışma şeklini tasarlama ve anlama konusunda da programcılara yardımcı olur. UML güçlü bir araçtır ve programlama işini hem kolaylaştırabilir hem de daha etkin kılabilir.

Bölüm 1'de UML'e genel bir bakış sunduk. Daha sonra kitap boyunca UML'in spesifik özelliklerini ele aldık. Her UML özelliğini, o özelliğin, ele alınan nesne yönelimli programlama konusunu açıklamaya yardımcı olacağı yerde tanıttık. Bu şekilde UML'i sıkıntı çekmeden öğrenebilirsiniz; UML de C++ öğrenmenize yardımcı olur.

Programlama Dilleri ve Geliştirme Platformları

Nesne yönelimli programlar arasında en yaygın kullanılanı açık farkla C++'tır. Nesne yönelimli programlama dilleri alanına yeni eklenen Java'da bazı özellikler –örneğin işaretçiler, şablonlar ve çoklu kalıtım- eksiktir. Java bu yüzden C++'tan daha az güçlü ve esnektir. (Eğer günün birinde Java öğrenecek olursanız, bu dilin sözdizimi C++'a çok benzediğinden, C++ öğrenmek size Java öğrenime konusunda avantaj sağlayacaktır.)

Yakın dönemde C# gibi nesne yönelimli birkaç dil daha ortaya çıktı ama bunlar henüz C++ kadar yaygın kabul görmüş değildir.

Son zamanlara kadar C++ standartları sürekli gelişmekteydi. Bu da her derleyici yayıncısının belli ayrıntıları kendine göre halletmesine neden oluyordu. Ancak Kasım 1997'de ANSI/ISO C++ standartları komisyonu artık Standart C++ olarak bilinen nihai taslağı kabul etti. (ANSI, American National Standards Institute'e yani Amerikan Ulusal Standartlar Enstitüsü'ne karşılık gelir; ISO da, International Standards Institute yani Uluslararası Standartlar Enstitüsü'dür.)

Standart C++, C diline Standart Şablon Kütüphanesi (Standard Template Library - STL) gibi pek çok yeni özellik eklemiştir. Bu kitapta (özel olarak belirteceğimiz birkaç noktanın dışında) Standart C++'a uyduk.

C++ için en popüler geliştirme ortamları Microsoft ve Borland (Inprise) tarafından üretilmekte ve muhtelif Windows sürümleri üzerinde çalışmaktadır. Bu kitapta, tüm örnek programların hem Borland hem de Microsoft derleyicilerinin mevcut sürümleri üzerinde çalışabiliyor olmasına itina gösterdik. (Bu derleyiciler konusunda daha fazla bilgi için Ek C "Microsoft Visual C++" ve Ek D "Borland C++Builder"a bakınız.)

Bu Kitap Ne İşe Yarar?

Bu kitap, C++ dilini ve Microsoft ya da Borland derleyicilerini kullanarak nesne yönelimli programlamayı öğretir. Kitap aynı zamanda, UML'i ve yazılım geliştirme süreçlerini tanıtır. Kitap gerek profesyonel programcılar için, gerek öğrenciler için, gerek de konunun meraklıları için uygun bir kaynaktır.

Yeni Kavramlar

Nesne yönelimli programlama Pascal, Basic ve C gibi geleneksel dillerde programlama yapanlar açısından yeni kavramlar içerir. Sınıflar, katılım ve çok biçimlilik gibi bu yeni kavramlar, nesne yönelimli programlamanın kalbinde yatar. Nesne yönelimli bir programlama dilinin spesifik özelliklerini ele alırken, bu temel kavramları gözden kaçırma tehlikesi doğabilir. Pek çok kitap, okuyucuları dilin özelliklerinin ayrıntılarına boğarken bu özelliklerin neden varolduğunu açıklamayı ihmal eder. Bu kitap, genel manzarayı sürekli göz önünde tutmaya ve ayrıntıları bu genel kavramlarla ilintilendirmeye çalışır.

Aşamalı Yaklaşım

Kitapta aşamalı bir yaklaşım izledik. Önce çok basit programlama örneklerinden başladık, sonra öğrene öğrene tam tekmilli nesne yönelimli uygulamalara vardık. Yeni kavramları yavaş yavaş tanıttık ve bu sayede yeni bir kavrama geçmeden önce mevcut konuyu hazmetmenizi sağlamayı amaçladık. Yeni kavramları açıklamak için mümkün olan her yerde şekil ve resimlerden yararlandık. Kitabın sınıf ortamındaki yararını desteklemek için çoğu bölümde sorulara ve programlama alıştırmalarına yer verdik. Soruların ve ilk bir kaç (yıldızla işaretlen-

miş) alıştırmanın cevapları Ek G'de yer alır. Öğrencinin kendisini farklı düzeylerde zorlayabilmesi için alıştırmalar değişik zorluk derecelerinde sunulmuştur.

Kitaptan Yararlanabilmeniz İçin

Daha önce hiç programlama tecrübeniz olmadıysa bile bu kitabı kullanabilirsiniz. Ancak, şu da bir gerçek: böyle bir tecrübeniz varsa, örneğin Visual Basic biliyorsanız, bunun da kesinlikle bir zararı olmaz!

Kitabı izlemek için C dilini bilmenize gerek yok. Pek çok C++ kitabı, C bildiğinizi varsayar ama bizim kitabımız böyle değil. Bu kitap C++'ı sıfırdan başlayarak öğretir. C biliyorsanız zararı yok. Ama C ile C++'ın ortak özelliklerinin ne kadar kısıtlı olduğu sizi şaşırtabilir.

Ayrıca, Microsoft Windows'un, uygulamaları başlatma ve dosya kopyalama gibi temel özelliklerini kullanabiliyor olmanız gerekli.

Yazılım ve Donanım

Bir C++ derleyicisine ihtiyacınız var. Kitaptaki programlar Microsoft Visual C++ ve Borland C++Builder üzerinde test edilmiştir. Her iki derleyicinin de öğrenciler için ucuz sürümleri mevcuttur.

Ek C'de Microsoft derleyicisi, Ek D'de de Inprise (Borland) derleyicisi hakkında ayrıntılı bilgi verdik. Standart C++'a sadık kalan diğer derleyiciler de muhtemelen bu kitaptaki programların çoğunluğunu herhangi bir değişikliğe gerek duymadan kullanabilir.

Bilgisayarınızın, seçtiğiniz derleyiciyi çalıştırabilecek işlemci hızına, belleğe ve sabit disk alanına sahip olması gereklidir. Bu konudaki ihtiyaçları saptamak için derleyici üreticisinin spesifikasyonlarını kontrol edebilirsiniz.

Konsol Modu Programlar

Kitap boyunca çok sayıda örnek program var. Bunlar, derleyici ortamında ya da doğrudan bir MS-DOS kutusunda, karakter bazlı bir pencerede çalışan programlardır. Bu sayede tamamen grafik tabanlı çalışan Windows programları yazmanıza gerek kalmadan örnekleri çalıştırabilirsiniz.

Örnek Programların Kaynak Kodları

Örnek programların kaynak kodlarını `http://www.samspublishing.com` adresindeki Sams Publishing Web sitesinden indirebilirsiniz.

Kitapla ilgili verilere ulaşmak için, kitabın İngilizce orijinal adını (künye sayfasında görebilirsiniz) ya da İngilizce baskının ISBN numarasını (ISBN 0-672-32308-7) yazın ve Search düğmesini tıklayın. Verileri bulunca örnek programların kaynak koduna ulaşmak için Source Code bağlantısını tıklayın.

Console Graphics Lite

Resim çizmek için birkaç örnek program, bizim Console Graphics Lite adını verdiğimiz bir grafik kütüphanesinden yararlanır. Grafikler konsol karakterlerine dayalı olduğu için pek sofistike sayılmazlar. Ama bazı ilginç programların yazılmasına imkan sağlarlar. Bu kütüphanenin dosyaları da, örnek programların kaynak koduyla birlikte yukarıda adresi verilen Web sitesinde mevcuttur.

Grafik örneklerini derlemek ve çalıştırmak için programınıza bir başlık dosyası eklemeniz gereklidir. Bu başlık dosyası, derleyicinize bağlı olarak `MSOFTCON.H` ya da `BORLACON.H` dosyasıdır. Ayrıca, yine derleyicinize uyacak şekilde, `MSOFTCON.CPP` ya da `BORLACON.CPP` dosyalarını da projelerinize eklemeniz gerekir. "Console Graphics Lite" adlı Ek E, bu dosyaların listesini ve nasıl kullanılacağını anlatır. Ek C ve D, Microsoft ve Borland derleyici ortamlarında dosya ve projelerle nasıl çalışılacağını açıklar.

Programlama Alıştırmaları

Her bölümde aşağı-yukarı 12 alıştırma mevcuttur. Bunların her biri tam bir C++ programının meydana getirilmesini gerektirir. Her bölümdeki ilk 3-4 alıştırmanın çözümleri Ek G'de verilmiştir. Diğer alıştırmaları ise okuyucuların kendi başlarına çözmesi gereklidir. (Ancak C++ dersi veriyorsanız, bu bölümünün sonundaki "Öğretmenlere Not" başlıklı bölümü okuyunuz.)

Sandığınızdan Daha Kolay

C++'ı öğrenmenin zor olduğu yönünde bir şeyler duymuş olabilirsiniz. Ama gerçekte C++ diğer dillere oldukça benzer; bir-iki "büyük kavram" ilave edilmiştir. Bu yeni kavramlar zaten kendi başlarına ilginçtir; bunları öğrenmenin size büyük keyif vereceğini düşünüyoruz. Söz konusu yeni kavramlar bir yandan da programlama kültürünün bir parçası haline gelmektedir. Tıpkı evrim ve psikoanaliz gibi herkesin bu konularla ilgili biraz bilgi sahibi olmasında yarar vardır. Kitabın bir yandan C++ programlamanın ayrıntılarını anlatırken, öte yandan bu yeni kavramları öğrenmeyi de sizin için bir zevk haline getireceğini umuyoruz.

Öğretmenlere Not

Öğretmenler ve C++ ya da C bilen okuyucular, kitapta izlediğimiz yaklaşım ve kitabın yapısı konusunda aşağıda sunduğumuz ilave bilgilerden yararlanabilirler.

Standart C++

Derleyicilerin tuhaflıklarını çözmek için kullandığımız küçük bazı istisnaların dışında bu kitaptaki tüm programlar Standart C++ ile uyumludur. Standart C++'a dahil olan Standart Şablon Kütüphanesi'ne (Standard Template Library - STL) tam bir bölüm ayırdık.

Birleştirilmiş Modelleme Dili (Unified Modeling Language - UML)

Bu kitapta UML'i kitabın içine yaydık. UML konularını yeri geldikçe anlattık. Örneğin, farklı sınıflar arasındaki iletişimi anlatmaya başladığımız noktada UML sınıf şemalarına da girdik. Genelleme de kalıtımla ilgili bölümde anlatıldı.

"Genel Görünüm" adlı Bölüm 1'de, çeşitli UML konularının ilk olarak nerede anlatılmaya başlandığına dair bir liste bulabilirsiniz.

Yazılım Geliştirme Yöntemleri

Biçimsel yazılım geliştirme yöntemleri programlamanın giderek daha fazla önem kazanan bir unsuru haline gelmektedir. Ayrıca, öğrenciler sık sık nesne yönelimli programlama sürecini şaşırtıcı bulurlar. Bunları dikkate alarak yazılım geliştirme yöntemlerine bir bölüm ayırdık ve

burada nesne yönelimli programlamaya ağırlık verdik. Daha eski kitaplarda CRC kartları üzerine yoğunlaşılmıştır. Ama o zamandan bu yana yazılım geliştirmedeki ağırlık noktası daha çok vaka analizine doğru kaydığı için biz de programlama projelerinin analizinde bu yöntemi kullandık.

C++, C İle Aynı Şey Değil

Bazı kurumlarda hala, öğrencilerin C++ öğrenmeden önce C öğrenmeleri istenir. Bizim görüşümüze göre bu bir hatadır. C ve C++ tamamen ayrı dillerdir. Sözdizimlerinin benzer olduğu ve C'nin, C++'ın bir altkümesi olduğu doğrudur. Ama bu benzerlik büyük ölçüde kronolojik bir tesadüften ibarettir. Gerçekte bir C++ programında izlenen temel yaklaşım, bir C programındaki ile karşılaştırıldığında tamamen farklıdır.

C++, ciddi yazılım geliştirme alanında C'nin yerini almıştır. Bu nedenle C++ öğrenmeden önce C bilmek gerektiğine ya da C bilmenin iyi olacağına inanmıyoruz. C bilmeyen öğrenciler, önce C öğrenip sonra C++ öğrenmek gibi verimsiz bir yöntemi uygulama zahmetinden kurtulup, doğrudan C++ öğrenebilirler. C bilen öğrenciler ise bazı bölümlerin belli kısımlarını hızlıca okuyabilirler. Ancak, bu kitaptaki konuların kayda değer bir kısmının onlar için de çok yeni olduğunu göreceklerdir.

Nesne Yönelimli Programlama için En Uygun Düzenleme

Kitaba, C ve C++'ın ortak özelliği olan prosedürel kavramları anlatarak başlayabilir, bunlar hazmedildikten sonra yeni nesne yönelimli programlama kavramlarına geçebilirdik. Bu yaklaşım bize zararlı geldi. Çünkü, amaçlarımızdan biri gerçek nesne yönelimli programlamaya mümkün olduğunca çabuk girmekti. Bu amacımıza uygun olarak Bölüm 6'da sınıflara başlayana kadar prosedürel temeller konusunda sadece asgari bilgilere yer verdik. İlk bölümler bile, yoğun bir şekilde, C yerine C++ kullanımına yöneltilmiştir.

Bazı kavramları C konusunda yazılmış geleneksel kitaplara göre daha erken noktalarda tanıttık. Örneğin, yapılar C++'ı anlamak için önemli bir özelliktir; çünkü sınıflar, sözdizimi açısından yapıların bir uzantısıdır. Bu sebeple yapıları Bölüm 5'de anlatmaya başlayarak, sınıf konusuna girdiğimizde yapılarla ilgili temelin hazır olmasını sağladık.

İşaretçiler gibi bazı kavramlar ise geleneksel C kitaplarına göre daha ileri noktalarda tanıtıldı. Nesne yönelimli programlamanın temel kavramlarını izleyebilmek için işaretçileri anlamaya gerek yoktur. Ayrıca işaretçiler C ve C++ öğrencileri için genelde öğrenmesi zor bir konudur. Bu düşünceyle işaretçiler konusunu nesne yönelimli programlamanın ana kavramları iyice hazmedilene kadar erteledik, ileriki bölümlere bıraktık.

C++'ın Üstün Alternatif Özellikleri

C'nin bazı özelliklerinin yerini C++'ta daha üstün yeni yaklaşımlar almıştır. Örneğin C'de giriş/çıkışın temelini oluşturan `printf()` ve `scanf()` fonksiyonları C++'ta ancak nadiren kullanılırlar. Çünkü `cout` ve `cin` daha üstündür. Bu nedenle, artık geride kalan bu C fonksiyonlarını anlatmadık. Benzer şekilde, C'deki `#define` sabitleri ve makroların yerine de artık büyük ölçüde C++'ın `const` niteleyicisi ve yerel (inline) fonksiyonlar geçtiğinden `#define` sabitlerini ve makroları çok kısaca ele aldık.

Gereksiz Özellikleri Geri Plana Atmak

Kitabın asıl amacı nesne yönelimli programlamayı öğretmek olduğundan, nadiren kullanılan bazı C özellikleri ile nesne yönelimli programlama ile özellikle alakalı olmayan konuları bir kenara atabiliriz. Örneğin, nesne yönelimli programlamayı anlamak için C'nin bit tabanlı (bitwise) operatörlerini (tek tek bit'ler üzerinde işlem yapan operatörleri) anlamak gerekli değildir. Bu ve birkaç diğer konuyu dışarıda bırakırsak ya da özet halinde ele alırsak C++'ın ana özelliklerini anlamak konusunda bir kaybımız olmayacaktır.

Tüm bunların neticesinde nesne yönelimli programlamanın temelleri üzerine odaklanan; okuyucuyu nazik ama seri bir şekilde, yeni kavramları ve bunların gerçek programlama problemlerinde kullanımını anlamaya götüren bir kitap ortaya çıkmıştır.

Alıştırmalar

Yıldızla işaretlenmemiş alıştırmaların cevapları kitapta yer almaz. Ancak yetkili öğretim üyeleri, önerilen çözümleri Sams Publishing Web şitesinden temin edebilirler. Bunun için kitabın künye sayfasında yer alan İngilizce orijinal adını ya da İngilizce baskının ISBN numarasını (ISBN 0-672-32308-7) yazın ve Search düğmesini tıklayarak kitabın sayfasına ulaşın. Daha sonra da Downloads bağlantısını tıklayın.

Alıştırmaların zorluk derecesi ciddi ölçüde değişiktir. Her bölümde ilk alıştırmalar gayet kolaydır. Sonrakiler ise giderek zorlaşır. Öğretmenler muhtemelen sadece sınıfın seviyesine uygun alıştırmaları ödev olarak vermeyi uygun göreceklerdir.

BİRİNCİ BÖLÜM

GENEL GÖRÜNÜM

Nesne Yönelimli Programlamaya Neden İhtiyaç Duyarız?

Nesne Yönelimli Dillerin Özellikleri

C++ ve C

Esaslar Üzerine

Birleştirilmiş Modelleme Dili (Unified Modeling Language – UML)

Bu kitap size C++ dilinde nasıl programlama yapabileceğinizi öğretir. C++, *nesne yönelimli programlamayı* destekleyen bir bilgisayar dilidir. Nesne yönelimli programlamaya neden gerek duyarız? Nesne yönelimli programlama, C, Pascal ve BASIC gibi geleneksel dillerin yapamadığı neleri yapabilir? Nesne yönelimli programlamanın ardında yatan prensipler nelerdir? Nesne yönelimli programlamada iki temel kavram mevcuttur: *Nesneler* ve *sınıflar*. Bu terimler ne anlama gelir? C++ ve daha eski bir dil olan C arasındaki ilişki nedir?

Bu bölümde bu soruların cevaplarını araştırıyoruz, ayrıca kitabın kalan kısmında ele alınacak özelliklerle ilgili genel bir bilgi veriyoruz. Burada anlattıklarımız ister istemez daha çok – kısa ve öz olmakla beraber- genel bir bilgi niteliğinde olacak. Anlatılanları biraz soyut bulursanız, endişelenmeyin. İleriki bölümlerde ayrıntılı olarak üzerinde durdukça burada bahsedilen kavramlar yerine oturacaktır.

Nesne Yönelimli Programlamaya Neden İhtiyaç Duyarız?

Nesne yönelimli programlama, daha önceki programlama tekniklerinde yaşanan kısıtlamalar neticesinde geliştirildi. Nesne yönelimli programlamanın kıymetini anlayabilmek için bu kısıtlamaların neler olduğunu ve geleneksel programlama dillerinde bu kısıtların nasıl ortaya çıktığını anlamamız gerekiyor.

Prosedürel Diller

C, Pascal, FORTRAN ve benzeri diller *prosedürel dillerdir*. Yani, dildeki her ifade bilgisayara bir iş yapması gerektiğini söyler: Girdiyi al, girdi olarak alınan sayıları topla, altıya böl, sonucu ekranda göster. Prosedürel dilde yazılan bir program, bir komut listesidir.

Çok küçük programlar için başka bir düzenleyici prensibe (çoğunlukla *paradigma* olarak adlandırılır) gerek yoktur. Programcı komut listesini oluşturur, bilgisayar bu listedeki komutları yerine getirir.

Fonksiyonlara Bölmek

Programların hacmi büyüdükçe tek bir komut listesinin idaresi güçleşir. Birkaç yüz ifadeden daha fazlasına sahip bir program, daha küçük birimlere bölünmediği takdirde az sayıda programcı tarafından anlaşılabilir. Bu nedenle, programların kendilerini üreten insanlara daha anlaşılır görünmesi amacıyla *fonksiyon* (*function*) yapısı geliştirildi. ("Function" terimi C ve C++'ta kullanılır. Diğer dillerde bu kavram subroutine, subprogram veya procedure olarak adlandırılır.) Prosedürel bir program fonksiyonlara bölünmüştür. Her fonksiyonun (en azından ideal olarak) açıkça tanımlanmış bir amacı vardır ve program içindeki diğer fonksiyonlarla arasında açıkça tanımlanmış bir arayüz mevcuttur.

Bir programı fonksiyonlara bölme fikri daha da genişletilebilir: Birkaç fonksiyon biraraya getirilip *modül* denilen (bu, genellikle bir dosyadır) daha büyük bir birim içinde gruplanabilir. Fakat prensip her iki durumda da aynıdır: Bir komut listesini yürüten bileşenlerin gruplanması.

Bir programı fonksiyonlara ve modüllere ayırmak *yapısal programlamanın* esaslarından biridir. Yapısal programlama, nesne yönelimli programlamanın ortaya çıkışına kadarki on yıllar boyunca programlama yapısını etkilemiş olan, bir ölçüde üstünkörü tanımlanmış bir disiplindir.

Yapısal Programlamada Karşılaşılan Problemler

Programlar sürekli büyüyüp daha karmaşıklaştıkça, yapısal programlama yaklaşımı da zorlanmaya başlar. Program geliştirilirken yaşanılanlarla ilgili anlatılan dehşet hikayelerini duymuş olabilirsiniz; hatta böyle bir hikayenin içinde yer almış bile olabilirsiniz. Proje fazlasıyla karmaşıktır, planların gerisinde kalınmıştır, programcı sayısı artırılır, karmaşıklık daha da artar, maliyet hızla yükselir, planların daha da gerisinde kalınır ve nihayet felaket kaçınılmazdır.

Bu başarısızlıkların ardındaki nedenler incelendiğinde asıl zayıflığın prosedürel yaklaşımın kendisinden kaynaklandığı görülür. Yapısal programlama ne kadar iyi gerçekleştirilirse gerçekleştirilsin, büyük hacimli programlar haddinden fazla karmaşık hale gelir.

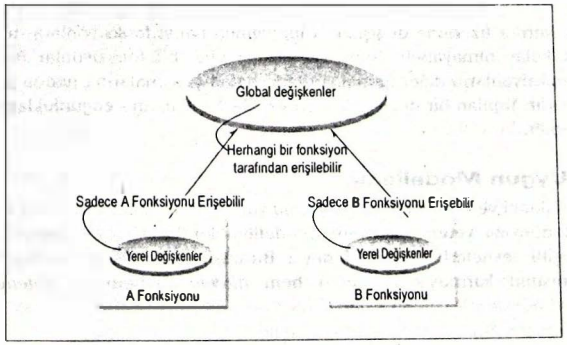
Prosedürel dillerde yaşanan bu problemlerin nedenleri nelerdir? Konuyla ilgili iki problem vardır. Birincisi, fonksiyonların global verilere erişimlerinin sınırsız olmasıdır. İkincisi de, prosedürel yaklaşımın temeli olan iki husus olan ilişkisiz fonksiyon ve verilerin, gerçek dünyanın modellenmesinde yetersiz olmalarıdır.

Bu problemleri bir envanter programı örneğinde inceleyelim. Böyle bir programda, envanterdeki mal yığını önemli bir global veridir. Yeni malın envantere girilmesi, gösterimi, mal ile ilgili değişiklik yapılması ve benzeri işlemler için çeşitli fonksiyonların bu veriye erişmesi gerekir.

Sınırsız Erişim

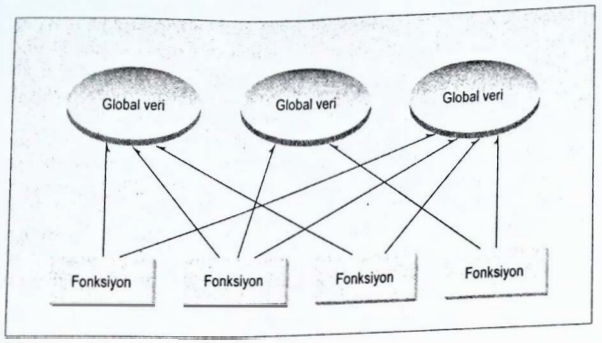
Prosedürel bir programda, mesela C dilinde yazılmış olsun, iki çeşit veri vardır. *Yerel veri* (*local data*), bir fonksiyon içinde gizlidir ve yalnızca bu fonksiyon tarafından kullanılır. Envanter programında bir gösterim fonksiyonu, ekranda hangi malı gösterdiğini hatırlaması amacıyla yerel veri kullanabilir. Yerel veri fonksiyonun işleviyle yakından ilgilidir ve diğer fonksiyonların yapabileceği değişikliklerden etkilenmez.

Ancak, iki veya daha fazla fonksiyon aynı veriye erişmek zorunda ise –ve bu durum programın en önemli verileri için geçerliyse- o halde bu verinin, örneğimizdeki envanterdeki mal yığını gibi, *global* olarak tanımlanması gereklidir. Global veri, program içindeki *tüm* fonksiyonlar tarafından erişilebilir. (Fonksiyonların modül olarak gruplanmasını bu noktada dikkate almıyoruz, çünkü bu, konumuzu önemli ölçüde etkilemiyor.) Prosedürel bir programdaki yerel ve global değişkenler arasındaki ilişki Şekil 1.1'de gösterilmiştir.



ŞEKİL 1.1: Global ve yerel değişkenler.

Büyük hacimli bir programda bir çok fonksiyon ve bir çok global veri mevcuttur. Bu durumun Şekil 1.2'de gösterildiği gibi fonksiyonlar ve veriler arasında daha da fazla sayıda potansiyel bağlantıya neden olması prosedürel yöntemlerde karşılaşılan bir problemdir.



ŞEKİL 1.2: Prosedürel yaklaşım.

Bu çok sayıdaki bağlantı, birkaç açıdan probleme neden olur. Birincisi; bu durum programın kavramsal olarak modellenmesini zorlaştırır. İkincisi de; bu durum program üzerinde değişiklik yapılmasını güç hale getirir. Global bir veri üzerinde yapılan bir değişiklik, bu veriye erişen bütün fonksiyonların yeniden yazılmasını zorunlu kılabilir.

Örneğin, envanter programımızda, bir programcı envanter mallarının ürün kodlarını 5 basamaktan 12 basamağa çıkarmaya karar verebilir. Bu durumda veri tipinin **short**'tan **long**'a çevrilmesi gerekebilir.

Şimdi bu veri üzerine işlem yapan tüm fonksiyonların **short** yerine **long** ile işlem görecek şekilde değiştirilmesi zorunludur. Bu, mahallenizdeki süpermarkette ekmeğin dördüncü koridordan yedinci koridora aktarılması neticesinde karşılaşılan durumun bir benzeridir. Marketin sürekli müşterilerinin bu yeni durumu fark edip, alışveriş alışkanlıklarını ona göre ayarlamaları gerekir.

Büyük programlarda, veriler üzerinde değişiklik yapıldığında hangi fonksiyonların bu verilere eriştiğini söylemek kolay olmayabilir. Bunu saptasanız bile, bu fonksiyonlar üzerinde yapılacak değişiklikler fonksiyonların diğer global verilerle hatalı çalışmalarına neden olabilir. Herşey herşeyle bağlantılıdır. Yapılan bir değişiklik, ucu uzaklara dokunan - çoğunlukla planlamamış - sonuçlar doğurabilir.

Gerçek Dünyaya Uygun Modelleme

Prosedürel yöntemlerdeki ikinci ve en önemli problem, bu yöntemlerin farklı veri ve fonksiyonları düzenlerken gerçek dünyayı yetersiz biçimde modellemeleridir. Fiziksel dünyada biz; insanlar, arabalar gibi çeşitli nesnelerle temas içindeyiz. Bu nesneler ne veriye benzer ne de fonksiyona. Gerçek dünyanın karmaşık nesneleri hem *niteliklere* hem de *yeteneklere (davranışlara)* sahiptir.

Nitelikler

Göz rengi ve mesleki unvan insanlar için örnek olarak verilebilecek niteliklerdendir (kimi zaman niteliklere *özellikler* de denir.) Arabalar için ise beygir gücü ve kapı sayısı bu tür niteliklerdendir. Görüldüğü gibi gerçek dünyadaki nitelikler programdaki verilere karşılık geliyor: Her biri belirli bir değere sahiptir. Söz gelişi, mavi (göz rengi için) veya dört (kapı sayısı için).

Davranış

Davranış, gerçek dünyaya ait bir nesnenin bir etkiye karşılık gösterdiği tepkidir. Patronunuzdan zam istediğinizde, patronunuz sizi çoğunlukla evet veya hayır diye yanıtlar. Araba kullanırken frene basarsanız, araba çoğunlukla durur. Bu örneklerdeki patronun cevabı veya arabanın durması, birer davranış örneğidir. Davranış bir fonksiyon gibidir: Bir şey yapması için bir fonksiyon çağırırsınız (mesela, envanter dökümü için) ve fonksiyon bu işlemi yapar.

Sonuç olarak, ne veri ne de fonksiyonlar gerçek dünya nesnelerinin modellenmesi için tek başlarına yeterli değildir.

Nesne Yönelimli Yaklaşım

Nesne yönelimli dillerin ardındaki ana fikir *verileri* ve *bu veriler üzerinde işlem yapan fonksiyonları* tek bir birim içinde birleştirmektir. Bu birim *nesne (object)* olarak adlandırılır.

Bir nesnenin fonksiyonları, bu nesnenin verilerine erişmeyi sağlayan tek yoldur. C++ dilinde bu fonksiyonlar *üye fonksiyonlar* olarak adlandırılır. Bir nesnenin içindeki bir veriyi okumak istiyorsanız, bu nesnenin içindeki bir üye fonksiyonu çağırırsınız. Bu üye fonksiyon veriye erişip, değerini size döndürür. Siz veriye doğrudan ulaşamazsınız. Veriler *gizlenmiştir*, böylece kazara meydana gelebilecek değişikliklerden korunmuş olurlar. Veriler ve ilgili fonksiyonlar tek bir birim içine *paketlenmişlerdir*. *Verilerin paketlenmesi* (*encapsulation*) ve *veri gizliliği* (*data hiding*) nesne yönelimli dillerin tanımlanmasında kullanılan temel terimlerdir.

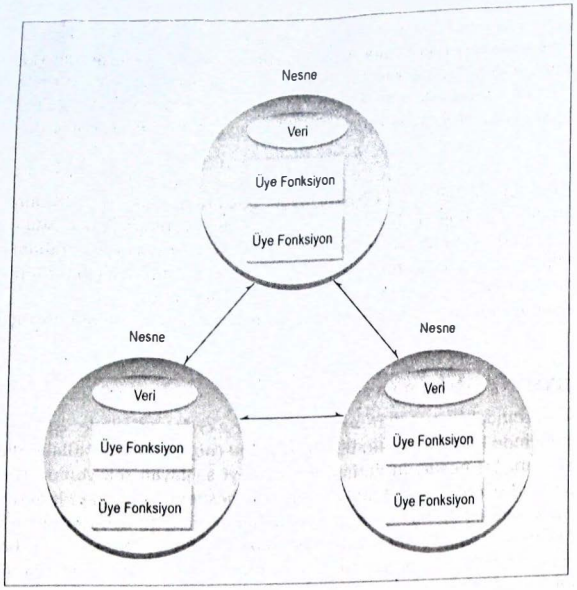
Bir nesneye ait verileri değiştirmek istediğinizde, bu verilerle etkileşen fonksiyonların neler olduğunu tam olarak bilmeniz gerekir: Nesnenin üye fonksiyonları. Diğer hiçbir fonksiyon bu verilere erişemez. Bu, kodlamayı, hataları ayıklamayı ve programın bakımını kolaylaştırır.

Bir C++ programı tipik olarak birkaç tane nesne içerir. Bu nesneler diğerlerinin üye fonksiyonlarını çağırarak birbirleriyle iletişim kurarlar. Şekil 1.3'te bir C++ programının organizasyonu görülüyor.

Şunu vurgulamamız gerekir: C++'ta üye fonksiyon olarak ifade edilen fonksiyonlar diğer nesne yönelimli dillerde *yöntem (method)* olarak adlandırılır (mesela, ilk nesne yönelimli programlama dili olan Smalltalk'ta böyledir.) Ayrıca, verilere de *nitelikler* (*attributes*) veya *örnek değişkenler* (*instance variables*) denilmektedir. Bir nesnenin üye fonksiyonlarını çağırmak bu nesneye *mesaj göndermektir*. Bu terimler C++ terminolojisinin resmi terimleri değildir ama, özellikle nesne yönelimli tasarımlarda artan sıklıkla kullanılmaya başlanmıştır.

Bir Benzetme

Nesneleri bir şirket içindeki bölümler olarak düşünebilirsiniz: Söz gelişi, satış, muhasebe, insan kaynakları bölümleri gibi. Bölümler, kurumsal organizasyonlara önemli katkılar sağlar. Çok küçükleri hariç pek çok şirkette insanlar bir gün personel problemleriyle, diğer gün maaşlarla uğraşıp sonra da satışa çıkmazlar. Her bölümün, görevleri açıkça belirlenmiş kendi personeli vardır. Ayrıca her bölüm kendi verilerine sahiptir: Muhasebe bölümü maaş bilgilerini, satış bölümü satış rakamlarını, personel bölümü de çalışanların kayıtlarını tutar.

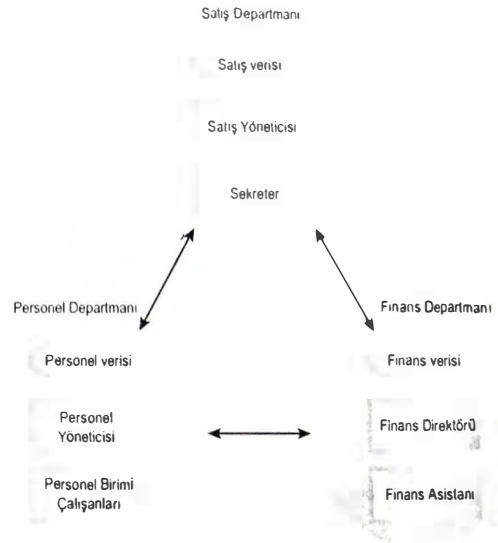


ŞEKİL 1.3: Nesne yönelimli yaklaşım.

Bölümlerdeki insanlar kendi bölümlerinin verilerini denetler ve bu veriler üzerinde işlem yaparlar. Şirketi bölümlere ayırmak, şirket faaliyetlerini anlamayı ve denetlemeyi kolaylaştırır; ayrıca şirket bilgilerinin entegrasyonunun sürekliliğine de yardımcı olur. Muhasebe bölümü, mesela, maaş verilerinden sorumludur. Eğer siz satış müdürüyseniz ve Temmuz ayında güney bölgesinde yapılan maaş ödemelerinin yekûnünü merak ediyorsanız, muhasebe bölümüne gidip dosya dolaplarının altını üstüne getirmezsiniz. Muhasebe bölümündeki ilgili bir kişiye bir not gönderir; daha sonra görevli kişinin veriye erişip, istediğiniz bilgiyi size göndermesini beklersiniz. Bu, veriye hatasız erişilmesini garanti eder ve verinin izinsiz erişimlerle bozulmasını önler. Kurumsal organizasyonlarla ilgili bu bakış açısı Şekil 1.4'te gösterilmiştir. Aynı şekilde, nesneler de, program verilerinin bütünlüğünü korumaya yardımcı olur; aynı zamanda programları organize etmenin yollarından biridir.

Nesne Yönelimli Programlama: Düzenli Programlara Doğru

Şunu aklınızdan çıkarmayın: Nesne yönelimli programlama aslında programın çalışma ayrıntılarıyla ilgilenmez. Onun yerine, programın genel organizasyonuyla ilgilenir. C++'taki bir çok program ifadesi prosedürel dillerdeki ifadelere benzer, pek çoğu da C'deki ifadelerin aynısıdır. Gerçekten, bir C++ programındaki bir üye fonksiyonun tümü, C'deki bir prosedürel fonksiyona çok benziyor olabilir. Bir ifade veya fonksiyonun prosedürel bir C programının veya nesne yönelimli C++ programının parçası olup olmadığı ancak daha genel açıdan bakıldığında görülebilir.



ŞEKİL 1.4: Kurumsal organizasyon örneği.

Nesne Yönelimli Dillerin Özellikleri

Şimdi, genel olarak nesne yönelimli dillerin, özel olarak da C++'ın birkaç temel öğesine bir göz atalım.

Nesneler

Nesne yönelimli bir dilde programlama problemini çözmeniz gerektiğinde, artık, problemin fonksiyonlara nasıl bölüneceğini değil de nesnelere nasıl bölüneceğini düşünmeniz gerekir. Fonksiyonlar yerine nesneler cinsinden düşünmenin, programların kolaylıkla tasarlanmasında şaşırtıcı bir etkisi vardır. Bu, programlardaki nesneler ile gerçek dünyadaki nesnelerin birbiriyle yakından eşleşmesinden kaynaklanır. Bu konu, "Nesne Yönelimli Yazılım Geliştirme" başlıklı 16. Bölümde ayrıntılı olarak anlatılmıştır.

Nesne yönelimli programlarda neler nesne olarak tanımlanabilir? Bu sorunun cevabı hayal gücünüzle sınırlıdır. Ancak sizi düşünmeye sevk etmesi açısından bazı tipik kategoriler şöyle sıralanabilir:

- **Fiziksel nesneler**
  - Trafik akışı simülasyonundaki otomobiller
  - Devre tasarımı programındaki elektriksel bileşenler
  - Ekonomik bir modeldeki ülkeler
  - Hava kontrol sistemindeki hava taşıtları

- **Bilgisayar kullanımındaki öğeler**
  - Pencereler
  - Menüler
  - Grafik nesneler (çizgiler, dikdörtgenler, çemberler)
  - Fare, klavye, disk sürücüleri, yazıcı
- **Veri yedekleme yapıları**
  - Özelleştirilmiş diziler
  - Yığınlar
  - Bağlı listeler
  - İkili ağaçlar
- **İnsani değerler**
  - Çalışanlar (personel)
  - Öğrenciler
  - Müşteriler
  - Satış elemanları
- **Veri Yığınları**
  - Bir envanter
  - Bir personel dosyası
  - Bir sözlük
  - Dünyadaki şehirlerin enlem ve boylam bilgilerini saklayan bir tablo
- **Kullanıcı tanımlı veri tipleri**
  - Zaman
  - Açılar
  - Karmaşık sayılar
  - Düzlem üzerindeki noktalar
- **Bilgisayar oyunlarındaki bileşenler**
  - Oto yarışındaki otomobiller
  - Oyun tahtası üzerindeki pozisyonlar (satranç, dama)
  - Bir ekolojik simülasyondaki hayvanlar
  - Macera oyunlarındaki rakipler ve arkadaşlar

Programlama nesneleri ve gerçek dünya nesnelerinin eşlenebiliyor olması, veriler ve fonksiyonların birleştirilebilmesinin bir sonucudur. Ortaya çıkan nesneler, program tasarımında bir devrim niteliğindedir. Prosedürel dillerde, programlama yapıları ve modellenen birimler arasında böyle bir yakın eşleşme söz konusu değildir.

Sınıflar

Nesne yönelimli programlamada, nesneler *sınıfların* (*class*) birer üyesidir. Bu ne anlama gelir? Benzer bir örnek verelim. Hemen hemen tüm bilgisayar dillerinde standart veri tipleri mevcuttur. Söz gelişi, tamsayı anlamında bir veri tipi olan **int**, C++'ta standart olarak tanımlıdır ("Döngüler ve Kararlar" başlıklı 3. Bölümde bu veri yapılarını göreceğiz.) Programınızda **int** tipinde istediğiniz sayıda değişken tanımlayabilirsiniz:

```
int gun;
int sayac;
```



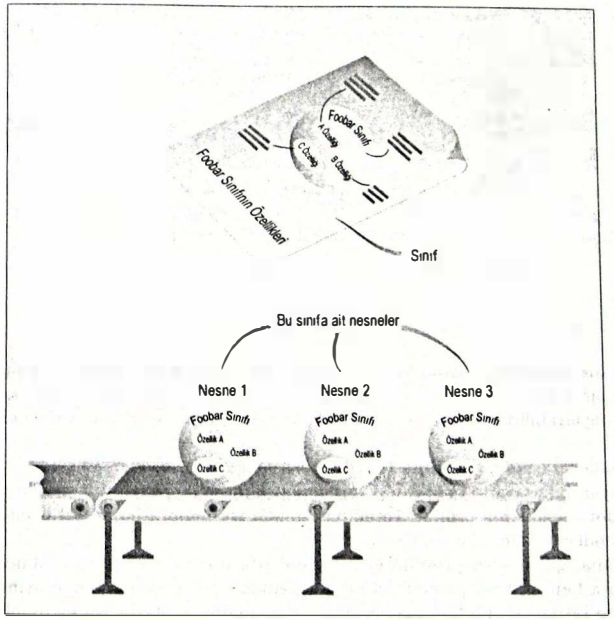
```
int bolen;
int cevap;
```

Benzer biçimde, Şekil 1.5'te gösterildiği gibi, aynı sınıfa ait bir çok nesne tanımlayabilirsiniz. Bir sınıf adeta bir plan gibidir. Sınıf, o sınıfın nesneleri içinde hangi verilerin ve fonksiyonların bulunduğunu belirtir. Bir sınıfın tanımlanması, o sınıf tipinde nesnelerin oluşturulması anlamına gelmez. Nasıl ki, veri tipi olarak **int** tek başına bir değişken oluşturulmasını sağlayamıyorsa, sınıf için de durum aynıdır.

Yani, bir sınıf, bir dizi benzer nesnenin genel tanımıdır. Bu, *sınıf* sözcüğünün teknik olmayan tanımıyla da örtüşür. Prince, Sting ve Madonna rock müzisyenleri sınıfının birer üyesidir. "Rock müzisyeni" adında tek bir kişi yok ama; bazı özelliklere sahip, belirli isimleri olan belirli kişiler bu sınıfın üyeleridir. Nesneler genellikle sınıfların birer "örnek kopyası" (instance) olarak adlandırılırlar.

Kalıtım

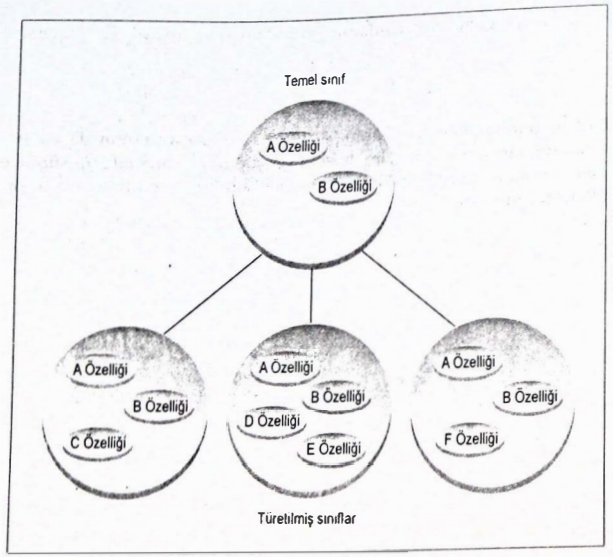
Sınıf kavramı bizi *kalıtım* (*inheritance*) kavramına götürür. Günlük yaşantımızda alt sınıflara ayrılmış sınıf kavramlarını kullanırız. Mesela, hayvanlar aleminin memeliler, sürüngenler, böcekler, kuşlar vs alt sınıflarına ayrıldığını biliyoruz. Taşıt sınıfı da araba, kamyon, otobüs, motosiklet vs alt sınıflarına ayrılmıştır.



ŞEKİL 1.5: Bir sınıf ve nesneleri.

Bu tür bir bölünmedeki temel prensip, her sınıfın kendisinden türetilen alt sınıfla ortak özellikler paylaşıyor olmasıdır. Araba, kamyon, otobüs ve motosikletlerin hepsinin tekerlekleri ve motorları vardır; bunlar taşıtları tanımlayan özelliklerdir. Sınıfın diğer üyeleriyle paylaşılan özelliklere ek olarak, her altsınıfın ayrıca kendisine özgü özellikleri de mevcuttur: Otobüslerin, örneğin, çok sayıda kişinin oturabilmesi için koltukları, kamyonların ise ağır yükleri taşımak için kasaları vardır.

Şekil 1.6, kalıtım kavramını gösterir. Temel sınıfın birer parçası olan A ve B'nin, türetilmiş sınıfların hepsinde ortak olduğuna dikkat edin. Ancak, türetilmiş sınıfların her biri kendisine ait özellikler de içerirler.



ŞEKİL 1.6: Kalıtım.

Aynı şekilde, nesne yönelimli programlamada kullanılan bir sınıf, birkaç tane alt sınıfın ebeveyni (parent) olabilir. C++'ta orijinal sınıf *temel sınıf* olarak adlandırılır. Diğer sınıflar, temel sınıfın özelliklerini paylaşabilirler, ancak kendi özelliklerini de ekleyebilirler. Bu sınıflara *türetilmiş sınıflar* denir.

Nesneler ve sınıflar arasındaki ilişki ile temel sınıf ve türetilmiş sınıflar arasındaki ilişkiyi birbirine karıştırmayın. Bilgisayarın belleğinde yer kaplayan nesneler, adeta bir şablon gibi ait oldukların sınıfın tüm özelliklerini taşırlar. Türetilmiş sınıflar ise temel sınıfın bazı özelliklerini taşıyor olsalar da kendi özelliklerini de ekleyebilirler.

Kalıtım bir bakıma, geleneksel prosedürel programlarda programın basitleştirilmesi için fonksiyon kullanmaya benzer. Eğer prosedürel bir programda neredeyse tamamen aynı işi yapan üç farklı bölüme rastlarsak, bu bölümlerin ortak elemanlarını saptayıp, bu elemanları tek

bir fonksiyon içine yerleştirme şansını yakalamış oluruz. Programın bu üç bölümü ortak işlemlerin yürütülmesi için bu fonksiyonu çağırırlar, diğer yandan kendi işlemlerini de gerçekleştirirler. Aynı şekilde, bir temel sınıf da bir grup türetilmiş sınıfta ortak olan elemanlar içerir. Prosedürel programlardaki fonksiyonlar gibi, kalıtım da nesne yönelimli bir programın uzunluğunu azaltır ve program öğeleri arasındaki ilişkileri netleştirir.

Yeniden Kullanılabilirlik

Bir sınıf yazılıp, oluşturulduktan ve hataları giderildikten sonra, diğer programcılara kendi programlarında kullanmaları için dağıtılabilir. Buna *yeniden kullanılabilirlik* denir. Bu, prosedürel bir dildeki kütüphane fonksiyonlarının değişik programlara dahil edilmesine benzer.

Bununla birlikte, nesne yönelimli programlamada kalıtım kavramı, yeniden kullanılabilirlik fikrini önemli ölçüde genişletir. Bir programcı mevcut bir sınıfı alır, bu sınıf üzerinde değişiklik yapmadan ona ilave özellikler ve beceriler ekleyebilir. Bu, mevcut bir sınıftan yeni bir sınıf türeterek gerçekleştirilir. Yeni sınıf eskisinin becerilerine sahiptir ancak, kendisine ait yeni özellikler de ekleyebilir.

Örneğin, Windows veya diğer GUI'lerdeki (Grafik Kullanıcı Arayüzü) gibi bir menü sistemi oluşturan bir sınıf yazdığınızı (veya başkasından satın aldığınızı) varsayalım. Bu sınıf düzgün çalışır ve üzerinde değişiklik yapmanıza gerek yoktur. Fakat diyelim ki, bazı menü elemanlarının yanıp sönmesini istiyorsunuz. Bunu yapmak için mevcut sınıfın tüm özelliklerini taşıyan, ayrıca yanıp sönme özelliğini de içeren yeni bir sınıf oluşturmanız yeterlidir.

Mevcut yazılımın yeniden kullanılmasının getirdiği kolaylık, nesne yönelimli programlamanın sağladığı önemli bir avantajdır. Pek çok şirket, mevcut sınıfları ikinci bir projede yeniden kullanabilmenin, orijinal programlama yatırımlarının getirisini artırdığının farkındadır. Sonraki bölümlerde bu konuyla ilgili daha fazla konuşacağız.

Yeni Veri Tipleri Oluşturmak

Programcıya yeni veri tipleri oluşturmak için elverişli bir yöntem sunmaları, nesnelerin sağladığı bir avantajdır. Varsayalım ki, programınızda, x-y koordinatları veya enlem-boylam gibi iki boyutlu konumlarla çalışıyorsunuz. Bu konumsal değerler üzerinde gerçekleştireceğiniz işlemleri normal aritmetik işlemleri kullanarak ifade etmek istediğinizi düşünelim. Örneğin,

```
pozisyon1 = pozisyon2 + orijin
```

Burada **pozisyon1**, **pozisyon2** ve **orijin** değişkenlerinin her biri, birbirinden bağımsız nümerik değerleri gösterir. Bu iki değeri kapsayan bir sınıf oluşturarak ve **pozisyon1**, **pozisyon2** ve **orijin**'i bu sınıfın nesneleri olarak tanımlayarak, aslında yeni bir veri tipi tanımlamış oluruz. C++'ın birçok özelliği, bu şekilde yeni veri tiplerinin oluşturulmasını kolaylaştırmak amacına yöneliktir.

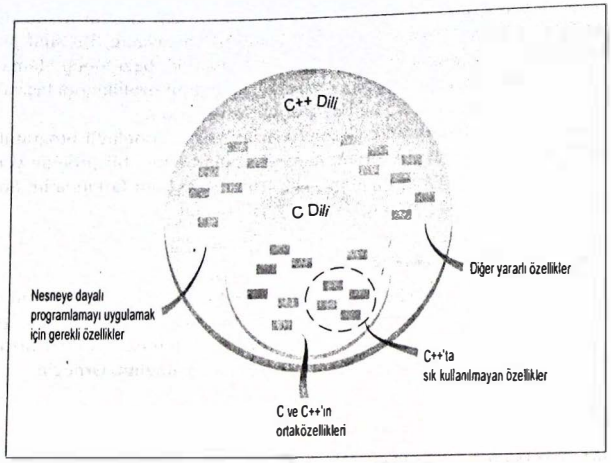
Çok Biçimlilik ve Operatörlerin Aşırı Yüklenmesi

Yukarıdaki aritmetik örneğinde kullanılan = (eşittir) ve + (artı) operatörlerinin, **int** gibi standart veri tiplerinin kullanıldığı işlemlerdeki gibi çalışmadığına dikkat edin. **Pozisyon1** ve diğerleri C++'ta önceden tanımlı nesneler değildir. Bunlar programcının tanımladığı **Pozisyon** sınıfının nesneleridir. Peki bu durumda = ve + operatörleri bu nesneler üzerinde işlem yapacaklarını nereden biliyorlar? Yanıt: Bu operatörler için yeni davranışlar tanımlayabiliriz. Bu işlemler **Pozisyon** sınıfının üye fonksiyonları olacaklardır.

Operatör ve fonksiyonları işlevlerine bağlı olarak değişik şekillerde kullanmaya *çok biçimlilik* (*polymorphism*) (birkaç tane farklı forma sahip olan şey) denir. Mevcut bir operatörün (mesela + veya =), yeni bir veri tipinde işlev görebilme becerisi, operatörün *aşırı yüklenmesi* (*overloading*) olarak ifade edilir. Operatörün aşırı yüklenmesi bir tür çok biçimliliktir; ayrıca nesne yönelimli programlamanın önemli bir özelliğidir.

C++ ve C

C++, C dilinden türetilmiştir. Tam manasıyla söylemek gerekirse, C++, C'nin üst kümesidir; C'deki hemen hemen hatasız her ifade C++'ta da doğru bir ifadedir. Ancak bunun tersi doğru değildir. Sınıflar, nesneler ve nesne yönelimli programlama, C++'ın ortaya çıkışında C'ye eklenen en önemli özellikleri oluştururlar. (C++ önceleri "Sınıflı C" olarak anılmaktaydı.) Bununla beraber, C++'ın yeni bir çok özelliği daha mevcuttur. Giriş/çıkışlar (I/O) için geliştirilmiş bir yöntem ve açıklamaları yazmanın yeni bir yolu, bu özellikler arasında yer alır. Şekil 1.7, C ve C++ arasındaki ilişkiyi gösterir.



ŞEKİL 1.7: C ve C++ arasındaki ilişki.

Aslında C ve C++ arasındaki pratik farklar tahmin edebileceğinizden çok daha fazladır. C++ programınızı, C'de yazılmış bir programı andırır biçimde yazabiliyor olabilirsiniz. Ancak bunu hemen hemen hiç kimse yapmaz. C++ programcıları, C++'ın yeni özelliklerinden yararlanmanın yanı sıra, geleneksel C özelliklerini de C programcılarına kıyasla daha farklı boyutlarda kullanarak, bunların da önemini vurgularlar.

C'yi önceden biliyorsanız, C++'a bir adım önden başlıyorsunuz demektir (eğer öğrendiklerinizi unutmak gibi kötü huylarınız yoksa, tabii.) Ancak konunun büyük bir bölümü yenidir.

Esaslar Üzerine

Amacımız, nesne yönelimli programlama örneklerini yazmaya mümkün olduğunca erken başlamanıza yardımcı olmaktır. Bununla birlikte, önceden de bahsettiğimiz gibi, C++'ın büyük bir kısmı C'den aktarılmıştır. C++'ın genel yapısı nesne yönelimli olmasına karşın programlama çıkmazlarından kurtulmak için bazı eski moda prosedürel özellikleri de bilmeniz gerekebilir. Bu nedenle, 2. ve 5. arasındaki bölümlerde C++'ın -pek çoğu C'de mevcut olan- "geleneksel" özellikleri incelenmiştir. Bu bölümlerde değişkenler ve I/O hakkında, döngü ve kararlar gibi denetim yapıları hakkında ve fonksiyonlar hakkında bilgi edineceksiniz. Ayrıca yapılar hakkında da bilgi sahibi olmuş olacaksınız, çünkü sınıflarda kullanılan sözdiziminin aynısı yapılarda da kullanılır.

C'yi zaten biliyorsanız bu bölümleri atlamayı düşünebilirsiniz. Yine de, C ve C++ arasında kimi açık kimi güç fark edilen bir çok farklılık olduğunu göreceksiniz. Tavsiyemiz bildiklerinizi gözden geçirerek ve C++'ın C'den farklı olduğu noktalara dikkatinizi vererek bu bölümleri okumanız.

Nesne yönelimli programlamayla ilgili spesifik açıklamalar "Nesneler ve Sınıflar" başlıklı 6. bölümden itibaren başlıyor. O bölümden itibaren verilen örnekler nesne yönelimli programlardan oluşacak.

Birleştirilmiş Modelleme Dili (UML)

UML, bilgisayar programlarını modellemek için kullanılan grafiksel bir "dildir". "Modelleme", bir şeyin sadeleştirilmiş görüntüsünü oluşturmak demektir. Tıpkı bir ev planının evi modellemesi gibi. UML, gerçek kodun ayrıntılarına gömülmeden, programlara daha yüksek seviyeden bakıp, program organizasyonunu gözünüzde canlandırma yollarından biridir.

UML başlangıçta üç ayrı modelleme dili olarak ortaya çıktı. Bunlardan biri, Rational Software'de Grady Booch tarafından; diğeri, General Electric'te James Rumbaugh tarafından, öbürü ise Ericsson'da Ivar Jacobson tarafından geliştirildi. Sonunda Rumbaugh ve Jacobson, Rational'da Booch'a katıldılar. Bu üçlü Rational'da üç amigolar olarak tanındı. 1990'ların sonlarında kendi modelleme dillerini birleştirerek (dilin ismi buradan geliyor) Birleştirilmiş Modelleme Dilini oluşturdular. Ortaya çıkan ürün, endüstri standartları geliştiren şirketlerin oluşturduğu bir konsorsiyum olan OMG (Object Management Group) tarafından kabul edildi.

UML'e neden ihtiyaç duyuyoruz? Büyük bilgisayar programlarında, sadece koda bakarak programın hangi parçalarının birbiriyle ilintili olduğunu anlamak çoğunlukla zordur. Gerçekten, nesne yönelimli programlama, prosedürel programlar üzerine büyük bir ilerlemedir. Bununla birlikte, bir programın ne yapması gerektiğini anlamak -en iyimser yaklaşımla- program listeleri üzerinde sıkı bir çalışma gerektirir.

Koddaki problem, kodun çok ayrıntılı olmasından kaynaklanır. Programa tepeden bakabilen ve programın ana parçalarını saptayıp, bunların birlikte nasıl çalıştıklarını gösteren bir mekanizma olsa, hiç fena olmazdı. İşte, UML bu ihtiyaca cevap verir.

UML'in en önemli parçasını değişik türde şemalar oluşturur. Sınıf şemaları, sınıflar arasındaki ilişkileri; nesne şemaları, spesifik nesneler arasındaki ilişkileri; sekans şemaları, zaman içinde nesneler arası iletişimi; use case şemaları, program kullanıcılarının programla nasıl etkileşebileceklerini gösterir. Bu şemalar, bir programa ve onun işleyişine çok çeşitli açılardan bakma imkanı sağlar.

UML, bir programın nasıl çalıştığını anlamamıza yardımcı olmanın yanı sıra daha pek çok rol üstlenebilir. 16. bölümde de göreceğimiz gibi, UML bir programın ilk tasarımında da yar-

dımcı olabilir. Aslında UML, yazılım geliştirme sürecinin tüm aşamalarında, başlangıç spesifikasyonlarından, dokümantasyona, test ve bakıma kadar bütün aşamalarda işe yarar.

UML, bir yazılım geliştirme yöntemi değildir. Geliştirme sürecinin aşamalarını belirleyen böyle bir çok yöntem mevcuttur. UML sadece geliştirilmekte olan yazılıma bir bakış şeklidir. UML, her cins programlama diline uygulanabilir olsa da, özellikle nesne yönelimli programlama için daha uygundur.

Giriş bölümünde de bahsettiğimiz gibi, kitabın akışı içinde UML'in belirli özelliklerini tanıtacağız.

- Bölüm 1: (Bu ayrım) UML'e giriş
- Bölüm 8: Sınıf şemaları, ilişkiler ve ilişkilerin yönü
- Bölüm 9: Genelleme, toplama ve düzenleme
- Bölüm 10: Durum şemaları ve katlanabilirlik
- Bölüm 11: Nesne şemaları
- Bölüm 13: Daha karmaşık durum şemaları
- Bölüm 14: Şablonlar, bağımlılık ilişkileri ve klişeler (stereotypes)
- Bölüm 16: Use case'ler, use case şemaları, faaliyet şemaları ve sekans şemaları

Özet

Nesne yönelimli programlama, programları organize etme yollarından biridir. Önemli olan kodlama ayrıntıları değil, programların tasarlanış şeklidir. Nesne yönelimli programlar nesneler yardımıyla organize edilir. Nesneler hem veri hem de bu veriler üzerinde işlev gören fonksiyonlar içerir. Bir sınıf ise birkaç tane nesne için şablon görevi görür.

Kalıtım, mevcut bir sınıftan, bu sınıfı değiştirmeden başka bir sınıf türetmeye imkan verir. Türetilmiş sınıf, üst sınıfın tüm veri ve fonksiyonlarını içerir, ayrıca kendisine yenilerini de ekleyebilir. Kalıtım bir sınıfın yeniden kullanılmasını mümkün kılar, yani değişik programlarda defalarca kullanılmasına imkan verir.

C++, C'nin bir üst kümesidir. C++, C diline nesne yönelimli programlama becerisi katar. Ayrıca bir çok başka özellik daha ilave eder. Bununla birlikte, C++ ile programcıların odağı değişmiştir. Şöyle ki, C'nin sık kullanılan bazı özellikleri C++'ta olmasına rağmen bunlar pek ender kullanılır, diğer özellikler ise sıklıkla tercih edilir. Sonuçta C++, şaşırtıcı ölçüde farklı bir dildir.

Birleştirilmiş Modelleme Dili (UML), programın yapısını ve işlevini şemalarla gözünüzde canlandırmayı sağlayan standart bir yöntemdir.

Bu bölümde bahsedilen genel kavramlar C++'ı ayrıntılarıyla öğrendikçe daha da anlaşılır hale gelecektir. Kitapta ilerlerken geri dönüp bu bölüme başvurmak isteyebilirsiniz.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz. Kitabın içindeki tüm seçmeli soruların birden fazla doğru cevabı olabileceğine dikkat edin.

1. Pascal, BASIC ve C p_____dillerdir, C++ n_____ bir dildir.
2. Evin planı evi modeller, nesne _____ modeller.
   a. üye fonksiyonu
   b. sınıfı
   c. operatörü
   d. veriyi

3. Bir nesnenin iki temel bileşeni _____ ve onun üzerinde _____ fonksiyonlardır.
4. C++'ta bir sınıfın içindeki bir fonksiyona _____ denir.
   a. üye fonksiyon
   b. operatör
   c. sınıf fonksiyonu
   d. yöntem
5. Verileri izinsiz erişimden koruma işlevine _____ denir.
6. Aşağıdakilerden hangisi nesne yönelimli bir dil kullanmak için iyi bir sebeptir?
   a. Kendi veri tiplerinizi tanımlayabilirsiniz.
   b. Prosedürel dillere kıyasla program ifadeleri daha kısadır.
   c. Nesne yönelimli bir dil kendi hatalarını düzeltecek şekilde öğretilebilir.
   d. Kavramsal bazda düşünme nesne yönelimli dillerde daha kolaydır.
7. _____ gerçek dünyadaki olguları fonksiyonlara kıyasla daha gerçeğe yakın modeller.
8. Doğru/Yanlış: Bir C++ programı kodlama ayrıntıları haricinde C programının aynısıdır.
9. Veri ve fonksiyonları biraraya getirip paketleme işlemine _____ denir.
10. Bir dilin yeni veri tipleri üretebilme becerisi varsa, bu dil
    a. eleştirilebilir.
    b. paketlemiştir.
    c. aşırı yüklenmiştir.
    d. genişletilebilirdir.
11. Doğru/Yanlış: Herhangi iki satıra bakarak bir programın C veya C++ dilinde yazıldığını kolaylıkla anlayabilirsiniz.
12. Bir fonksiyon veya operatörün değişik veri tiplerini değişik işlevlerde kullanabilmesi becerisine _____ denir.
13. Yeni tanımlanmış veri tipleri üzerinde özel işlev gören normal bir C++ operatörüne _____ operatör denir.
    a. yüceltilmiş
    b. paketlenmiş
    c. sınıflandırılmış
    d. aşırı yüklenmiş
14. C++'ta kullanılan yeni terimleri ezberlemek
    a. kritik öneme sahiptir.
    b. sonraya bırakabileceğiniz bir iştir.
    c. servete ve başarıya götüren yoldur.
    d. tamamen önemsizdir.
15. Birleştirilmiş Modelleme Dili
    a. fiziksel modeller kuran bir programdır.
    b. bir programın organizasyonunu gözden geçirmeye yarayan bir yöntemdir.
    c. C++ ve FORTRAN'ın birleştirilmiş halidir.
    d. yazılım sistemleri geliştirmek için yararlıdır.

İKİNCİ BÖLÜM

C++ PROGRAMLAMA TEMELLERİ

Başlarken
Temel Program Yapısı
cout Kullanarak Çıktı Almak
Direktifler
Açıklamalar
Tamsayı Değişkenler
Karakter Değişkenleri
cin ile Giriş Almak
Kayan Noktalı Tipler
bool Veri Tipi
setw Manipülatörü
Değişken Tiplerinin Özeti
Tip Dönüşümü
Aritmetik Operatörler
Kütüphane Fonksiyonları

Hangi dilde olursa olsun, çok basit bir program yazmak için bile, dile ait bazı esasları bilmeniz gerekir. Bu bölümde, bu tür üç temel özellik tanıtılmıştır: Temel program yapısı, değişkenler ve giriş/çıkış (I/O). Ayrıca dilin diğer özelliklerinden, mesela açıklamalar, aritmetik operatörler, artırma operatörü, veri dönüşümleri ve kütüphane fonksiyonlarından da kabaca bahsedilmiştir.

Bu konular kavramsal olarak zor değildir ama, BASIC veya Pascal gibi dillerle kıyasladığınızda C++'ın stilini biraz zor bulabilirsiniz. Öğrenmeye başlamadan önce C++ programı size bir programlama dilinden çok bir matematik formülü gibi görünebilir. Bu sizi endişelendirmesin. C++'ı daha yakından tanımaya başlayınca C++ daha az rahatsız edici görünmeye başlayacaktır. Hatta öyle ki, diğer dillerin gereksiz yere süslü ve gereksiz sözlerle dolu olduğunu düşüneceksiniz.

Başlarken

Giriş bölümünde bahsettiğimiz gibi, bu kitapla birlikte Microsoft veya Borland derleyicilerinden birini kullanabilirsiniz. Bu derleyicilerin işleyişiyle ilgili ayrıntıları Ek C ve D'de bulabilirsiniz. (Diğer derleyiciler de kullanılabilir.) Derleyiciler kaynak kodu alır ve çalıştırılabilir bir dosya haline dönüştürür. Bilgisayarınız diğer programları çalıştırırken bu dosyayı da çalıştırabilir. Kaynak dosyalar, bu kitaptaki program listelerine karşılık gelen (.CPP uzantılı) metin dosyalarıdır. Çalıştırılabilir dosyalar .EXE uzantılıdır; derleyicinin içinden çalıştırılabilir veya eğer MS-DOS'a aşinaysanız, doğrudan DOS penceresinden de çalıştırılabilir.

Programlar Microsoft derleyicisiyle veya MS-DOS penceresinden program üzerinde değişiklik gerektirmeksizin çalışırlar. Ancak, eğer Borland derleyicisi kullanıyorsanız, çalıştırmadan önce programları bir parça değiştirmeniz gerekir. Aksi halde çıktı, ekranda göremeyeceğiniz kadar kısa bir süre kalır. Bunun nasıl yapıldığını görmek için Ek D'deki "Borland C++ Builder" bölümünü okumayı ihmal etmeyin.

Temel Program Yapısı

Şimdi çok basit bir C++ programına bir göz atalım. Programın ismi **FIRST**, bu nedenle kaynak dosyanın ismi de **FIRST.CPP**. Bu program sadece ekrana bir cümle yazar:

```
#include <iostream>
using namespace std;
int main()
   {
   cout << "Every age has a language of its own\n";
   return 0;
   }
```

Bu, küçük bir program olmasına rağmen, C++ programlarının nasıl oluşturulduğu hakkında oldukça fazla bilgi içerir. Şimdi bu programı ayrıntılarıyla inceleyelim.

Fonksiyonlar

Fonksiyonlar C++'ın en temel bloklarından biridir. **FIRST** neredeyse sadece **main()** isimli tek bir fonksiyondan ibarettir. Bu programda fonksiyonun parçası olmayan tek kısım **#include** ve **using** ile başlayan ilk iki satırdır.

Bölüm 1'de bahsettiğimiz gibi bir fonksiyon bir sınıfın parçası olabilir. Bu durumda bu fonksiyona *üye fonksiyon* denir. Bununla birlikte, fonksiyonlar sınıflardan bağımsız olarak da bulunabilirler. Sınıflar hakkında konuşmak için henüz hazır değiliz, bu nedenle, şimdilik örnekteki **main()** fonksiyonu gibi kendi başlarına bulunabilen fonksiyonlardan bahsedeceğiz.

Fonksiyon İsmi

main kelimesinin peşinden gelen parantezler bir fonksiyonun ayırt edici özelliklerindendir. Parantezler olmazsa, derleyici **main** kelimesinin bir değişkenle veya başka bir programın öğesiyle ilgili olduğunu düşünürdü. Fonksiyonları anlatırken biz de C++'ta kullanıldığı şekliyle kullanacağız: Fonksiyon isminden sonra parantez koyacağız. Daha sonra bu parantezlerin her zaman boş olmadığını göreceğiz. Parantezler ayrıca fonksiyon *argümanları* için de kullanılır. Argümanlar, fonksiyonu çağıran programın, fonksiyona gönderdiği değerlerdir.

Fonksiyon isminden önce gelen **int** kelimesi, bu fonksiyonun **int** tipinde bir değer döndürdüğüne işaret ediyor. Bunu şu an için dert etmeyin; veri tiplerini bu bölüm içinde, fonksiyonun değer döndürmesini "Fonksiyonlar" başlıklı 5. bölümde öğreneceğiz.

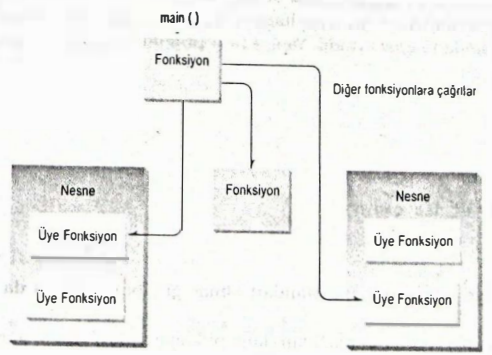
Küme Parantezleri ve Fonksiyonun Gövdesi

Fonksiyonun *gövdesi, küme parantezleri* içine alınır. Bu parantezler diğer dillerdeki **BEGIN** ve **END** anahtar sözcüklerinin yaptığı işi yaparlar: Program ifadelerinden oluşan bir bloğu çevreler ya da *sınırlandırırlar*. Her fonksiyonun gövdesi etrafında bu parantezlerden bir çift bulunmak zorundadır. Örneğimizde fonksiyonun gövdesi içinde yalnızca iki ifade vardır: Bunlar, **cout** ve **return** ile başlayan satırlardır. Aslında bir fonksiyon gövdesi içinde pek çok ifade bulunabilir.

Her Zaman main() ile Başla

Bir C++ programı çalıştırdığınızda, **main()** isimli fonksiyonun başlangıcında yer alan ifade, derleyici tarafından çalıştırılacak ilk ifadedir. (Bu en azından, bu kitaptaki konsol modunda çalışan programlar için geçerlidir.) Programın içinde bir çok fonksiyon, sınıf ve diğer program öğeleri olabilir ama, ilk başta kontrol her zaman **main()**'e gider. Eğer programınızda **main()** isimli bir fonksiyon yoksa, programı çalıştırdığınızda bir hata mesajı ile karşılaşırsınız.

C++ programlarının çoğunda, daha sonra göreceğimiz gibi **main()**, çeşitli nesnelerin üye fonksiyonlarını çağırarak programın asıl yapması gereken işi gerçekleştirir. **main()** fonksiyonundan ayrıca, bir nesneye ait olmayan bağımsız fonksiyonlara da çağrılar yapılır. Bu, Şekil 2.1'de gösterilmiştir.



ŞEKİL 2.1: Nesneler, fonksiyonlar ve **main()**.

Program İfadeleri

Program *ifadeleri* C++ programlarının en temel birimleridir. `FIRST` isimli programda iki tane ifade vardır:

```
cout << "Every age has a language of its own\n";
```

ve

```
return 0;
```

İlk ifade bilgisayara, tırnak içindeki deyimin ekrana yazılmasını bildirir. Bir çok ifade bilgisayara bir şeyler yapmasını bildirir. Bu bağlamda, C++ ifadeleri diğer dillerdeki ifadelerle aynıdır. Aslında, önceden bahsettiğimiz gibi, C++ ifadelerinin büyük çoğunluğu C'deki ifadelerle aynıdır.

Noktalı virgül, ifadenin bittiğini gösterir. Bu, söz diziminin kritik ama unutulması kolay bir parçasıdır. BASIC gibi bazı dillerde satırın sonu aynı zamanda ifadenin de sonu anlamına gelir. Fakat bu, C++ için geçerli değildir. Eğer noktalı virgülü unutursanız, derleyici (her zaman olmasa da) genellikle hata mesajı verir.

Fonksiyon gövdesindeki en son ifade `return 0;` ifadesidir. Bu ifade `main()` fonksiyonuna, kendisini çağırana 0 değerini döndürmesini bildiriyor. Bu durumda `main()` fonksiyonunu çağıran işletim sistemi veya derleyicidir. C++'ın eski sürümlerinde, `main()` fonksiyonunun dönüş tipi `void` olarak belirlenebiliyordu ve böylece `return` ifadesine gerek kalmıyordu. Ancak bu, Standart C++'ta doğru değildir. 5. Bölümde `return` hakkında daha fazla bilgi edineceğiz.

Boşluklar

C++ derleyicisi açısından satır sonunun önemli olmadığını söylemiştik. Gerçekten, derleyici bazı özel karakterlerin hemen hemen hiçbirini dikkate almaz. Boşluk karakteri (space), line-feed (satır atlama), carriage return (paragraf başı), sekme ve düşey sekme karakterleri, *boşluklar* (*whitespace*) denilen karakterleri oluşturur. Bu karakterler derleyici için görünmezdirler. Tek bir satır üzerinde bir çok sayıda boşluk karakteri ve sekmeyle birbirinden ayrılmış birkaç tane ifade yazabilirsiniz veya aynı ifadeyi iki veya daha fazla satıra yayarak çalıştırabilirsiniz. Derleyici açısından hepsi aynıdır. Yani, `FIRST` programını şöyle de yazabiliriz:

```
#include <iostream>
using
namespace std;

int main() { cout
<<
"Every age has a language of its own\n"
; return
0;}
```

Böyle bir söz dizimini tavsiye etmiyoruz. Bu, standart olmadığı gibi okunması da zor, ama hatasız çalışan bir program.

Boşlukların derleyici açısından önemi olmadığı kuralının birkaç istisnası vardır. `#include` ile başlayan, programın ilk ifadesi bir önişlemci direktifi olduğu için tek bir satırda yazılmak zorundadır. Ayrıca `"Every age has a language of its own"` gibi karakter katarı sabitleri de

ayrı satırlara bölünerek yazılamaz. (Eğer uzun bir karakter katarı sabitine ihtiyacınız varsa, satır sonuna ters bölü işareti (\) ekleyebilir veya karakter katarını, tırnakla çevrelenmiş iki ayrı karakter katarına bölebilirsiniz.)

cout Kullanarak Çıktı Almak

Gördüğünüz gibi,

```
cout << "Every age has a language of its own\n";
```

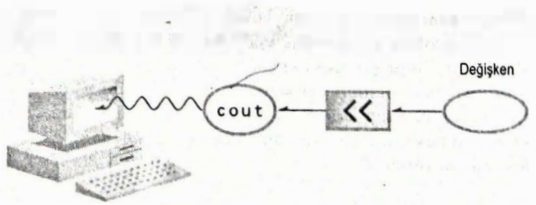
ifadesi tırnak içindeki deyimin ekrana yazılmasını sağlar. Peki bu nasıl çalışır? Bu ifadeyi tam manasıyla anlamak için nesneler, aşırı yüklenmiş operatörler ve diğer konuların anlaşılmış olması gerekir. Bu konuları daha sonraki bölümlerde öğreneceğiz ama, şimdilik kısaca bir gözden geçirelim.

`cout` (İngilizce "C out" olarak okunur) aslında bir *nesnedir*. `cout`, C++'ta önceden tanımlıdır ve *standart çıktı akışına* (*standard output stream*) karşılık gelir. *Akış* (*stream*), veri akışına karşılık gelen bir soyutlamadır. Standart çıktı akışı normalde çıktıyı ekrana gönderir –çıktı diğer çıkış cihazlarına da yönlendirilebilir. Akış (ve yönlendirme) konularını "Akışlar ve Dosyalar" başlıklı 12. bölümde ele alacağız.

`<<` operatörüne *ekleme operatörü* denir. `<<` operatörü, sağ tarafındaki değişkenin değerini sol tarafındaki nesneye yönlendirir. `FIRST` programında `"Every age has a language of its own\n"` karakter katarı sabitini `cout`'a yönlendirir; `cout` da bu karakter katarını ekrana gönderir.

(C biliyorsanız, `<<` operatörünün bit tabanlı *sola kaydırma* operatörü olduğunu fark edip, aynı zamanda çıktıyı yönlendirmek için nasıl kullanıldığını merak ediyorsunuzdur. C++'ta operatörler aşırı yüklenebilir. Yani, operatörler kontekste bağlı olarak değişik faaliyetlerde bulunabilirler. "Operatörlerin Aşırı Yüklenmesi" başlıklı 8. bölümde bu konuyu öğreneceğiz.)

`cout` ve `<<` operatörünün kullanımın ardında yatan kavramlar şu aşamada anlaşılmaz gibi görünse de, bunları kullanmak kolaydır. Hemen hemen tüm örnek programlarda bunlar da yer alır. Şekil 2.2'de `cout` ve ekleme operatörünün nasıl kullanıldığı gösteriliyor.



ŞEKİL 2.2: `cout` ile çıktı almak.

Karakter Katarı Sabitleri

Tırnak içinde yer alan `"Every age has a language of its own\n"` deyimi *karakter katarı sabitlere* bir örnektir. Bilebileceğiniz gibi, değişkenlerden farklı olarak sabitler, program çalıştığı

zaman yeni bir değer almazlar. Sabitlerin değeri program yazılırken belirlenir ve program çalıştığı sürece sabitler değerlerini korur.

İleride göreceğimiz gibi, C++'ta karakter katarlarını ilgilendiren durum oldukça karmaşıktır. Karakter katarlarını ele alış yöntemi çoğunlukla iki türlüdür. Bir karakter katarı, ya bir karakter dizisi ile ya da bir sınıfın nesnesi olarak gösterilebilir. Her iki çeşit karakter katarı hakkında daha fazlasını "Diziler ve Karakter Katarları" başlıklı 7. bölümde öğreneceğiz.

String sabitinin sonundaki '\n' karakteri, bir *kaçış sekansı* (escape sequence) örneğidir. Bu, bir sonraki metin çıktısının alt satırda gösterilmesine neden olur. Burada bunu kullanmamızın amacı programın sonunda bazı derleyiciler tarafından ekrana yazılan "Devam etmek için bir tuşa basın." gibi ifadelerin yeni satırda gösterilmesini sağlamaktır. Kaçış sekanslarını bu bölüm içinde ele alacağız.

Direktifler

FIRST programının başında yer alan ilk iki satır birer *direktiftir*. Bunlardan ilki bir *önişlemci direktifi*, ikincisi ise bir **using** *direktifidir*. Direktifler, temel C++ dilinin bir parçası değildirler ama, yine de gereklidirler.

Önişlemci Direktifleri

FIRST programındaki ilk satır

```
#include <iostream>
```

bir program ifadesi gibi gözükebilir ama, değil. Bu, bir fonksiyonun bir parçası değildir. Ayrıca noktalı virgül ile de bitmez. Oysa, program ifadelerinin sonunda mutlaka noktalı virgül olması lazımdır. Bunun yerine, # simgesi ile başlar. Buna *önişlemci direktifi* denir. Program ifadelerinin *bilgisayara* bir şeyler yaptırmaya yarayan komutlar olduğunu hatırlayın. Söz gelişi, iki sayının toplanması veya bir ifadenin yazıcıdan bastırılması gibi. Öte yandan, bir önişlemci direktifi ise, *derleyiciye* verilen bir komuttur. Derleyicinin *önişlemci* denilen bir parçası gerçek derleme işlemi başlamadan önce bu direktiflerle ilgilenir.

Önişlemci direktifi olan **#include** derleyiciye, kaynak dosyası içine başka bir dosya eklemesi gerektiğini bildirir. Aslında **#include** direktifinin yerine belirtilen dosya içeriği yerleştirilir. **#include** direktifini kullanarak kaynak dosyanızın içine başka bir dosya eklemek kelime işlemcinizin içine bir metin bloğunu yapıştırmaya benzer.

#include pek çok önişlemci direktifinden yalnızca bir tanesidir. Bu direktiflerin hepsi # simgesi ile başlar. Önişlemci direktiflerinin kullanımı C++'ta C'de olduğu kadar yaygın değildir. Yine de biz, ilerledikçe bir-iki ilave örneğe daha göz atacağız. **#include** ile birlikte kullanılan tip dosyasına *başlık dosyası* adı verilir.

Başlık Dosyaları

FIRST örneğinde, **#include** önişlemci direktifi derleyiciye, programı derlemeden önce **IOSTREAM** kaynak dosyasını **FIRST.CPP** dosyasına eklemesini bildirir. Bu niçin gereklidir? **IOSTREAM**, bir *başlık dosyası* (*header file* - bazen buna *include dosyası* da denir) örneğidir. **IOSTREAM** en temel giriş/çıkış işlemlerinden sorumludur; **cout** ve **<<** operatörü için gerekli tanımları içerir. Bu tanımlar olmadan derleyici **cout**'u tanıyamaz ve **<<** operatörünün hatalı kullanıldığını düşünebilir. Buna benzer bir çok başlık dosyası vardır. En yeni Standart C++ başlık dosyalarından dosya uzantısı kaldırılmıştır. Ancak C günlerinden kalma bazı eski başlık dosyaları hâlâ **.H** uzantısını içerir.

IOSTREAM'in içinde neler olduğunu merak ediyorsanız, derleyicinizin **include** dizinini bulun ve kaynak dosyayı Edit penceresinde açın. (Bunun için tavsiyeleri kitabın sonundaki ilgili ekte bulabilirsiniz.) Ya da bu dosyaya WordPad veya Notepad yardımıyla da bakabilirsiniz. Dosyanın içeriği şimdilik pek bir şey ifade etmeyebilir. Ancak **IOSTREAM**'in de normal ASCII karakteriyle yazılmış bir kaynak dosyası olduğunu gözlerinizle görmüş olursunuz.

using Direktifi

Bir C++ programı değişik *isim uzaylarına* (*namespace*) bölünebilir. *İsim uzayı*, programdaki belirli isimlerin derleyici tarafından tanınmasını sağlayan program parçasıdır. İsim uzayının dışında bu isimler tanınmaz.

```
using namespace std;
```

direktifi, bunu takip eden tüm program ifadelerinin **std** isim uzayında olduğunu belirtir. Çeşitli program bileşenleri, mesela **cout**, bu isim uzayında tanımlıdır. Eğer **using** direktifini kullanmamış olsaydık, pek çok program öğesinin başına **std** kelimesini eklemek zorunda kalacaktık. Örneğin, **FIRST** programında şöyle yazmamız gerekirdi:

```
std::cout << "Every age has a language of its own.";
```

Program boyunca **std::** deyimini defalarca kullanmak yerine **using** direktifini kullanıyoruz. İsim uzaylarını "Birden Fazla Dosya Kullanan Programlar" başlıklı 13. Bölümde ele alacağız.

Açıklamalar

Hangi program olursa olsun, açıklamalar programların önemli bir parçasıdır. Açıklamalar, programı yazan kişiye kolaylık sağlar; ayrıca, kaynak dosyayı okuması gereken kişilerin ne olup bittiğini anlamalarına yardımcı olur. Derleyici açıklamaları dikkate almaz. Böylece, açıklamalar dosya büyüklüğünü değiştirmez, çalıştırılabilir programların çalışma sürelerini etkilemez.

Açıklamaların Sözdizimi

İlk programımız olan **FIRST** programını kaynak dosyaya açıklamalar ekleyerek tekrar yazalım. Bu yeni programa **COMMENTS** ismini verelim:

```
// comments.cpp
// program aciklamalari
#include <iostream>          //onislemci direktifi
using namespace std;         //"using" direktifi
int main()                   //"main" isminde fonksiyon
   {                         //fonksiyonun govdesi burada baslar
   cout <<"Every age has a language of its own\n";       //program ifadesi
   return 0;                 //program ifadesi
   }                         //fonksiyonun govdesi burada sona erer
```

Açıklamalar çift bölü (//) sembolüyle başlar ve satır sonunda biter. (Bu, derleyicinin boşlukları dikkate almadığına dair kuralla ilgili bir istisnadır.) Açıklamalar satır başından

başlayabilir ya da bir program ifadesinin peşinden aynı satıra da yazılabilir. Muhtemel iki durum COMMENTS örneğinde verilmiştir.

Açıklamaları Ne Zaman Kullanırız?

Açıklamalar hemen hemen her zaman iyidir. Programcıların bir çoğu bunları yeteri kadar kullanmaz. Programlarınızda şayet açıklama yapmama gibi bir niyetiniz varsa, herkesin sizin kadar zeki olmadığını, programınızın ne yaptığı hakkında açıklamalara sizden daha fazla ihtiyaç duyabileceklerini aklınızdan çıkarmayın. Ayrıca, mesela önümüzdeki ay programınızın en temel ayrıntılarını unuttuğunuzda olayları siz de bugünkü kadar çabuk kavrayamıyor olabilirsiniz.

Programı listenize bakan bir kimseye ne yaptığınızı anlatmak amacıyla açıklama kullanın. Ayrıntılar zaten program ifadelerinde vardır. Yani, açıklamalar bir veya bir grup program ifadesini kullanma sebeplerinizi netleştirerek, yapılan işin geneline odaklanmalıdır.

Alternatif Açıklama Sözdizimi

C++'ta ikinci bir açıklama stili daha mevcuttur:

```
/* bu eski tip bir aciklama satiridir */
```

Bu tip açıklama (orijinal olarak C'de mevcut olan tek açıklama stili) `/*` karakter çiftiyle başlar ve `*/` karakterleriyle sonlanır (satır sonu açıklamanın bittiğini göstermez.) Bu simgelerin tuşlanması zordur, üstelik bunlar satırda daha çok yer kaplar. Bu nedenle, C++'ta bu stil pek kullanılmaz. Ancak, bu stil bazı özel durumlarda avantajlı olabilir. Sadece iki açıklama simgesi kullanarak birden fazla satır kaplayan açıklamalar yazabilirsiniz.

```
/* bu
cok uzun
birkac satir
uzunlugunda
bir
aciklama
*/
```

Açıklama olarak uzun metin pasajları yazmak gerektiğinde, her satıra `//` simgelerini yazmaktan kurtardığı için bu, iyi bir yöntemdir.

Ayrıca, bir program satırı içinde herhangi bir yere `/* */` tipinde bir açıklama ekleyebilirsiniz:

```
func1 ()
   {  /* ici bos bir fonksiyon govdesi */  }
```

Bu örnekte aynı şekilde `//` tipinde bir açıklama kullanmış olsaydınız, bu tip açıklamalar satır sonu kadar uzandığı için kapanan parantez simgesi derleyici tarafından fark edilmeyecek ve kod hatasız olarak derlenmeyecekti.

Tamsayı Değişkenler

Değişkenler herhangi bir dilin en temel parçasıdır. Bir değişkenin sembolik bir ismi vardır. Bir değişkene çeşitli değerler verilebilir. Değişkenler bilgisayarın belleğindeki belirli bölgelerde

yerleşiktirler. Bir değişkene bir değer verildiğinde bu değer gerçekte, bellekte o değişken için ayrılmış olan yere alınır. Birçok popüler dil aynı değişken tiplerini kullanır. Örneğin, tamsayılar, kayan noktalı sayılar, karakterler vs. Dolayısıyla, bunların ardında yatan fikirlere muhtemelen aşina olabilirsiniz.

Tamsayı değişkenler 1, 30,000 veya –27 gibi tamsayıları simgelerler. Bu tür sayılar ayrık sayıdaki nesneleri saymak için kullanılır; 11 kalem veya 99 şişe bira gibi. Kayan noktalı sayılardan farklı olarak tamsayıların kesirli kısmı bulunmaz. *Dört* kavramını tamsayı kullanarak ifade edebilirsiniz ama, *dört buçuk* kavramını ifade edemezsiniz.

Tamsayı Değişkenleri Tanımlama

Tamsayı değişkenler birkaç değişik büyüklükte olabilir. En yaygın kullanılanı `int` tipidir. Tamsayıların bellekte kapladıkları yer sisteme bağlı olarak değişir. Windows gibi 32 bit sistemlerde bir `int` 4 byte yer kaplar. Bu, `int` tipinde bir değişkenin –2,147,483,648 - 2,147,483,647 aralığındaki sayıları saklamasına imkan verir. Şekil 2.3'te bir tamsayı değişkenin bellekteki konumunu gösterilmiştir.

`int` tipi şimdiki Windows tabanlı bilgisayarlarda 4 byte yer kaplamasına karşın, MS-DOS'ta ve Windows'un ilk sürümlerinde sadece 2 byte yer kaplardı. Çeşitli tiplerin hangi aralıkları kapsadığı `LIMITS` isimli başlık dosyasında listelenmiştir. Bunlara ayrıca, derleyicinizin yardım sistemini kullanarak da ulaşabilirsiniz.

İşte size `int` tipinde birkaç tane değişken tanımlayan ve kullanan bir program:

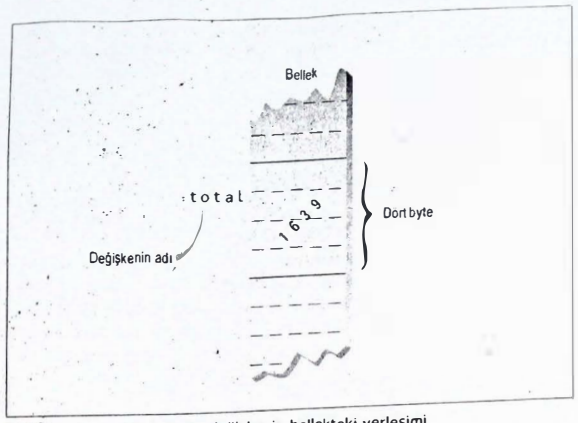
```
// intvars.cpp
// tamsayi degiskenler
#include <iostream>
using namespace std;

int main()
   {
   int var1;                  //var1'i tanimla
   int var2;                  //var2'yi tanimla
   var1 = 20;                 //var1'e deger ata
   var2 = var1 + 10;          //var2'ye deger ata
   cout << "var1+10 is ";     //metni ekrana yazdir
   cout << var2 << endl;      //var2'nin degerini ekrana yazdir
   return 0;
   }
```

Derleyicinizin düzenleme ekranında bu programı yazın (veya Web sitesinden yükleyin), derleyin, bağlayın (link), sonra da çalıştırın. Çıktı ekranını inceleyin.

```
int var1;
int var2;
```

ifadeleri ile `var1` ve `var2` adında iki tamsayı değişken tanımlanır. `int` anahtar kelimesi değişkenin tipini belirtir. *Deklarasyonlar* (*declarations*) olarak adlandırılan bu ifadeler, diğer ifadelerde olduğu gibi noktalı virgül ile sona ermelidir.

Bir değişkeni kullanmadan önce deklare etmelisiniz. Değişken deklarasyonlarını programın herhangi bir yerinde yapabilirsiniz. Değişkenleri çalıştırılabilir ilk ifadeden önce deklare etmeniz şart değildir (C'de bu gerekliydi.) Yine de, sıkça kullanılan değişkenler programın başında yer alırsa program muhtemelen daha rahat okunacaktır.



ŞEKİL 2.3: `int` tipinde bir değişkenin bellekteki yerleşimi.

Deklarasyonlar ve Tanımlar

Bir anlığına konumuzdan ayrılıp, *tanım* ve *deklarasyon* terimleri arasındaki ayrımsanması zor farkın değişkenler bağlamında ne olduğuna bir bakalım.

Deklarasyon, bir değişkenin ismini (örneğin, `var1`) ve tipini (örneğin, `int`) programa tanıtır. Bununla birlikte, eğer bir deklarasyon, değişken için bellekte yer de ayırıyorsa, bu aynı zamanda bir *tanımdır*. `INTVARS` isimli programdaki

```
int var1;
int var2;
```

ifadeleri hem tanım hem de deklarasyondur, çünkü `var1` ve `var2` için bellekte yer ayırırlar. Biz çoğunlukla tanım özelliği olan deklarasyonlarla ilgileneceğiz. Fakat ileride, tanım olmayan çeşitli deklarasyon türlerini de göreceğiz.

Değişken İsimleri

`INTVARS` isimli program `var1` ve `var2` isminde değişkenler kullanıyor. Değişkenlere (ve diğer program öğelerine) verilen isimlere *tanımlayıcı* (*identifier*) denir. Tanımlayıcılarla ilgili yazım kuralları nelerdir? Küçük ve büyük harfleri, 1 – 9 arası rakamları kullanabilirsiniz. Bir de altçizgi (_) simgesini kullanabilirsiniz. İlk karakter harf veya altçizgi olmalıdır. Tanımlayıcılar istediğiniz uzunlukta olabilir ancak, birçok derleyici sadece ilk birkaç yüz karakteri tanıyabilir. Derleyici büyük ve küçük harf ayrımı yapar. Bu nedenle, `var` kelimesi `var` veya `VAR` ile aynı değildir.

C++ anahtar kelimelerini değişken ismi olarak kullanamazsınız. Bir anahtar kelime özel anlamı olan önceden tanımlı bir sözcüktür. `int`, `class`, `if` ve `while` anahtar kelime örneklerindendir. Anahtar kelimelerin tam listesini Ek B'de, "C++ Öncelik Tablosu ve Anahtar Kelimeler" bölümünde ve derleyicinizin dokümanında bulabilirsiniz.

C++ programcılarının bir çoğu değişken isimleri için sadece küçük harfleri tercih ederler. Diğerleri, büyük ve küçük harflerin karışımını kullanıyorlar. Örneğin, `IntVar`, `dataCount` gibi. Bir kısmı ise altçizgi simgesini cömertçe kullanıyorlar. Hangi yaklaşımı tercih ederseniz edin, en iyisi program boyunca tutarlı olmaktır. Tümü büyük harflerden oluşan bazı isimler sabitler için ayrılmıştır (bir sonraki bölümde `const` ile ilgili açıklamalara bir bakın.) Aynı konvansiyonlar sınıf ve fonksiyon gibi diğer program öğelerinin adlandırılmasında da geçerlidir.

Bir değişkenin ismi, program listesini okuyan birine değişkenin amacını ve nasıl kullanıldığını açıkça ifade edebilmeli. Yani, `kaynamaSicakligi`, şifrelenmiş gibi görünen `kS` veya `s` gibi değişken isimlerinden çok daha iyidir.

Değer Atama (Assignment) İfadeleri

```
var1 = 20;
var2 = var1 + 10;
```

ifadeleri iki değişkene değer atar. Eşit işareti (=), tahmin edeceğiniz gibi, sağdaki değerin soldaki değişkene atanmasını sağlar. C++'taki = simgesi Pascal'daki := karakterlerine ya da BASIC'te = simgesine denktir. Burada gösterilen ilk satırda önceden bir değeri olmayan `var1` değişkenine 20 değeri verilir.

Tamsayı Sabitler

20 sayısı bir tamsayı sabittir. Sabitlerin değeri program akışı süresince değişmez. Bir tamsayı sabiti nümerik rakamlardan ibarettir. Tamsayı sabiti içinde, ondalık kesir ifade eden nokta simgesi (.) olmamalıdır. Ayrıca tamsayı sabitinin değeri tamsayılar aralığında bulunmalıdır.

Burada gösterilen ikinci program satırında artı işareti (+) `var1`'in değeri ile 10'u toplar. 10 da bir başka sabittir. Bu toplamanın sonucu `var2` değişkenine atanır.

Çıktı Varyasyonları

```
cout << "var1+10 is ";
```

ifadesi, önceden de gördüğümüz gibi, bir karakter katarı sabitini ekrana yazdırır. Sonraki ifade

```
cout << var2 << endl;
```

`var2` değişkeninin değerini görüntüler. Konsol çıktı pencerenizde programın çıktısının

```
var1 + 10 is 30
```

olduğunu görebilirsiniz.

Dikkat ederseniz `cout` ve << operatörü, tamsayı ve karakter katarı sabitlerini farklı olarak ele alacaklarını bilirler. Bunlara bir karakter katarı gönderirsek, çıktıyı metin olarak basarlar. Eğer bir tamsayı gönderirsek, bunu sayı olarak basarlar. Bunun böyle olacağı zaten bellidir. Ancak bu, C++'ın en temel özelliği olan, operatörleri aşırı yükleme işleminin bir başka örneğidir. (C programcıları hatırlayacaklardır, C'deki `printf()` gibi fonksiyonlar için ekrana yazılacak değişkenden başka, değişkenin tipinin de bildirilmesi gerekir. Bu ise, sözdizimini çok daha az sezgisel kılar.)

Gördüğünüz gibi, `cout` ifadelerinin her ikisinin de çıktıları ekranda aynı satırda görülür. Otomatik olarak satır atlanmaz. Şayet yeni bir satırdan başlamak istiyorsanız, linefeed (satır atlama) işlemini sizin eklemeniz gerekir. `'\n'` kaçış sekansı ile bunun nasıl yapılabileceğini görmüştünüz. Şimdi bir başka yöntem öğreneceğiz: *Manipülatör* denilen bir şey kullanarak satır atlama.

endl Manipülatörü

`INTVARS` programındaki en son `cout` ifadesi tanıdık olmayan bir kelime ile biter: `endl`. Bu, çıktıda satır atlanmasını sağlar. Böylece bunu takip eden metin, bir sonraki satırda görüntülenir. Bu, `'\n'` karakteriyle aynı etkiye sahiptir, fakat biraz daha anlaşılır bir özelliktir. Bu aynı zamanda bir *manipülatör* örneğidir. Manipülatörler çıktı akışını değişik yollardan değiştiren komutlardır. İleriki bölümlerde manipülatörlerden daha çok söz edeceğiz. Tam anlamıyla söylemek gerekirse, `endl` (`'\n'` deyiminden farklı olarak) çıktı tamponunun (buffer) temizlenmesini de sağlar. Fakat bu, göz önünde gerçekleşmediğinden bir çok amaç için her iki yöntem de aynı şekilde sonuç verir.

Diğer Tamsayı Tipleri

`int` tipinin yanı sıra birkaç tane daha nümerik tamsayı tipi vardır. En yaygın iki tip `long` ve `short` tipleridir. (Tam olarak söylemek gerekirse, `char` tipi de aslında bir tamsayı tipidir ancak, bunu ayrıca ele alacağız.) `int` tipinin büyüklüğünün sisteme bağlı olduğuna dikkat çekmiştik. Buna karşın, `long` ve `short` tipleri, hangi sistem kullanılırsa kullanılsın sabit büyüklüğe sahiptir.

`long` her zaman 4 byte kaplar. Bu, 32 bit Windows sistemlerinde `int` tipinin kapladığı büyüklükle aynıdır. Dolayısıyla, `long` da -2,147,483,648 ile 2,147,483,647 arasındaki (aynı) aralıkta tanımlıdır. `long` ayrıca `long int` olarak da yazılabilir; `long int`, `long` ile aynı anlamı taşır. 32 bit sistemlerde `long` tipini kullanmanın pek bir esprisi yoktur, çünkü `long` zaten `int`'e eştir. Bununla birlikte, eğer programınızı MS-DOS veya Windows'un eski sürümleri gibi 16 bit sistemlerde çalıştırmanız gerekiyorsa, `long` tipini kullanmanız 4 byte tamsayı tipini garanti edecektir. 16 bit sistemlerde `int` tipinin tanım aralığı `short` ile aynıdır.

`short` tipi, tüm sistemlerde 2 byte yer kaplar. Dolayısıyla, -32,768 – 32,767 aralığında tanımlıdır. Modern Windows sistemlerinde, eğer bellekten kazanmak önemli değil ise `short` tipini kullanmanın muhtemelen pek fazla bir anlamı olmayacaktır. `int`, `short`'tun iki katı büyüklükte olmasına rağmen `int`'e erişim çok daha hızlıdır.

Eğer `long` tipinde bir sabit tanımlamak isterseniz, nümerik değerin ardından `L` harfini ekleyin. Mesela,

```
longvar = 7678L;  //long tipinde sabit 7678 degeri longvar degiskenine ataniyor
```

Birçok derleyici, kullanılan bit sayısını açıkça belirten tamsayı tipleri sunuyorlar. (Bir byte'ın 8 bit'ten oluştuğunu hatırlayın.) Bu tip isimlerinin önünde çift altçizgi simgesi bulunur. Bu tipler `__int8`, `__int16`, `__int32` ve `__int64` tipleridir. `__int8`, `char` tipine karşılık gelir. `__int16` (en azından 16 bit sistemlerde) `short`'a; `__int32` ise `int` ve `long`'a karşılık gelir. `__int64` tipi en fazla 19 basamağa kadar varan çok büyük sayıları saklar. Bu tip isimlerini kullanmanın avantajı, bir değişken için ayrılan byte sayısının uygulamaya bağlı olmamasıdır. Yine de, bu sıkla karşılaşılan bir problem değildir; bu nedenle, bu tipler ender olarak kullanılırlar.

Karakter Değişkenleri

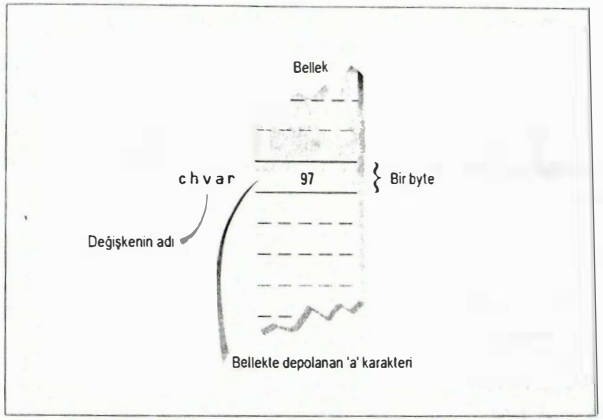
`char` tipi –128 ile 127 arasındaki tamsayıları saklar. Bu tipteki değişkenler bellekte 1 byte (8 bit) yer kaplar. Karakter değişkenleri kimi zaman belirtilen aralıkla sınırlı sayıları saklamak amacıyla da kullanılabilir, fakat çok daha yaygın olarak, ASCII karakterleri saklamak için kullanılırlar.

Önceden bilebileceğiniz gibi, ASCII karakter seti, karakterleri sayılarla simgeleme yollarından biridir. (`'a'`, `'B'`, `'$'` ve `'3'` birer karakter örneğidir.) Bu sayılar 0 ile 127 arasındadır. Birçok Windows sistemi bu aralığı 255'e kadar genişletir. Böylelikle çeşitli yabancı dil ve grafik karakterlerine de yer verilmiş olur. Ek A'daki "ASCII Tablosu" ASCII karakter setini gösterir.

Yabancı dil kullanıldığında karmaşa doğar. Hatta programları bilgisayar sistemleri arasında aktarırken aynı dilde bile sorunlar çıkabilir. Bunun sebebi, 128 – 255 aralığındaki karakterler standartlaştırılmamıştır. Ayrıca 1 byte büyüklüğündeki `char` tipi, Japonca gibi bir çok dilin karakter sayısını saklamak için çok küçüktür. Standart C++ yabancı dillerdeki karakterlerle başa çıkabilmek için `wchar_t` adında daha büyük bir karakter tipini destekler. Uluslararası bazda dağıtılacak programlar yazıyorsanız bu, önemlidir. Ancak, biz bu kitapta `wchar_t` tipini göz ardı edip, Windows'un şu anki sürümlerinde mevcut ASCII karakter setiyle ilgileneceğiz.

Karakter Sabitleri

Karakter sabitlerinde bir karakter etrafına tek tırnak işareti kullanırlar. Söz gelişi, `'a'` ve `'b'` gibi. (Bunun çift tırnak kullanılan karakter katarı sabitlerinden farklı olduğuna dikkat edin.) C++ derleyicisi böyle bir karakter sabitiyle karşılaşınca, bu karakter sabitini karşılık gelen ASCII koduna çevirir. Şekil 2.4'te gösterildiği gibi, program içindeki, mesela bir `'a'` sabiti 97'ye çevrilecektir.



ŞEKİL 2.4: `char` tipinde bir değişkenin bellekteki yerleşimi.

Karakter değişkenlerine değer olarak karakter sabitleri atanabilir. Aşağıdaki program bazı karakter sabitlerine ve değişkenlerine örnekler içerir:

```
// charvars.cpp
// karakter degiskenleri
#include <iostream>          //cout vs. icin gerekli
using namespace std;

int main()
   {
   char charvar1 = 'A';      //char tipinde degiskeni karakter olarak tanimla
   char charvar2 = '\t';        //char tipinde degiskeni sekme olarak tanimla

   cout << charvar1;         //karakteri goruntule
   cout << charvar2;         //karakteri goruntule
   charvar1 = 'B ';          //char tipinde degiskene char tipinde sabit degerini ata
   cout << charvar1;         //karakteri goruntule
   cout << '\n';             //newline karakterini goruntule
   return 0;
   }
```

İlk Kullanıma Hazırlama (İlk Değer Atama)

Değişkenlere tanımlandıkları anda ilk değeri atanabilir. Bu programda `char` tipindeki iki değişkene –`charvar1` ve `charvar2`– ilk değer olarak `'A'` ve `'\t'` karakter sabitleri atanır.

Kaçış Sekansları

İkinci karakter sabiti olan `'\t'` biraz tuhaftır. Önceden karşılaştığımız `'\n'` gibi, bu da *kaçış sekansı* (*escape sequence*) örneklerinden biridir. Kaçış sekanslarındaki ters bölü (\) işareti, kaçış sekansını oluşturan karakterin normal anlamıyla anlaşılmasını önler. Bu örnekte t, 't' karakteri olarak değil, sekme karakteri olarak yorumlanır. Sekme, ekrandaki görüntüleme işleminin bir sonraki sekme durak yerinden devam etmesine neden olur. Konsol modunda çalışan programlarda sekme durak yerleri sekiz boşluk arayla konumlandırılmışlardır. Programın son satırında bir başka karakter sabiti olan `'\n'`, doğrudan `cout`'a gönderilir.

Kaçış sekansları ayrı birer karakter gibi kullanılabilir veya karakter katarı sabitlerinin içine gömülebilir. Tablo 2.1'de, yaygın olarak kullanılan kaçış sekanslarının bir listesi görülüyor.

TABLO 2.1: Yaygın Olarak Kullanılan Kaçış Sekansları

Kaçış Sekansı	Karakter
`\a`	Zil (bip)
`\b`	Backspace (geri al)
`\f`	Formfeed
`\n`	Newline (yeni satır)
`\r`	Return (paragraf sonu)
`\t`	Sekme
`\\`	Ters bölü (Backslash)
`\'`	Tek tırnak işareti
`\"`	Çift tırnak işareti
`\xdd`	Onaltılık notasyon

Ters bölü işareti, tek tırnak işareti ve çift tırnak işareti sabit olarak kullanıldıklarında özel anlam içerdikleri için bunları karakter olarak görüntülemek istersek kaçış sekansları ile simgelememiz gerekir. Tırnak içindeki bir deyimin karakter katarı sabiti içinde gösterimine bir örnek verelim:

```
cout << "\"Run, Spot, run,\" she said.";
```

Bu, ekranda şu şekle dönüşür:

```
"Run, Spot, run," she said.
```

Kimi zaman klavyede bulunmayan karakter sabitlerini simgelemeniz gerekebilir. Söz gelişi, ASCII kodu 127'den büyük grafik karakterleri gibi. Bunu yapabilmek için `'\xdd'` simgesini kullanabilirsiniz. `'\xdd'` içindeki her `d` bir onaltılık (hexadecimal) rakama karşılık geliyor. Örneğin, içi dolu bir dikdörtgen çizmek istiyorsanız, ondalık değeri 178 olan böyle bir karakterin listelenmiş olduğunu göreceksiniz. Bu, ASCII tabloda `B2` onaltılık sayısına denk geliyor. Yani, bu karakter `'\xB2'` karakter sabiti ile simgelenecektir. İleride bununla ilgili bazı örnekler göreceğiz.

`CHARVARS` programı `charvar1` ve `charvar2`'nin değerlerini ekrana basar. (`charvar1`'in değeri `'A'`, `charvar2`'ninki ise sekme karakteridir.) Daha sonra `charvar1`'e yeni bir değer (`'B'`) atar, bunu ekrana yazar ve son olarak da newline karakterini yazar. Çıktı, ekranda şöyle görünür:

```
A  B
```

cin ile Giriş Almak

Değişken tiplerinin kullanımını öğrendiğimize göre, şimdi de bir programın nasıl giriş aldığını görelim. Sıradaki örnek kullanıcıya Fahrenheit cinsinden sıcaklık değerini sorar, bunu Celsius'a çevirir ve sonucu görüntüler. Programda tamsayı değişkenler kullanılmıştır.

```
// fahren.cpp
// cin ve newline
#include <iostream>
using namespace std;

int main()
   {
   int ftemp;     //fahrenheit cinsinden sicaklik degeri icin

   cout << "Enter temperature in fahrenheit: ";
   cin >> ftemp;
   int ctemp = (ftemp-32) * 5 / 9;
   cout << "Equivalent in Celsius is: " << ctemp << '\n';
   return 0;
   }
```

Programdaki

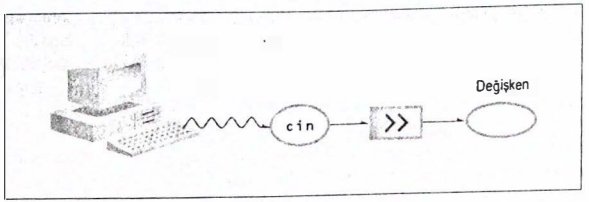
```
cin >> ftemp;
```

ifadesi kullanıcının bir sayı tuşlaması için programın beklemesini sağlar. Elde edilen sayı `ftemp` değişkenine yerleştirilir. `cin` (C in olarak okunur) anahtar kelimesi standart giriş akışına karşılık gelen, C++'ta önceden tanımlı bir nesnedir. Bu akış, (yönlendirilmediği sürece) klavyeden gelen verileri simgeler. `>>` operatörü, *çıkarma (extraction)* veya *get from* operatörüdür. Solundaki akış nesnesinden değeri alır, sağındaki değişkene yerleştirir.

İşte size bu programla ilgili bazı etkileşim örnekleri:

```
Enter temperature in fahrenheit: 212
Equivalent in Celsius is: 100
```

Şekil 2.5 `cin` ve `>>` operatörünü kullanarak input almayı gösteriyor.



ŞEKİL 2.5: `cin` ile input almak.

Kullanım Anında Değişken Tanımlama

`FAHREN` programı giriş becerisinin yanı sıra birkaç yeni özelliğe de sahiptir. Program listesine daha yakından bakalım. `ctemp` değişkeni nerede tanımlanmış? Programın en başında değil de, aritmetik işleminin sonucunu saklamak kullanıldığı yerin hemen yanı başında tanımlanmış. Önceden belirttiğimiz gibi, değişkenleri sadece programın başında değil, program boyunca her yerde tanımlayabilirsiniz. (Bir çok dil, C de dahil olmak üzere, tüm değişkenlerin çalıştırılabilir ilk ifadeden önce tanımlanmasını zorunlu kılar.)

Değişkenleri kullanıldığı yerde tanımlamak program listesini okumayı kolaylaştırır, çünkü böylece değişken tanımlarını bulmak için her defasında listenin başına dönmek zorunda kalmazsınız. Yine de, bunu sağduyulu kullanmak gerekir. Bir fonksiyon içinde bir çok yerde kullanılan değişkenlerin fonksiyonun başladığı yerde tanımlanmaları daha iyidir.

Basamaklanmış << Operatörleri

Ekleme operatörü `<<`, `FAHREN` programındaki ikinci `cout` ifadesinde tekrar tekrar kullanılmıştır. Bu tamamen kurallara uygun bir işlemdir. Program önce *Equivalent in Celsius is:* deyimini, sonra `ctemp`'in değerini, son olarak da newline `'\n'` karakterini `cout`'a gönderir.

Aynı şekilde, çıkarma operatörü `>>` de, kullanıcının bir dizi giriş yapmasını temin etmek için `cin`'in peşi sıra kullanılabilir. Ancak bu özellik sıkça kullanılmaz, çünkü girdiler arasında kullanıcıya hangi girişi yapması gerektiğini bildirme fırsatını ortadan kaldırır.

Deyimler

Değişken, sabit ve operatörlerin bir hesaplama belirtecek şekilde düzenlenmesine bir *deyim* (*expression*) denir. Yani, `alpha + 12` ve `(alpha-37)*beta/2` birer deyimdir. Deyimde belirtilen

hesaplama gerçekleştirildiğinde sonuç genellikle bir değer olur. Yani, eğer `alpha`'nın değeri 7 ise burada verilen ilk deyim 19 değerine sahip olacaktır.

Deyimlerin parçaları da bir deyim olabilir. İkinci örnekte `alpha-37` ve `beta/2` birer deyimdir. Hatta tek bir değişken veya sabit bile, `alpha` veya 37 gibi, deyim olarak ele alınabilir.

Deyimlerin ifadelerle aynı olmadığına dikkat edin. İfadeler derleyiciye bir şey yapmasını bildirir ve noktalı virgül ile sona erer; deyimler ise bir hesaplama belirtirler. Bir ifade içinde birkaç tane deyim yer alabilir.

Öncelikler

```
(ftemp-32) * 5 / 9
```

deyimindeki parantezlere dikkat edin. Parantezler olmasaydı, * operatörü, - operatöründen daha yüksek önceliğe sahip olduğu için ilk önce çarpma işlemi yapılacaktı. Parantezler sayesinde önce çıkartma işlemi, ardından çarpma işlemi yapılır, çünkü parantezlerin içindeki işlemler önceliğe sahiptir. Peki, * ve / işaretleri arasındaki öncelik nasıldır? Şayet iki aritmetik operatörü aynı önceliğe sahip ise, sol tarafta olan önce hesaplanır. Bu örnekte, sıradaki işlem çarpma olacak, ardında da bölme yapılacaktır. Öncelikler ve parantezler, cebirde ve diğer bilgisayar dillerinde de normal olarak aynı şekilde uygulanır. Bu nedenle, bunların kullanımları muhtemelen size çok doğal gelmiştir. Ancak, öncelikler, C++'ta önemli bir konudur. Değişik tipteki operatörleri tanıtacağımız zaman bu konuya tekrar döneceğiz.

Kayan Noktalı Tipler

Sayıları tamsayı -yani, kesirli kısmı olmayan sayılar- olarak simgeleyen `int` ve `char` tiplerinden bahsetmiştik. Şimdi sayıları saklamanın farklı bir yöntemini inceleyelim: Kayan noktalı (floating-point) değişkenler.

Kayan noktalı değişkenler sayıları ondalık kısmıyla birlikte simgelerler. Örneğin, 3.1415927, 0.0000625, -10.2 vs. Bu tür değişkenlerin hem tamsayı kısmı hem de kesirli kısmı vardır. Tamsayı kısmı, ondalık kesri ifade eden noktanın sol tarafında; kesirli kısmı ise sağ tarafındadır. Kayan noktalı değişkenler, matematikçilerin *reel sayı* olarak adlandırdıkları, ölçülebilir çoklukları simgelerler. Örneğin, mesafe, alan ve sıcaklık gibi. Bunlar çoğunlukla kesirli çokluklardır.

C++'ta üç çeşit kayan noktalı değişken mevcuttur: **`float` tipi, `double` tipi ve `long double` tipi**. Şimdi, bunların en küçüğünden, `float`'tan başlayalım.

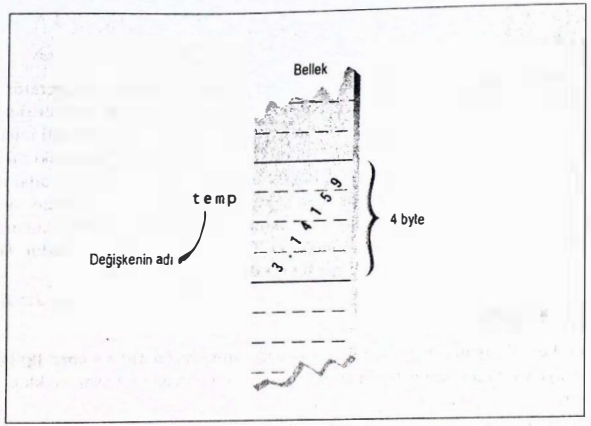
float Tipi

`float` tipi, 3.4×10^{-38} ile 3.4×10^{38} arasındaki sayıları saklar. Ondalık kısmı noktadan sonra en fazla 7 basamak uzunluğunda olabilir. Şekil 2.6'da görüldüğü gibi `float`, bellekte 4 byte (32 bit) yer kaplar.

Aşağıdaki program, dairenin yarıçapını simgelemek üzere kullanıcının bir kayan noktalı sayı tuşlamasını ister. Sonra da dairenin alanını hesaplayıp, ekranda gösterir.

```
// circarea.cpp
// kayan noktali degiskenler
#include <iostream>                    //cout vs. icin
using namespace std;

int main()
   {
```

```
float rad;                              //float tipinde degisken
const float PI = 3.14159F;              //float tipinde sabit

cout << "Enter radius of circle: ";     //kullanicidan giris iste
cin >> rad;                             //yaricapi al
float area = PI * rad * rad;            //alani hesapla
cout << "Area is " << area << endl;     //yaniti goruntule
return 0;
}
```



ŞEKİL 2.6: `float` tipinde bir değişkenin bellekteki yerleşimi.

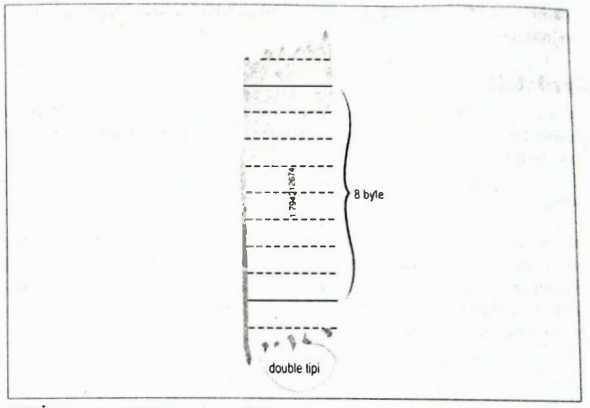
Programla ilgili bir etkileşim örneği şöyle olabilir:

```
Enter radius of circle: 0.5
Area is 0.785398
```

Bu, 0.5 fit yarıçapa sahip, 12-inç LP plağının fit-kare cinsinden alanıdır. Bir zamanlar bu, vinil üreticileri için çok önemli bir nicelikti.

double ve long double Tipleri

Kayan noktalı tiplerin daha büyüklerinden olan **`double`** ve **`long double`**, **`float`** ile aynıdır. Sadece, bunlar bellekte daha fazla yer getirirler; ayrıca, daha geniş bir değer aralığına sahiptirler ve daha uzun bir ondalık kısma imkan verirler. **`double`** tipi 8 byte yer kaplar, 1.7×10^{-308} ile 1.7×10^{308} arasındaki sayılarla başa çıkabilir ve ondalık kısmı 15 basamak uzunluğunda olabilir. **`long double`**, derleyiciye bağlı olmakla beraber genellikle **`double`** ile aynıdır. **`double`** tipi Şekil 2.7'de gösterilmiştir.



ŞEKİL 2.7: `double` tipinde değişken.

Kayan Noktalı Sabitler

`CIRCAREA` programındaki 3.14159F sayısı *kayan noktalı sabite* bir örnektir. Sayının içindeki nokta bu sayının tamsayı değil de, reel sayı olduğunu; **`F`** harfi, tipinin **`double`** veya **`long double`** yerine **`float`** olduğunu belirtir. Sayı, normal ondalık kesir notasyonunda yazılır. **`double`** tipindeki sabitler için sayının sonuna bir harf eklemenize gerek yok, çünkü **`double`** zaten varsayılan (default) değerdir. **`long double`** tipi için sayı sabitinin sonuna **`L`** harfini eklemeniz gerekir.

Kayan noktalı sabitleri *üstel* (*exponential*) *notasyon* kullanarak da yazabilirsiniz. Üstel notasyon, büyük sayıları bir çok sıfır yazmaya gerek kalmadan yazma yollarından biridir. Örneğin, 1,000,000,000 sayısı üstel notasyonla 1.0E9 olarak yazılabilir. Aynı şekilde, 1234.56 sayısı 1.234E3 olarak yazılabilir. (Bu, 1.234 kere 10^3 demektir.) **`E`** harfini takip eden sayıya *üs* (*exponent*) denir. Üs, sayıyı ondalık notasyona çevirmek için noktayı kaç basamak kaydırmak gerektiğini gösterir.

const Niteleyicisi

`CIRCAREA` programı **`float`** tipinde değişkenlerin yanı sıra, **`const`** niteleyicisini de tanıtır. **`const`** niteleyicisi şu ifadede kullanılır:

```
const float PI = 3.14159F;     //float tipinde sabit
```

Sabit (constant) deyimine karşılık gelen **`const`** anahtar kelimesi, değişkenin veri tipinden önce gelir. **`const`**, değişkenin değerinin program boyunca değişmeyeceğini bildirir. Bu niteleyici ile tanımlanmış bir değişkenin değerini değiştirmeye yönelik her türlü girişim derleyicinin hata mesajı vermesine neden olur.

`const` niteleyicisi, sabit olarak tanımladığınız bir değişkenin değerini, mesela **`CIRCAREA`**'daki **`PI`**'nin değerini, programınızın yanlışlıkla değiştirmemesini garanti eder. Ayrıca program listesini okuyan kişiye, değişkenin değerinin değiştirilmemesi gerektiğini hatırlatır. **`const`** niteleyi-

cisi, temel değişkenlerin yanı sıra diğer öğelere de uygulanabilir. Sonraki bölümlerde bununla ilgili daha fazlasını öğreneceğiz.

#define Direktifi

C++ yapısında tavsiye edilmiyor olsa da, sabitler `#define` direktifi kullanılarak da tanımlanabilir. Bu direktif, bir tanımlayıcı ile metinsel bir deyim arasında denklik ilişkisi kurar. Örneğin, programınızın başında bulunan

```
#define PI 3.14159
```

satırı, programınız boyunca `PI` tanımlayıcısının geçtiği her yerde `PI`'nin `3.14159` metni ile değiştirileceğini belirtir. Bu, C'de popüler olan bir yapıydı. Ancak, `#define` kullandığınızda sabitin veri tipini belirtemiyorsunuz; bu durum program hatlarına yol açabilir. Bu nedenle, C'de bile, normal değişkenlerle kullanılan `const` niteleyicisi, `#define`'ın yerini almıştır. Her şeye rağmen, eski programlarda bu yapıya rastlayabilirsiniz.

bool Tipi

Konunun tamamlanması için burada `bool` tipinden de bahsetmemiz gerekir. Aslında bu tip, bir sonraki bölümde ilişkisel operatörler ele alınana kadar önemli olmayacaktır.

Gördüğümüz gibi, `int` tipindeki değişkenler milyarlarca değişik değer alabilir; `char` tipindekiler ise 256 farklı değer alabilir. `bool` tipindeki değişkenler ise sadece iki değerden birine sahip olabilir: `true` (doğru) veya `false` (yanlış). Teorik olarak `bool` tipindeki bir değişken bir bit yer kaplar. Fakat pratikte derleyiciler bu tip değişkenler için byte cinsinden yer ayırırlar; çünkü bir byte çok daha çabuk erişilebilir, bir bit'e erişmek için bunun byte içinden çıkarılması gerekir –ki, bu ekstra zaman gerektirir.

İleride bahsedeceğimiz gibi, `bool` tipi çoğunlukla karşılaştırma sonuçlarını saklamak için kullanılır. `alpha`, `beta`'dan küçük mü? Eğer küçükse, bir `bool` değeri `true` olur; değilse, bir `bool` değeri `false` olur.

`bool` tipi ismini 19. Yüzyıl İngiliz matematikçisi George Bool'dan alıyor. Bool, doğru ve yanlış değerleri yardımıyla mantıksal operatörleri kullanmayı icat etti. Bu nedenle, bu tür doğru/yanlış değerleri genellikle *Boolean* değerler olarak adlandırılır.

setw Manipülatörü

Manipülatörlerin, verilerin görüntülenme şeklini değiştirmek veya düzenlemek için ekleme operatörü (<<) ile kullanılan operatörler olduğundan bahsetmiştik. Ayrıca `endl` operatörünü de görmüştük. Şimdi bir başkasına göz atacağız: Bu, çıktının genişliğini değiştiren `setw` manipülatörüdür.

`cout` ile görüntülenen her değerin bir alan kapladığını düşünebilirsiniz. Bu alan, belli genişliği olan hayali bir kutudur. Varsayılan alan, karşılık gelen değeri saklamaya yetecek genişliktedir. Yani, 567 sayısı üç karakter genişliğinde; "pijamalar" karakter katarı ise dokuz karakter genişliğinde yer kaplar. Ancak belli bazı durumlarda bu, optimal bir sonuca götürmez. Bir örnek verelim. `WIDTH1` programı bir sütunda üç şehrin isimlerini, diğer sütunda bu şehirlerin nüfuslarını görüntüler.

```
// width1.cpp
// setw manipulatorune duyulan gereksinim
```



```
#include <iostream>
using namespace std;

int main()
   {
   long pop1=2425785, pop2=47, pop3=9761;
   cout << "LOCATION " << "POP." << endl
        << "Portcity " << pop1 << endl
        << "Hightown " << pop2 << endl
        << "Lowville " << pop3 << endl;
   return 0;
   }
```

Bu programın çıktısı şöyledir:

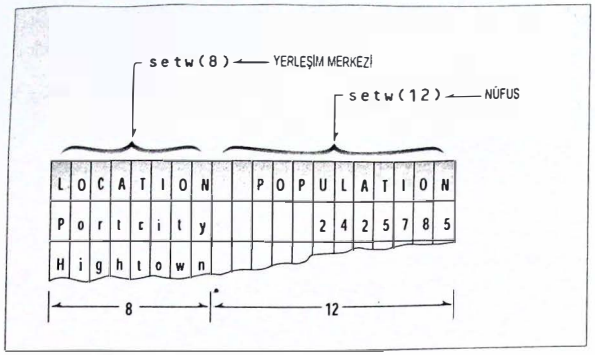
```
LOCATION POP.
Portcity 2425785
Hightown 47
Lowville 9761
```

Ne yazık ki, bu format sayıları karşılaştırmayı zorlaştırır. Sayılar sağa dayalı listenmiş olsa daha iyi olurdu. Ayrıca, şehir isimlerini sayılardan ayırmak için program içinde şehir isimlerinin sonuna boşluk eklemek zorunda kaldık. Bu da uygun olmayan bir durumdur.

Şimdi bu programın değiştirilmiş bir versiyonunu yazalım. `WIDTH2`, bu problemleri ortadan kaldırmak amacıyla, isimler ve sayılar için alan genişliği tanımlamayı sağlayan `setw` manipülatörünü kullanır.

```
// width2.cpp
// setw manipulatoru
#include <iostream>
#include <iomanip>              //setw icin gerekli
using namespace std;

int main()
   {
   long pop1=2425785,pop2=47,pop3=9761;

   cout << setw(8) << "LOCATION " << setw(12)
        << "POPULATION " <<endl
        << setw(8) << "Portcity" << setw(12) << pop1 << endl
        << setw(8) << "Hightown" << setw(12) << pop2 << endl
        << setw(8) << "Lowville" << setw(12) << pop3 << endl;
   return 0;
}
```

`setw` manipülatörü, ekrana basılacak sayı (veya karakter katarının) `n` karakter genişliğindeki bir alana yazılmasını sağlar. Buradaki `n`, `setw(n)` fonksiyonunun argümanıdır. Değer, alan içinde sağa dayalı olarak yazılır. Bunun ekranda nasıl göründüğünü Şekil 2.8'de görebilirsiniz. Nüfus bilgileri için `long` tipi kullanılır. En büyük değerin 32,767 olduğu, 2 byte yer kaplayan tamsayı tipi kullanan sistemlerde potansiyel taşma problemlerini önlemek için bu tanımlama yapılır.



ŞEKİL 2.8: Alan genişliği ve `setw`.

`WIDTH2` programının çıktısı şöyledir:

```
LOCATION    POPULATION
Portcity       2425785
Hightown            47
Lowville          9761
```

Ekleme Operatörünün Basamaklandırılması

`WIDTH1` ve `WIDTH2` programlarında sadece bir tane `cout` ifadesi olduğuna dikkat edin. Gerçi birkaç satır uzunluğunda olsa da, yine de her iki programda tek bir `cout` ifadesi mevcuttur. Bunu böyle yaparken, derleyicinin boşlukları dikkate almamasından ve ekleme operatörünü de peş peşe (cascade) kullanabilme avantajlarından yararlanıyoruz. Bu işlem, her biri `cout` ile başlayan dört ayrı ifade yazmakla yanı etkiye sahiptir.

Çoklu Tanımlama

`pop1`, `pop2` ve `pop3` değişkenlerini tanımlarken, aynı zamanda ilk değerlerini de atadık. Bu, `CHARVARS` örneğindeki `char` değişkenlerine ilk değer atama yöntemine benziyor. Fakat burada her üç değişkeni tek satırda, `long` anahtar kelimesini kullanarak ve değişken isimlerini virgül ile ayırarak tanımladık. Birkaç tane değişkenin aynı tipte olduğu durumlarda bu yöntem alandan kazandırır.

IOMANIP Başlık Dosyası

`endl` haricindeki manipülatörlerin tanımları genellikle `IOSTREAM` başlık dosyasında değildir; `IOMANIP` denilen ayrı bir dosya içindedir. Bu manipülatörleri kullandığınızda bu başlık dosyasını `#include` direktifi ile programınıza eklemeniz gerekir. `WIDTH2` örneğinde bu şekilde yapmıştık.

Değişken Tiplerinin Özeti

Örnek programlarımızda şimdiye kadar dört veri tipini gördük: `int`, `char`, `float` ve `long`. İlaveten, `bool`, `short`, `double` ve `long double` tiplerinden de bahsettik. Şimdi bir ara verelim ve bu veri tiplerini özetleyelim. Tablo 2.2'de tipi tanımlamak için gerekli anahtar kelime, tipin tanımlı olduğu nümerik aralık, ondalık kısmın basamak uzunluğu (kayan noktalı sayılar için) ve 32 bit ortamlarda bellekte kapladığı byte cinsinden alan listelenmiştir.

TABLO 2.2: Temel C++ Değişken Tipleri

	Nümerik Aralık			
Anahtar Kelime	**Alt Sınır**	**Üst Sınır**	**Ondalık Kısımda Maksimum Basamak Sayısı**	**Bellek alanı (byte cinsinden)**
`bool`	false	true	yok	1
`char`	-128	127	yok	1
`short`	-32,768	32,767	yok	2
`int`	-2,147,483,648	2,147,483,647	yok	4
`long`	-2,147,483,648	2,147,483,647	yok	4
`float`	3.4x10-38	3.4x1038	7	4
`double`	1.7x10-308	1.7x10308	15	8

unsigned Veri Tipleri

Karakter ve tamsayı tiplerindeki işareti kaldırarak, bu tiplerin aralıklarını 0'dan başlayacak şekilde değiştirebilir ve sadece pozitif sayıları kapsamasını sağlayabilirsiniz. Bu, sayıları işaretli versiyonlarına kıyasla iki kat daha büyük simgelemeye imkan verir. Tablo 2.3 tiplerin işaretsiz versiyonlarını gösteriyor.

TABLO 2.3: İşaretsiz (Unsigned) Tamsayı Tipleri

	Nümerik Aralık		
Anahtar Kelime	**Alt Sınır**	**Üst Sınır**	**Bellek alanı (byte cinsinden)**
`unsigned char`	0	255	1
`unsigned short`	0	65,535	2
`unsigned int`	0	4,294,967,295	4
`unsigned long`	0	4,294,967,295	4

Simgelenen tiplerin değeri her zaman pozitif ise (mesela, bir şeyin kaç tane olduğu simgelendiğinde) veya işaretli tiplerin pozitif aralığı yeterince büyük değil ise `unsigned` tipler kullanılır.

Bir tamsayı tipini işaretsiz tipe dönüştürmek için veri tipinin önüne `unsigned` anahtar kelimesini eklemelisiniz. Örneğin, `char` tipinde bir `unsigned` değişken şöyle tanımlanır:

```
unsigned char ucharvar;
```

İşaretli tiplerin aralıklarının dışına çıkmak gözden kaçabilecek program hatalarına yol açabilir. Bazı (muhtemelen ender rastlanan) durumlarda bu tür hatalar işaretsiz tipler kullanılarak ortadan kaldırılabilir. Örneğin, aşağıdaki program 1,500,000,000 (1.5 milyar) sabit sayısını hem `int` olarak `signedVar` değişkeninde, hem de `unsigned int` olarak `unsignVar` değişkeninde saklar.

```
// signtest.cpp
// isaretli ve isaretsiz tamsayilari sinar
#include <iostream>
using namespace std;

int main()
   {
   int signedVar = 1500000000;              //isaretli
   unsigned int unsignVar = 1500000000;     //isaretsiz

   signedVar = (signedVar * 2) / 3;         //hesaplama sonucu araliktan tasar
   unsignVar = (unsignVar * 2) / 3;         //hesaplama sonucu aralik icinde

   cout << "signedVar = " << signedVar << endl;   //yanlis
   cout << "unsignVar = " << unsignVar << endl;   //tamam
   return 0;
   }
```

Program her iki değişkeni de 2 ile çarpıyor, sonra da 3'e bölüyor. Hesaplama sonucu orijinal sayıdan küçük olsa da, ara hesaplamadan elde edilen sonuç orijinal sayıdan daha büyük. Bu sıklıkla karşılaşılan bir durumdur. Fakat bu durum problem doğurabilir. SIGNTEST programında her iki değişkende de orijinal değerin üçte ikisinin -yani 1,000,000,000- saklanacağını sanıyoruz. Ama ne yazık ki, `signedVar` değişkeninde çarpma sonucu elde edilen sonuç -3,000,000,000- `int` tipinde bir değişkenin menzilini aşıyor (tamsayı aralığı -2,147,483,648 ile 2,147,483,648 arasıdır.) Programın çıktısı şöyledir:

```
signedVar = -431,655,765
unsignVar = 1,000,000,000
```

Çarpma işleminin ara sonucunu saklayabilecek büyüklükte olan işaretsiz değişken, sonucu doğru olarak kaydederken, işaretli değişken hatalı sonuç veriyor. İşin özeti şu: Programınızın ürettiği değerlerin tümünün, bu değerleri saklayacak değişkenlerin menzili içinde olduğuna dikkat edin. (Sonuçlar, `int` tipi için değişik byte sayısı kullanan 16 bit veya 64 bit bilgisayarlarda farklı farklı olacaktır.)

Tip Dönüşümü

C++, aynı C gibi, birkaç değişik veri tipi içeren deyimleri ele alırken bazı dillere kıyasla daha esnek davranır. Örnek olarak `MIXED` programını inceleyelim:

```
// mixed.cpp
// karisik deyimler
#include <iostream>
using namespace std;

   int main()
```



```
   {
   int count = 7;
   float avgWeight = 155.5F;
   double totalWeight = count * avgWeight;
   cout << "totalWeight=" << totalWeight << endl;
   return 0;
   }
```

Burada `int` tipinde bir değişken, `double` tipinde bir sonuç elde etmek için `float` tipinde bir değişkenle çarpılır. Bu program derleme hatası vermez; derleyici farklı tipteki sayıları çarpmanızı (ya da herhangi bir aritmetik işlem yapmanızı) normal karşılar.

Her dil bu kadar esnek değildir. Bazıları karışık deyimlere izin vermez. Böyle bir dil, `MIXED` programındaki aritmetik işlem yapılan satırı hatalı olarak işaretlerdi. Bu tür diller, tipleri karıştırdığınız zaman hata yaptığınızı varsayar ve sizi sözden korumaya çalışır. Fakat C++ ve C, yaptığınız işi yapmanız için iyi bir sebebiniz olması gerektiğini varsayar ve amacınıza yönelik işlemlerinizi çalıştırabilmeniz için size yardımcı olurlar. C++ ve C'nin popüler olmasının bir sebebi de budur. Size daha fazla özgürlük verirler. Elbette daha fazla özgürlükle hata yapma şansınız da artar.

Otomatik Dönüşümler

Derleyici, `MIXED` örneğinde olduğu gibi karışık tipli deyimlerle yüz yüze gelince neler olabilir, inceleyelim. Tipler, kabaca, Tablo 2.4'te gösterilen sıralamaya dayanarak, "daha yüksek" ve "daha düşük" olarak ele alınabilir.

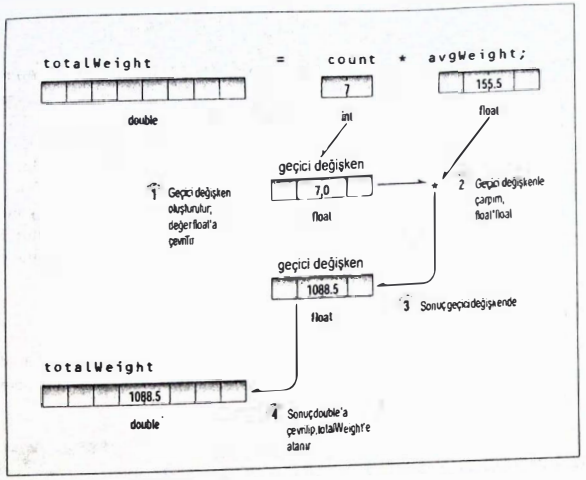
TABLO 2.4: Veri Tiplerinin Sıralanması

Veri Tipi	Sıralama
`long double`	En yüksek
`double`	
`float`	
`long`	
`int`	
`short`	
`char`	En düşük

`+` ve `*` gibi aritmetik operatörler, aynı tipte operandlar üzerine işlem yaparlar. Aynı deyim içinde farklı tipte iki operand karşı karşıya gelirse, düşük tipteki değişken, yüksek tipteki değişkenin tipine dönüştürülür. Yani, `MIXED` programında `count` değişkeninin `int` değeri `float` tipine dönüştürülür ve `float` tipinde bir değişken olan `avgWeight` ile çarpılmadan önce geçici bir değişkende saklanır. Hâlâ `float` tipinde olan sonuç, daha sonra `double`'a dönüştürülür; böylece, `double` tipindeki `totalWeight` isimli değişkene atanabilir. Bu işlem Şekil 2.9'da gösteriliyor.

Bu dönüşümler göz önünde gerçekleşmez. Üstelik, bunları çoğunlukla çok fazla düşünmenize de gerek yoktur; C++ yapmak istediklerinizi otomatik olarak gerçekleştirir. Ancak, bazen derleyici bu dönüşümlerden pek memnun olmaz. Birazdan bu konuyu göreceğiz. Ayrıca, nesneleri kullanmaya başlayınca aslında kendi veri tiplerimizi tanımlıyor olacağız. Tıpkı karışık deyimlerde normal değişkenleri kullandığımız gibi, bu yeni veri tiplerini de karışık deyimlerde

kullanmak isteyebiliriz. Böyle bir durumda, bir tipteki nesneleri başka tipteki nesnelere çevirmek için kendi çevrim rutinlerimizi oluştururken dikkatli olmamız gerekir. Buradaki standart veri tiplerinde olduğu gibi derleyici bunu bizim için yapmayacaktır.



ŞEKİL 2.9: Veri dönüşümü.

Tip Atamaları

Tip atamaları (*casts*) C++'ta, az önce bahsettiğimiz otomatik veri dönüşümlerinin aksine, programcı tarafından belirtilmiş veri dönüşümlerini ifade eder. Tip atamalarına ayrıca *tip kalıpları* da denir. Tip atamaları niçin gereklidir? Derleyicinin otomatik olarak yapmadığı durumlarda programcılar bazen bir değeri bir tipten diğerine dönüştürmek zorunda kalabilirler.

Standart C++'ta birkaç tür tip ataması vardır: Statik tip atamaları, dinamik tip atamaları, yeniden yorumlama (reinterpret) tip atamaları ve **const** tip atamaları. Biz burada sadece statik tip atamalarıyla ilgileneceğiz. Çok daha özel durumlarda kullanılan diğer tip ataması türlerini sonraki bölümlerde öğreneceğiz.

C++ tip atamalarının epeyce rahatsız edici bir görüntüsü vardır. **int** tipinde bir değişkeni **char** tipinde bir değişkene dönüştürmek için C++ tip atamaları kullanılan şu ifadeye bakalım:

```
aCharVar = static_cast<char>(anIntVar);
```

Burada tipi değiştirilecek olan değişken (**intVar**) parantez içine, değiştirilecek olan tip (**char**) de açılı parantez içine yerleştirilmiştir. Sonuç olarak **intVar**, **acharVar** değişkenine atanmadan önce tipi **char** olarak değiştirilir. Bu durumda, atama ifadesi dönüştürme işlemini kendisi de yapabilirdi. Ancak, tip atamalarının şart olduğu bazı özel durumlar da vardır.

SIGNTEST örneğinde, ara hesaplamadan elde edilen sonuç değişken tipinin kapasitesini aşmış ve hatalı yanıta neden olmuştu. Problemi, **int** yerine **unsigned int** kullanarak gidermiştik. Bu işe yaramıştı çünkü ara sonuç – 3.000.000.000 – işaretsiz değişkenin aralığı içinde kalıyordu.

Fakat varsayalım ara sonuç, işaretsiz tip aralığına da sığmamış olsun. Böyle bir durumda, bir tip ataması kullanarak problemi çözmeyi başarabiliriz. Bir örnekte görelim:

```
// cast.cpp
// isaretli ve isaretsiz tamsayilari sinar
#include <iostream>
using namespace std;

   int main()
   {
   int intVar = 1500000000;                    //1,500,000,000
   intVar = (intVar * 10) / 10;                //sonuc cok buyuk
   cout << "intVar = " << intVar << endl;      //yanlis yanit

   intVar = 1500000000;                        //double'a donustur
   intVar = (static_cast<double>(intVar) * 10) / 10;
   cout << "intVar = " << intVar << endl;      //dogru yanit
   return 0;
   }
```

intVar değişkenini 10 ile çarptığımızda, sonuç – 15,000,000,000 – **int** veya **unsigned int** tipindeki bir değişkene sığmayacak kadar büyük oluyor. Bu da, programın ilk kısmının çıktısında görüldüğü gibi hatalı bir sonuca yol açıyor.

Değişkenlerin veri tiplerini **double** olarak yeniden tanımlayabiliriz. **double** tipi 15 basamaklı sayıları saklayabildiği için bu işlem bol miktarda yer kazandırır. Fakat diyelim ki, bir nedenle, mesela programı küçük tutabilmek için, değişkenleri **double**'a çevirmeyi istemiyoruz. Böyle bir durumda başka bir çözüm daha vardır: **intVar**'ı çarpma işleminden önce **double** tipine çeviririz. Buna kimi zaman *zorlama* (*coercion*) denir; veri, bir başka tipe dönüştürülmek için zorlanır. Şu deyim

```
static_cast<double>(intVar)
```

intVar'ı **double** tipine dönüştürür. Aslında, **intVar** ile aynı değere sahip, **double** tipinde geçici bir değişken üretir. Bu değişken 10 ile çarpılır. Geçici değişken **double** tipinde olduğu için sonuç, tip aralığına sığar. Sonra sonuç 10'a bölünür ve normal **int** değişkeni olan **intVar**'a atanır. Programın çıktısı şöyledir:

```
intVar = 211509811
intVar = 1500000000
```

Tip ataması kullanılmayan ilk yanıt hatalıdır; ikincide ise tip ataması doğru sonuç verir. Standart C++'tan önce tip atamaları oldukça farklı bir biçimde ele alınırdı.

```
aCharVar = static_cast<char>(anIntVar);
```

yerine şöyle ifade ederdiniz:

```
aCharVar = (char)anIntVar;
```

veya alternatif olarak

```
aCharVar = char(anIntVar);
```

şeklinde ifade ederdiniz.

Bu yaklaşımlardaki problem, bu tür tip dönüşümlerinin güçlükle fark edilmesidir; söz dizimi programın kalan kısmıyla karışır. Bunları, ayrıca, kaynak kod editörünüzün Find (Bul) işlevini kullanarak aramak da zordur. Yeni format bu problemlerin üstesinden geliyor: `static_cast`'ın fark edilmesi ve aranması kolaydır. Bu tip eski tip atamaları hala çalışır ama, kullanımları önerilmez (daha teknik bir ifadeyle, *protesto edilir*.)

Tip atamaları sadece, mutlaka gerekli olduklarında kullanılmalıdır. Tip atamaları, *tip güvenliğinden* (tip güvenliği, değişken tiplerinin yanlışlıkla değişmemesini garanti eder) kontrollü biçimde uzaklaşma yöntemidir. Üstelik, tip atamaları derleyicinin potansiyel hataları saptayabilmesini imkansız kıldığı için sorunlara da yol açabilir. Yine de, kimi zaman tip ataması kullanımı önlenemez. İleriki bölümlerde tip atamalarının gerekli olduğu durumlarla ilgili bazı örnekler göreceğiz.

Aritmetik Operatörler

Şu ana dek muhtemelen fark etmişsinizdir, C++ dört tane normal aritmetik operatörü kullanır: Toplama için +, çıkarma için –, çarpma için * ve bölme için /. Bu operatörler hem tamsayı hem de kayan noktalı veri tiplerinde kullanılır. Kullanılışları, diğer dillerdeki kullanım şekliyle hemen hemen aynıdır; cebirdeki kullanımlarına ise çok yakındır. Bununla birlikte, kullanımları çok açık olmayan başka aritmetik operatörleri de mevcuttur.

Kalan Operatörü

Sadece tamsayı değişkenlerle (`char`, `short`, `int` ve `long` tipleri ile) işlem yapan beşinci bir operatör daha vardır. Buna *kalan operatörü* (*remainder operator*) denir ve yüzde işareti (%) ile simgelenir. Bu operatör (*modül operatörü* (*modulus operator*) olarak da adlandırılır), bir sayı diğerine bölündüğünde, bölme işleminin kalanını hesaplar. **REMAIND** programı bunun nasıl gerçekleştirildiğini gösteriyor.

```
// remaind.cpp
// kalan operatoru
#include <iostream>
using namespace std;

int main()
   {
   cout <<  6 % 8 << endl     // 6
        <<  7 % 8 << endl     // 7
        <<  8 % 8 << endl     // 0
        <<  9 % 8 << endl     // 1
        << 10 % 8 << endl;    // 2
   return 0;
   }
```

Bu programda, kalan operatörü kullanılarak, 6 ile 10 arasındaki sayılar 8 ile bölünür. Bölme işlemlerinin kalanları 6,7,0,1, ve 2 elde edilen cevaplardır. Kalan operatörü çok çeşitli durumlarda kullanılabilir. İlerledikçe bunların örneklerini vereceğiz.

Öncelikle ilgili bir not:

```
cout << 6 % 8
```

ifadesinde, ilk olarak kalan operatörünün değeri hesaplanır, çünkü kalan operatörünün önceliği, << operatörününkinden daha yüksektir. Böyle olmasaydı, << operatörünün etki etmesinden önce hesaplanmayı garanti etmek için `6 % 8` deyiminde parantez içinde yazmak zorunda kalacaktık.

Aritmetik Atama Operatörleri

C++, kodunuzu kısaltmak ve netleştirmek için çeşitli yollar öne sürer. Bunlardan bir tanesi *aritmetik atama operatörüdür*. C++ program listelerini diğerlerinden ayıran görünümü bu operatör yardımıyla sağlanır.

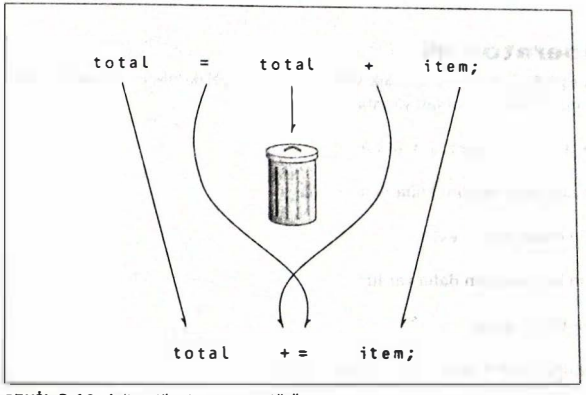
Aşağıdaki türde bir ifade bir çok dilde çok kullanılır:

```
total= total + item;     // "item"i "total"e ekler
```

Böyle bir durumda, mevcut bir değere bir şey ekliyorsunuz (veya başka bir aritmetik işlemi tercih etmiş olabilirsiniz.) Fakat bu ifadenin sözdizimi, cesaretin önemine inanlara rahatsız edici gelir, çünkü `total` ismi ifadede iki kez yer alır. Bu nedenle, C++ yoğunlaştırılmış bir yaklaşım sunuyor: Aritmetik atama operatörü. Aritmetik atama operatörü, aritmetik operatörü ve atama operatörünü birleştirerek tekrarlanan operandı ortadan kaldırıyor. İşte, önceki ifadeyle tamamen aynı etkiye sahip bir ifade:

```
total += item;     // "item"i "total"e ekler
```

Şekil 2.10 her iki formun denk olduğuna dikkat çekiyor.



ŞEKİL 2.10: Aritmetik atama operatörü.

Aritmetik işlemlere karşılık gelen aritmetik atama operatörleri mevcuttur: +=, -=, *=, /= ve %= (başka bazı operatörler de vardır). Aşağıdaki örnek aritmetik atama operatörlerinin kullanımını gösteriyor:

```
// assign.cpp
// aritmetik atama operatorleri
#include <iostream>
using namespace std;

int main()
   {
   int ans = 27;

   ans += 10;                  //ans = ans + 10; ile ayni
   cout << ans << ", ";
   ans -= 7;                   //ans = ans - 7; ile ayni
   cout << ans << ", ";
   ans *= 2;                   //ans = ans * 2; ile ayni
   cout << ans << ", ";
   ans /= 3;                   //ans = ans / 3; ile ayni
   cout << ans <<",";
   ans %= 3;                   //ans = ans % 3; ile ayni
   cout << ans << endl;
   return 0;
   }
```

Bu programın çıktısı şöyledir:

```
37, 30, 60, 20, 2
```

Program kodunuzda aritmetik alama operatörlerini kullanmak zorunda değilsiniz. Ancak bunlar, dilin sıklıkla kullanılan bir özelliğidir. Bu kitaptaki birçok örnekte de karşımıza çıkacaklar.

Artırma Operatörleri

İşte çok daha özelleştirilmiş bir operatör. Sık sık mevcut bir değişkenin değerine 1 eklemek zorunda kalırsınız. Bunu "normal" yoldan yapabilirsiniz:

```
count= count + 1;    //"count"a 1 ekler
```

Ya da, aritmetik atama operatörü kullanabilirsiniz:

```
count += 1;    //"count"a 1 ekler
```

Fakat daha yoğun bir yaklaşım daha vardır:

```
++count  //"count"a 1 ekler
```

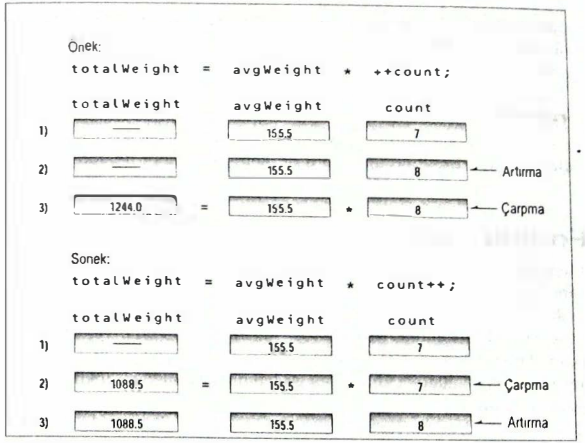
++ operatörü argümanını bir artırır (argümana 1 ekler).

Önek ve Sonek

Bu kadarı yeterince tuhaf değilmiş gibi, artırma operatörü iki değişik şekilde kullanılabilir: *önek* ve *sonek* olarak. Önek, operatörün değişkenin önünde; sonek, arkasında yer alması demektir. İkisi arasındaki fark nedir? Genellikle bir değişken, üzerinde başka işlemlerin de yapıldığı bir ifadede aynı zamanda artırılır. Örneğin,

```
totalWeight = avgWeight * ++count;
```

Buradaki sorun şu: Çarpma işlemi **count** arttırıldıktan önce mi, sonra mı gerçekleştirilecektir? Bu örnekte önce **count** artırılacaktır. Bunu nereden biliyoruz? Çünkü önek notasyonu kullanılmıştır: **++count**. Şayet sonek notasyonu kullanmış olsaydık, **count++**, öncelikle çarpma işleminin yapılması gerekecekti, ardından **count** artırılacaktı. Şekil 2.11'de bu gösteriliyor.



ŞEKİL 2.11: Artırma operatörü.

Artırma operatörünün önek ve sonek versiyonlarını gösteren bir örnek verelim:

```
// increm.cpp
// artirma operatoru
#include <iostream>
using namespace std;

int main()
   {
   int count=10;

   cout << "count=" << count << endl;      //ekranda 10 goruntulenir
   cout << "count=" << ++count << endl;    //ekranda 11 (onek) goruntulenir
   cout << "count=" << count << endl;      //ekranda 11 goruntulenir
```

```
cout << "count=" << count++ << endl;     //ekranda 11 (sonek) goruntulenir
cout << "count=" << count << endl;       //ekranda 12 goruntulenir
return 0;
}
```

Programın çıktısı şöyle olur:

```
count=10
count=11
count=11
count=11
count=12
```

count'un ilk arttırıldığı yerde önek operatörü kullanılır. Bu sayede, artırma işlemi ifadenin en başında, çıktı işleminden önce yapılır. **++count** deyiminin değeri görüntülendiğinde, artırma işlemi önceden yapılmış olduğu için << operatörü, 11 değerini görür. **count** ikinci kez artırıldığında sonek operatörü kullanılır. **count++** deyimi görüntülendiğinde, deyimin artırılmamış 11 değerini halen koruduğu görülür. Bu ifadenin tamamlanmasını takiben artırma işlemi gerçekleştirilir. Böylece, programın son ifadesinde **count**'un 12 değerini aldığını görürüz.

Eksiltme Operatörü

Eksiltme operatörü, —, artırma operatörüne çok benzer biçimde davranır. Tek farkla; eksiltme operatörü, operandını 1 azaltır. Eksiltme operatörü de hem önek hem de sonek biçimlerinde kullanılabilir.

Kütüphane Fonksiyonları

C++'taki pek çok etkinlik *kütüphane fonksiyonları* tarafından gerçekleştirilir. Bu fonksiyonlar diğer birçok işin yanı sıra dosya erişimi, matematiksel hesaplamalar ve veri dönüşümü işlerini yerine getirir. Fonksiyonların nasıl çalıştığını görmeden önce (Bölüm 5) kütüphane fonksiyonlarının içine çok fazla dalmak istemiyoruz. Ancak, basit kütüphane fonksiyonlarını nasıl çalıştıklarını tam olarak anlamadan kullanabilirsiniz.

Bir sonraki örnek olan **SQRT**, kullanıcı tarafından girilen bir sayının kare kökünü hesaplamak için **sqrt()** kütüphane fonksiyonunu kullanır.

```
//sqrt.cpp
//sqrt()kutuphane fonksiyonu
#include <iostream>              //cout vs icin
#include <cmath>                 //sqrt() icin
using namespace std;

int main()
  {
  double number, answer;         //sqrt(), double tipi gerektirir

  cout << "Enter a number:";
  cin >> number;                 //sayiyi al
  answer = sqrt(number);         //kare kokunu bul
  cout << "Square root is "
   << answer << endl;            //ekranda goruntule
  return 0;
  }
```


Programı öncelikle kullanıcıdan bir sayı alır. Bu sayı daha sonra **sqrt()** fonksiyonunda argüman olarak kullanılır. Şu ifadede gösterildiği gibi:

```
answer = sqrt(number);
```

Argüman, fonksiyona verilen giriştir; fonksiyon ismini takip eden parantezlerin içinde yer alır. Fonksiyon, bu argümanla işlem yapar ve bir değer döndürür; bu, fonksiyonun çıktısıdır. Bu örnekte, orijinal sayının kare kökü döndürülen değerdir. Bir değer döndürmek demek, fonksiyon deyimi bu değeri alıyor demektir. Bu değer, daha sonra bir değişkene atanabilir. Örneğimizde, fonksiyondan dönen değer **answer** isimli değişkene atanır. Program daha sonra bu değeri ekranda görüntüler. Programın bir çıktısı şöyle olabilir:

```
Enter a number: 1000
Square root is 31.622777
```

Hesap makinenizde 31.622777 sayısını kendisi ile çarparsanız yanıtın çok yakın olduğunu ispatlayabilirsiniz.

Fonksiyon argümanları ve fonksiyonun döndürdüğü değerler doğru veri tipinde olmalıdır. Bu veri tiplerinin neler olduğunu derleyicinin yardım dosyasındaki kütüphane fonksiyonlarının tanımlarına bakarak anlayabilirsiniz. Derleyicinin yardım dosyasında yüzlerce kütüphane fonksiyonunun her birinin tanımları mevcuttur. **sqrt()** fonksiyonunun tanımında argümanın ve fonksiyonunun döndürdüğü değerin **double** tipinde olması belirtiliyor. Bu nedenle, programda bu tipte değişkenler kullandık.

Başlık Dosyaları

cout ve benzer diğer nesnelerde olduğu gibi, kullandığınız kütüphane fonksiyonlarının tanımlamalarını içeren başlık dosyalarını programınıza **#include** direktifi ile dahil etmeniz gerekir. **sqrt()** fonksiyonunun dokümantasyonunda belirtilen başlık dosyasının **CMATH** olduğunu göreceksiniz. **SQRT** programında, önişlemci direktifi

```
#include <cmath>
```

kaynak dosyamızın içine bu başlık dosyasını dahil etme işlemini gerçekleştirir.

Bir kütüphane fonksiyonu kullandığınızda ilgili başlık dosyasını dahil etmezseniz, derleyiciden şuna benzer bir hata mesajı alırsınız: *'sqrt' unidentified identifier* (*'sqrt' tanımlı olmayan bir tanımlayıcıdır*).

Kütüphane Dosyaları

Önceden bahsettiğimiz gibi, kütüphane fonksiyonlarını ve nesnelerini içeren çeşitli dosyalar, çalıştırılabilir bir dosya oluşturmak amacıyla sizin programınıza bağlanır. Bu dosyalar, fonksiyonların hakiki makine kodlarını içerir. Bu tür dosyalar genellikle **.LIB** uzantısına sahiptir. **sqrt()** fonksiyonu da böyle bir dosya içinde bulunur. Bağlayıcı (linker) tarafından otomatik olarak dosyadan alınır, doğru bağlantılar yapılarak **SQRT** programınızdan çağrılabilir (yani, erişilebilir) hale getirilir. Bütün bu ayrıntıları sizin yerinize derleyiciniz gerçekleştirir. Bu nedenle, işlemleri dert etmenize çoğunlukla gerek yoktur. Yine de, bu dosyaların ne işe yaradığını bilmelisiniz.

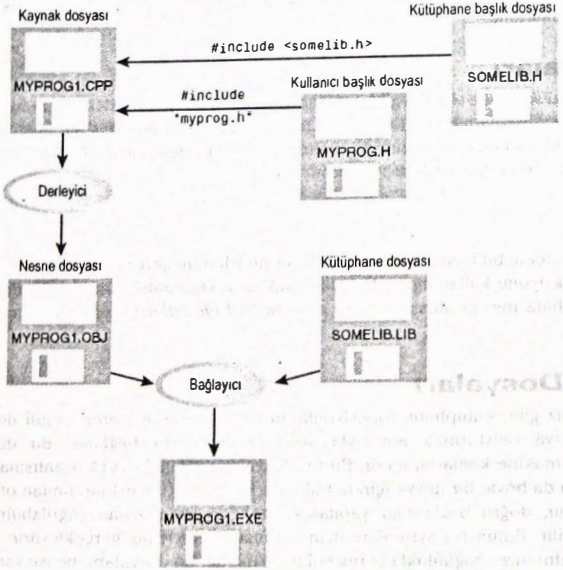
Başlık Dosyaları ve Kütüphane Dosyaları

Kütüphane dosyaları ve başlık dosyaları arasındaki ilişki kafa karıştırıcı olabilir, bu nedenle bunların bir kez daha tekrarlayalım. `sqrt()` gibi bir kütüphane fonksiyonunu kullanmak için bu fonksiyonu içeren kütüphane dosyasını programınıza bağlamanız gerekir. Daha sonra kütüphane dosyasındaki ilgili fonksiyonlar linker tarafından programınıza bağlanacaktır.

Ancak bu, hikayenin sonu değildir. Kaynak dosyanızdaki fonksiyonlar, kütüphane dosyasındaki fonksiyonların ve diğer öğelerin isimlerini ve tiplerini bilmek zorundadır. Kaynak dosyanızdaki fonksiyonlara bu bilgiler başlık dosyası ile verilir. Her başlık dosyası belli bir fonksiyon grubu için bilgi içerir. Fonksiyonların kendileri bir kütüphane dosyası içinde gruplanmıştır; ancak, bunlar hakkındaki bilgiler birkaç başlık dosyasına dağıtılmıştır. `IOSTREAM` başlık dosyası, `cout` da dahil olmak üzere çeşitli I/O (giriş/çıkış) fonksiyonları ve nesneleri için bilgi taşır. `CMATH` başlık dosyası, `sqrt()` gibi matematiksel fonksiyonlarla ilgili bilgi taşır. Eğer `strcpy()` gibi karakter katarı fonksiyonları kullanıyorsanız, `STRING.H` dosyasını dahil etmeniz gerekir, vs.

Şekil 2.12 başlık ve kütüphane dosyaları ile program geliştirirken kullanılan diğer dosyalar arasındaki ilişkiyi gösteriyor.

Başlık dosyaları kullanımı C++'ta çok yaygındır. Bir kütüphane fonksiyonu ya da önceden tanımlanmış nesne veya operatör kullandığınızda uygun tanımlamaları içeren başlık dosyasını da kullanmanız gerekir.



ŞEKİL 2.12: Başlık ve kütüphane dosyaları.

#include Direktifini Kullanmanın İki Yolu

`#include` direktifini iki şekilde kullanabilirsiniz. `SORT` örneğindeki `IOSTREAM` ve `CMATH` dosya isimlerini çevreleyen açılı parantezler, < ve >, derleyicinin bu dosyaları standart `INCLUDE` dizininde aramaya başlaması gerektiğini bildirir. Geleneksel olarak `INCLUDE` ismi verilen bu dizin, derleyici üreticisinin sistem için tedarik ettiği başlık dosyalarını içerir.

Dosya ismi etrafında açılı parantez kullanmak yerine tırnak işareti de kullanabilirsiniz. Örneğin,

```
#include "myheader.h"
```

Tırnak işaretleri derleyiciye, başlık dosyasını arama işlemine içinde bulunulan dizinden başlamasını bildirir. Bu dizin genellikle kaynak dosyanın olduğu dizindir. Tırnak işaretlerini, normalde, kendi yazdığınız başlık dosyaları için kullanırsınız (bunu Bölüm 13'te "Birden Fazla Dosya Kullanan Programlar" başlığı altında inceleyeceğiz.) Tırnak işaretleri veya açılı parantezler her durumda işe yarar. Ancak, uygun olanı seçmek, derleyiciye dosyanın yeri hakkında ipucu verdiği için derleme işlemini bir miktar hızlandırır.

Ek C'de "Microsoft Visual C++" ve Ek D'de "Borland C++ Builder" bölümleri spesifik derleyicilerde başlık dosyalarının nasıl ele alındığını açıklar.

Özet

Bu bölümde C++'ın en temel yapı bloklarından birinin *fonksiyon* olduğunu öğrendik. Bir programın çalıştırılması sırasında `main()` isimli fonksiyon her zaman ilk çalıştırılan ifadedir.

Bir fonksiyon, bilgisayara bir şeyler yapmasını söyleyen *ifadelerden* meydana gelir. Her ifade noktalı virgül ile biter. Bir ifade bir veya daha fazla *deyim* içerebilir. Bir deyim, genellikle spesifik bir değeri olan, değişken ve operatörler dizisidir.

C++'ta çıktı (output) en yaygın şekliyle `cout` nesnesi ve `<<` ekleme operatörü ile ele alınır. Bu ikisi birlikte, değişken veya sabitleri standart çıktı cihazına –genellikle bu, ekrandır- gönderir. Giriş, `cin` ve `>>` çıkarma operatörü yardımıyla ele alınır. `cin` ve `>>`, değerlerin standart giriş cihazından –genellikle bu, klavyedir- alınmasını sağlar.

C++'ın içine çeşitli veri tipleri yerleştirilmiştir: `char`, `int`, `long` ve `short` tamsayı tipleridir; `float`, `double` ve `long double` kayan noktalı tiplerdir. Bütün bu tipler işaretlidir. Tamsayı tiplerin işaretsiz versiyonları `unsigned` anahtar kelimesi ile belirtilir. İşaretsiz tipler negatif sayıları saklamaz, fakat iki kat fazla pozitif sayı saklar. `bool` tipi Boolean değişkenler için kullanılır ve sadece `true` (doğru) ve `false` (yanlış) değerlerini alabilir.

`const` anahtar kelimesi bir değişkenin değerinin program boyunca değişmeyeceğini şart koşar. Tam anlamıyla söylersek, değişken artık bir değişken değil, bir sabittir.

Bir değişken karışık deyimlerle (farklı veri tiplerinin yer aldığı deyimlerde) veya tip atamaları yardımıyla otomatik olarak bir tipten diğerine çevrilir. Tip atamaları programcılara veri tiplerini dönüştürme imkanı verir.

C++ olağan aritmetik operatörlerini kullanır: `+`, `–`, `*` ve `/`. Ek olarak, kalan operatörü, `%`, tamsayıların bölünmesinden kalanı döndürür.

Aritmetik atama operatörleri, `+=`, `–=` vs, aritmetik ve atama işlemlerini aynı anda gerçekleştirir. Artırma ve eksiltme operatörleri, `++` ve `––`, değişkeni 1 arttırır veya 1 azaltır.

Önişlemci direktifleri, bilgisayardan çok derleyici için komutlar içerir. `#include` direktifi, mevcut kaynak dosyasına başka bir dosya eklemesini derleyiciye bildirir. `#define` direktifi,

derleyiciye bir şeyin yerine başka bir şeyi yerleştirmesini bildirir. `using` direktifi, derleyiciye belirli bir isim uzayındaki isimleri tanımasını bildirir.

Programınızda bir kütüphane dosyası kullanıyorsanız, bu fonksiyonun kodu, programınıza otomatik olarak bağlanacak olan bir kütüphane dosyasının içindedir. Fonksiyonun tanımlamalarını içeren başlık dosyası `#include` ifadesi yardımıyla kaynak dosyanıza eklenmelidir.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Bir programı fonksiyonlara bölmek
   a. nesne yönelimli programlamanın temelidir.
   b. programı kavramayı kolaylaştırır.
   c. programın büyüklüğünü azaltabilir.
   d. programın hızlı çalışmasını sağlar.
2. Bir fonksiyon isminin ardından _____ gelmelidir.
3. Bir fonksiyon gövdesi _____ ile sınırlanır.
4. `main()` fonksiyonu niçin özeldir?
5. Bilgisayara bir iş yaptıran bir C++ komutuna _____ denir.
6. Normal bir C++ açıklaması ile eski moda açıklamalara birer örnek verin.
7. Bir deyim
   a. genellikle nümerik bir değere sahiptir.
   b. programın duygusal durumunu belirtir.
   c. her zaman fonksiyonun dışında yer alır.
   d. bir ifadenin parçası olabilir.
8. 32 bit sistemde aşağıdaki veri tiplerinin kaç byte yer kapladığını belirtin.
   a. `int` tipi
   b. `long double` tipi
   c. `float` tipi
   d. `long` tipi
9. Doğru/Yanlış: `char` tipinde bir değişken 301 değerini saklayabilir.
10. Aşağıdakiler ne tür program öğeleridir?
    a. 12
    b. 'a'
    c. 4.28915
    d. `JungleJim`
    e. `JungleJim()`
11. Aşağıdakileri ekranda görüntüleyen ifadeler yazın:
    a. '`x`' karakteri
    b. *Jim* ismi
    c. 509 sayısı
12. Doğru/Yanlış: Bir atama ifadesinde eşit işaretinin sol tarafındaki değer her zaman sağ taraftakine eşittir.
13. `george` değişkenini 10 karakter genişliğinde görüntüleyecek bir ifade yazın.

14. `cout` ve `cin`'i kullanmak için kaynak dosyanıza hangi başlık dosyasını `#include` direktifi ile dahil etmeniz gerekir?
15. Klavyeden nümerik bir değer alan ve bunu `temp` isimli değişkene yerleştiren bir ifade yazın.
16. `setw`'u kullanmak için programınıza hangi başlık dosyasını eklemeniz gerekir?
17. Derleyicinin görünmeyen karakterleri dikkate alamadığına dair kuralın iki istisnası _____ ve _____dir.
18. Doğru/Yanlış: Aynı aritmetik deyim içinde farklı veri tiplerine ait değişkenleri kullanmak tamamen uygundur.
19. `11%3` deyiminin değeri _____.
20. Bir aritmetik atama operatörü hangi iki operatörün etkilerini birleştirmiştir?
21. `temp` değişkeninin değerini 23 arttıran ve aritmetik atama operatörü kullanan bir ifade yazın. Aynı ifadeyi aritmetik atama operatörü kullanmadan yazın.
22. Artırma operatörü bir değişkenin değerini ne kadar arttırır?
23. `var1`'in 20 değeri ile başladığını varsayarsak, aşağıdaki kod parçasının çıktısı ne olur?

```
cout << var1 --;
cout << ++ var1;
```

24. Şimdiye kadar gördüğümüz örneklerde, başlık dosyaları hangi amaçla kullanılıyordu?
25. Kütüphane fonksiyonlarının gerçek kodları _____ dosyasında bulunur.

Alıştırmalar

Yıldız işaretli soruların cevaplarını Ek G'de bulabilirsiniz.

1. Bir metreküpün 7.481 galon aldığını varsayarak, kullanıcıya galon sayısını soran ve bunun metreküp cinsinden karşılığını görüntüleyen bir program yazın.*
2. Aşağıdaki tabloyu üreten bir program yazın.

```
1990    135
1991    7290
1992    11300
1993    16200
```

   Tüm çıktı için tek `cout` kullanın.
3. Aşağıdaki çıktıyı üreten bir program yazın.

```
10
20
19
```

   10 için bir tamsayı sabit kullanın. 20'yi üretmek için aritmetik atama operatörü, 19'u üretmek için eksiltme operatörü kullanın.
4. En sevdiğiniz şiiri görüntüleyen bir program yazın. Satır sonu için uygun bir kaçış sekansı kullanın. Eğer bir şiiriniz yoksa, Ogden Nash'ın şu şiirini ödünç alabilirsiniz:

```
Candy is dandy,
But liqour is quicker.
```

5. Bir kütüphane fonksiyonu, `islower()`, argüman olarak tek bir karakter (bir harf) alır ve eğer karakter küçük harf ise sıfırdan farklı bir tamsayı, büyük harf ise sıfır döndürür. Bu fonksiyon `CTYPE.H` başlık dosyasını gerektirir. Kullanıcının bir harf girmesini isteyen ve bu harfin küçük veya büyük harf olmasına bağlı olarak ekranda sıfır veya sıfırdan farklı bir tamsayı görüntüleyen bir program yazın. (İpucu için `SQRT` programına bakın.)
6. Belirli bir günde İngiliz Sterlin'i 1.487 Amerika dolarına denkti. Fransız Frank'ı $0.172, Alman Mark'ı $0.584 ve Japon Yen'i $0.00955 idi. Kullanıcının dolar olarak bir değer girmesini sağlayan ve bu değerin diğer dört para birimine çevrilmiş miktarını görüntüleyen bir program yazın.
7. Sıcaklığı 9/5 ile çarpıp 32 ekleyerek, Celsius'tan Fahrenheit'a çevirebilirsiniz. Kullanıcının, sıcaklığı Celsius cinsinden simgeleyen bir reel sayı girmesini sağlayan ve karşılık gelen Fahrenheit derecesini görüntüleyen bir program yazın.
8. Eğer bir değer `setw` ile belirtilen alandan daha küçükse, kullanılmayan yerler, varsayılan durumda boşluk karakteriyle doldurulur. `setfill()` manipülatörü, argüman olarak tek bir karakter alır ve boşluk karakteri olan alanları bu karakter ile değiştirir. `WIDTH` programını, her satırda şehir ismi ve nüfus sayısı arasını boşluk yerine nokta ile dolduracak şekilde yeniden yazın.
9. Eğer iki adet kesirli sayınız varsa, a/b ve c/d gibi, bunların toplamını şu formülden bulabilirsiniz:

```
a   c       a*d + b*c
--- + ---= -----------
b   d         b*d
```

Örneğin, 1/4 artı 2/3 şu toplamı verir:

```
1   2       1*3 + 4*2        3 + 8       11
--- + ---= ------------= --------= ----
4   3         4*3             12         12
```

Kullanıcıdan iki adet kesirli sayı girmesini isteyen ve bu sayıların toplamını kesirli biçimde görüntüleyen bir program yazın. (Kesirleri sadeleştirmenize gerek yok.) Programın kullanıcı ile etkileşimi şöyle olabilir:

```
Birinci kesri girin: 1/2
  Ikinci kesri girin: 2/5
  Toplam= 9/10
```

Çıkarma operatörünün (>>) bir kerede birden fazla nicelik okumak için peş peşe kullanılabilme özelliğinden yararlanabilirsiniz:

```
cin >> a >> dummychar >> b;
```

10. Britanya İmparatorluğunun altın çağında Büyük Britanya pound, shilling ve pence'e dayalı bir para sistemi kullanıyordu. 20 shilling bir pound'a; 12 pence bir shilling'e karşılık geliyordu. Bu eski sistem pound işareti, £, ve noktadan sonra iki basamak notasyonunu kullanıyordu. Örneğin, £5.2.8 deyimi 5 pound, 2 shilling, 8 pence demekti. (*Pence*, *penny*'nin çoğul halidir.) 1950'lerde kullanılmaya başlanan yeni para sistemi sadece pound'dan oluşuyor ve 100 pence bir pound'a karşılık geliyor (ABD doları ve cent gibi). Bu yeni sisteme *ondalık pound* diyelim. Dolayısıyla, eski notasyonla £5.2.8, ondalık

pound cinsinden £5.13 (aslında £5.1333333) demektir. Eski pound-shilling-pence biçimini ondalık pound'a çeviren bir program yazın. Kullanıcının programla etkileşim örneği şöyle olabilir:

```
Pound miktarini girin: 7
Shilling miktarini girin: 17
Pence miktarini girin: 9
Ondalik pound = £7.89
```

Bir çok derleyicide pound işaretini (£) simgelemek için 156 ondalık sayısını (onaltılık karakter sabiti olarak ise `'\x9c'`) kullanabilirsiniz. Bazı derleyicilerde ise, pound işaretini Windows Karakter Eşlem aksesuarını kullanarak doğrudan programınıza yapıştırabilirsiniz.

11. Varsayılan durumda çıktı, kendi alanı içinde sağa dayalıdır. `setiosflag(ios::left)` manipülatörünü kullanarak metin çıktılarını sola dayalı yazabilirsiniz. (Şimdilik bu yeni notasyonunun ne anlama geldiğini dert etmeyin.) Bu manipülatörü `setw()` ile birlikte kullanarak aşağıdaki çıktıyı üretin.

```
Soyisim   Isim          Adres                 Sehir          Eyalet
------------------------------------------------------------------
Jones     Bernard 109 Pine Lane  Littletown    MI
O'Brian   Coleen  42 E.99th Ave. Bigcity       NY
Wong      Harry   121-A Alabama  St.Lakeville  IL
```

12. 10. alıştırmanın tersinin yazın. Kullanıcı Büyük Britanya'nın yeni ondalık-pound notasyonunda (pound ve pence) bir değer girsin, program bunu eski pound-shilling-pence notasyonuna çevirsin. Programın bir etkileşim örneği şöyle olabilir:

```
Ondalik pound degerini girin: 3.51
Eski notasyondaki karsiligi= £3.10.2
```

Kayan noktalı bir değeri (diyelim ki, 12.34) bir tamsayı değişkene atadığınız zaman ondalık kısım (0.34) kaybolur; tamsayı değeri sadece 12 olur. Bu bilgiden programınızda yararlanabilirsiniz. Derleyicinin uyarı mesajı vermesini önlemek için tip ataması kullanın. Şu tür ifadeler yazın:

```
float decpounds;      // kullanicidan gelecek giris (yeni pound stili)
int pounds;           // eski stil (tamsayi)pound
float decfrac;        // ondalik kesir (1.0'dan kucuk)

pounds = static_cast<int>(decpounds);  // ondalik kesri kaldir
decfrac = decpounds -pounds;           // ondalik kesri yerlestir
```

`defrac`'ı 20 ile çarparak shilling değerini bulabilirsiniz. Pence için de benzer işlem gerekir.

ÜÇÜNCÜ BÖLÜM

3

DÖNGÜLER VE KARARLAR

İlişkisel Operatörler

Döngüler

Kararlar

Mantıksal Operatörler

Öncelik Sıralarının Özeti

Diğer Kontrol İfadeleri

Pek az program, başından sonuna kadar tüm ifadelerini sıkı bir sıraya göre çalıştırır. Bir çok program (insanlar gibi) değişen koşullar karşısında karar vermek durumundadır. Program içinde yapılan hesaplamalara bağlı olarak akış kontrolü, programın bir bölümünden diğerine atlar. Bu tür atlamalara neden olan program ifadelerine *kontrol ifadeleri* denir. İki ana kategori vardır: Döngüler ve kararlar.

Bir döngünün kaç kere çalıştırılacağı veya bir karar neticesinde kodun bir kısmının çalıştırılıp çalıştırılmayacağı belli deyimlerin true veya false olmasına bağlıdır. Bu deyimler tipik olarak, iki değeri karşılaştıran ve *ilişkisel operatör* olarak adlandırılan bir tür operatör içerirler. Döngü ve kararlarla ilgili işlemler bu operatörlerle yakından alakalı olduğu için önce bu operatörleri inceleyeceğiz.

İlişkisel Operatörler

Bir ilişkisel operatör iki değeri karşılaştırır. Bu değerler herhangi bir standart C++ tipinde olabilir, **char**, **int** ve **float** gibi veya –daha sonra göreceğimiz gibi- kullanıcının tanımladığı bir sınıf tipinde olabilir. Karşılaştırmalarda eşittir, küçüktür veya büyüktür gibi ilişkiler kullanılır. Karşılaştırmanın sonucu doğru veya yanlış değerindedir. Örneğin, iki değer ya birbirine eşittir (doğru) ya da değildir (yanlış).

İlk programımız **RELAT**, tamsayı değişkenlerin ve sabitlerin karşılaştırılmasında ilişkisel operatörleri tanıtıyor.

```
// relat.cpp
// iliskisel operatorler
#include <iostream>
using namespace std;

int main()
   {
   int numb;

   cout << "Enter a number:";
   cin >> numb;
   cout << "numb<10 is " << (numb < 10) << endl;
   cout << "numb>10 is " << (numb > 10) << endl;
   cout << "numb==10 is " << (numb == 10) << endl;
   return 0;
   }
```

Bu program, kullanıcının girdiği sayı ile 10 arasında üç tür karşılaştırma yapar. Kullanıcı 20 değerini girdiğinde programın çıktısı şöyle olur:

```
Enter a number:20
numb<10 is 0
numb>10 is 1
numb==10 is 0
```

Eğer **numb**, 10'dan küçükse birinci deyim doğrudur. **numb**, 10'dan büyükse ikinci deyim doğrudur; **numb**, 10'a eşitse üçüncü deyim doğrudur. Çıktıdan da görebileceğiniz gibi, C++ derleyicisine göre doğru bir deyim 1 değerine, yanlış deyim ise 0 değerine sahiptir.

Standart C++'ta **true** ve **false** sabit değerlerinden yalnız birini saklayabilen bir **bool** tipi olduğunu önceki bölümde bahsetmiştik. **numb<10** gibi ilişkisel deyimlerin sonuçlarının **bool** tipinde olacağını ve programın 0 yerine **false**, 1 yerine **true** basması gerektiğini düşünebilirsiniz. Aslında C++ bu konuda biraz şizofreniktir. İlişkisel işlemlerin sonuçlarını, hatta **bool** tipinde değişkenlerin değerlerini **cout <<** ile görüntülemek **false** veya **true** yerine **0** veya **1** sonucunu doğurur. Bunun sebebi tarihseldir, çünkü C++ **bool** tipinden önce ortaya çıkmıştı. Standart C++'ın ortaya çıkışından önce, yanlış ve doğru değerlerini ifade etmenin *tek* yolu 0 ve 1 kullanmaktı. Şimdi ise yanlış, **false** değerinde bir **bool** ile veya 0 değerinde bir tamsayı ile simgelenebilir. Doğru ise, **true** değerinde bir **bool** ile veya 1 değerinde bir tamsayı ile simgelenebilir.

Bir çok basit durumda her ikisi arasındaki fark belli değildir, çünkü doğru/yanlış değerlerini *görüntülemek* gereğini duymayız. Bu değerleri döngülerde ve kararlarda, programı bir sonraki adımda yapacağı işe yönlendirmek için *kullanırız*.

C++ ilişkisel operatörlerinin tam listesi aşağıdadır:

Operatör	Anlamı
>	Büyüktür
<	Küçüktür
==	Eşittir
!=	Eşit değildir
>=	Büyüktür veya eşittir
<=	Küçüktür veya eşittir

Şimdi, ilişkisel operatörler kullanan bazı deyimlere ve bu deyimlerin değerlerine göz atalım. İlk iki satır **harry** ve **jane** değişkenlerine değer atama ifadeleridir. Açıklamaları bir kağıtla kapatıp deyimlerin değerlerini doğru tahmin edip etmediğinizi sınayabilirsiniz.

```
jane = 44;          //deger atama ifadesi
harry = 12;         //deger atama ifadesi
(jane == harry)     //false
(harry <= 12)       //true
(jane > harry)      //true
(jane >= 44)        //true
(harry != 12)       //true
(7 < harry)         //true
(0)                 //yanlis (tanim geregi)
(44)                //true (0 olmadigi icin)
```

Eşittir operatörünün iki tane eşittir işareti kullandığına dikkat edin. İlişkisel operatör olarak bir tane eşittir işareti –değer atama işareti- kullanmak sıkça yapılan bir hatadır. Bu tehlikeli bir hatadır, çünkü derleyici hatalı bir şey olduğunu fark edemez. Programınız da gerektiği gibi çalışmaz (tabii, çok şanslı değilseniz.)

C++ true deyimleri belirtmek için 1 değerini üretirken, 0'dan farklı her değerin (mesela, -7 veya 44) true olduğunu varsayar; sadece 0 false değerindedir. Bu nedenle, listedeki en son deyim true değerindedir.

Şimdi bu operatörlerin tipik durumlarda nasıl kullanıldığını görelim. Öncelikle döngüleri, ardından kararları inceleyeceğiz.

Döngüler

Döngüler programınızın bir bölümünün belli sayıda tekrar edilmesini sağlar. Tekrarlar bir koşul doğrulanana kadar sürer. Koşul yanlış olursa, döngü biter ve kontrol döngüyü takip eden ifadeye geçer.

C++'ta üç tür döngü çeşidi vardır: **for** döngüsü, **while** döngüsü ve **do** döngüsü.

for Döngüsü

for döngüsü (pek çok kişi için) anlaşılması en kolay C++ döngüsüdür. **for** döngüsünün tüm döngü kontrol öğeleri tek bir yerde toplanmıştır. Diğer döngü yapılarında döngü kontrol öğeleri program içinde dağınıktır; bu durum, döngünün nasıl çalıştığını anlamayı zorlaştırabilir.

for döngüsü, kodun bir bölümünü sabit sayıda çalıştırır. Bu, genellikle (her zaman değilse de) döngüye girmeden önce kodun kaç kere çalıştırılacağını bildiğiniz durumlarda kullanılır.

FORDEMO örneği 0 ile 14 arasındaki sayıların karelerini görüntülüyor:

```
// fordemo.cpp
// basit bir FOR dongusu
#include <iostream>
using namespace std;

int main()
   {
   int j;                    //bir dongu degiskeni tanimla

   for(j=0; j<15; j++)       //0'dan 14'e kadar tekrarlanan dongu
   cout << j * j << " ";     //j'nin karesi goruntulenir
   cout << endl;
   return 0;
   }
```

Programın çıktısı şöyledir:

```
0  1  4  9  16  25  36  49  64  81  100  121  144  169  196
```

Bu program nasıl çalışır? **for** ifadesi döngüyü kontrol eder. **for** ifadesi, **for** anahtar kelimesi ile başlar. İçinde noktalı virgül ile ayrılmış üç deyimin yer aldığı parantezler **for** anahtar kelimesinin ardından gelir:

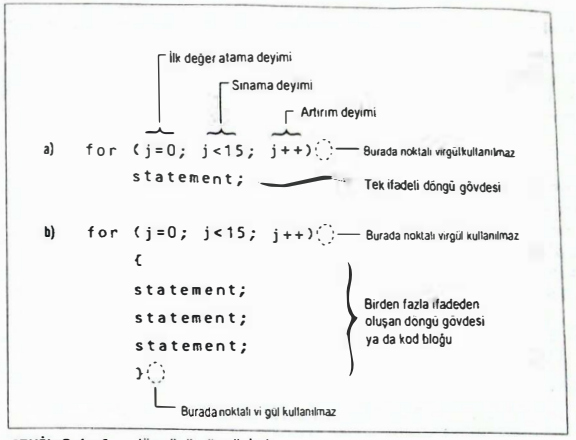
```
for(j=0; j<15; j++)
```

Bu üç deyim, Şekil 3.1'de görüldüğü gibi, *ilk değer atama deyimi*, *koşul sınama deyimi* ve *artırım deyimidir*.

Bu üç deyim genellikle (her zaman olmayabilir) aynı değişkeni içerir. Bu değişkene *döngü değişkeni* denir. **FORDEMO** örneğinde döngü değişkeni **j**'dir. Döngü değişkeni, döngü gövdesindeki ifadeler çalıştırılmaya başlamadan tanımlanır.

Döngü *gövdesi*, döngü sırasında çalıştırılması gereken program parçasıdır. Bu program parçasının tekrarlanması döngünün amacıdır. Bu örnekte döngü gövdesi tek ifadeden ibarettir:

```
cout << j * j << "  ";
```

ŞEKİL 3.1: **for** döngüsü söz dizimi.

Bu ifade j'nin karesini görüntüler. Her sayıyı çift boşluk takip eder. Sayıların karesi, j kendisi ile çarpılarak bulunur. Döngü çalıştıkça, j; 0, 1, 2, 3 dizisi boyunca 14'e kadar gider. Böylece, bu sayıların kareleri görüntülenir: 0, 1, 4, 9'dan 196'ya kadar.

for ifadesinin sonunda noktalı virgül olmadığına dikkat edin. Bunun sebebi, **for** ifadesi ve döngü gövdesinin birlikte bir program ifadesi oluşturmalarıdır. Bu, önemli bir ayrıntıdır. **for** ifadesinden sonra noktalı virgül koyarsanız, derleyici döngü gövdesinin olmadığını düşünecektir. Programınız muhtemelen hiç ummadığınız şekilde çalışır.

for ifadesindeki bu üç deyimin döngüyü nasıl kontrol ettiğine bir bakalım.

İlk Değer Atama Deyimi

Bu deyim sadece bir kez, döngü ilk başladığında çalıştırılır ve döngü değişkenine ilk değerini atar. **FORDEMO** örneğinde bu deyim, **j**'ye 0 değerini verir.

Koşul Sınama Deyimi

Bu deyim genellikle bir ilişkisel operatör içerir. Döngü süresince her seferinde, döngü gövdesi çalıştırılmadan hemen önce, bu deyimin değeri hesaplanır. Bu deyim, döngünün yeniden tekrarlanıp tekrarlanmayacağını belirler. Eğer deyimin değeri true ise, döngü bir kez daha çalıştırılır. Eğer false ise, döngü biter ve kontrol, döngüyü takip eden ifadeye geçer. **FORDEMO** örneğinde,

```
cout << endl;
```

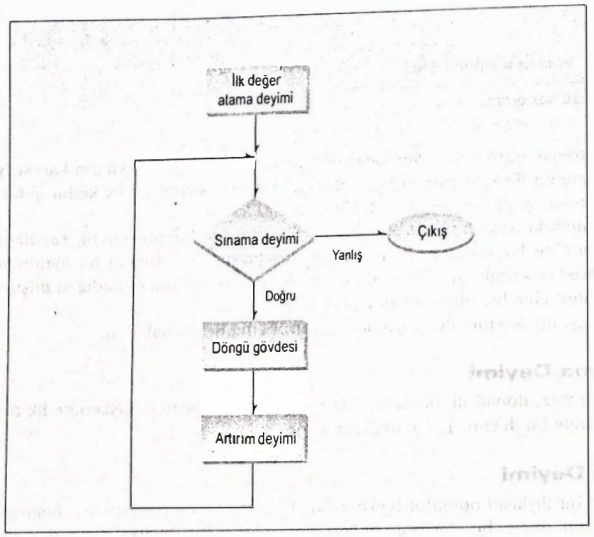
ifadesi, döngünün tamamlanmasından sonra çalıştırılır.

Artırım Deyimi

Artırım deyimi, döngü değişkeninin değerini, genellikle bir arttırarak değiştirir. Bu deyim her zaman döngünün sonunda, döngü gövdesi çalıştırıldıktan sonra gerçeklenir. Bu örnekte, artırım operatörü ++, döngü süresince her seferinde j'ye 1 ekler. Şekil 3.2, for döngüsünün çalışmasını gösteren bir akış şeması içerir.

Kaç Kere?

FORDEMO örneğindeki döngü tam olarak 15 kez çalıştırılır. İlk seferinde j, 0 değerindedir. Bu, ilk değer atama deyiminde gerçeklenir. Döngü son kez tekrarlandığında j, 14 olur. Bu, koşulun sınandığı j<15 deyimi ile belirlenir. j, 15 olduğunda döngü biter; j bu değeri aldığında döngü gövdesi çalıştırılmaz. Burada gösterilen düzenleme, genellikle, bir işin sabit sayıda tekrarlandığı durumlarda kullanılır: 0'dan başla; istenilen iterasyon sayısına eşit bir değer ve eşittir işaretinden oluşan bir koşul sınama deyimi kullan ve döngü değişkenini her iterasyondan sonra bir kez arttır.



ŞEKİL 3.2: for döngüsünün çalışma mantığı.

Bu da bir başka **for** döngüsü örneği:

```
for (count=0; count<100; count++)
// dongu govdesi
```

Bu örnekteki döngü gövdesi kaç kere tekrarlanacak? **count**, 0'dan 99'a kadar ilerleyecek, döngü tam olarak 100 kez tekrarlanacaktır.

Birden Fazla İfade İçeren Döngü Gövdeleri

Elbette, döngü gövdesi içinde birden fazla ifade çalıştırmak isteyebilirsiniz. Döngü gövdesi içindeki ifadeler, tıpkı fonksiyonlardaki gibi, küme parantezleriyle sınırlanır. Döngü gövdesi içindeki ifadelerden sonra noktalı virgül konuluyor olsa da, döngü gövdesinin son parantezinden sonra noktalı virgül konmaz.

Bir sonraki örnek CUBELIST, döngü gövdesi içinde üç tane ifade kullanır. Program, 1 ile 10 arasındaki sayıların küplerini çift sütun biçiminde basar.

```
// cubelist.cpp
// 1'den 10'a kadar olan sayilarin kuplerini listeler
#include <iostream>
#include <iomanip>                          //setw icin
using namespace std;

int main()
   {
   int numb;                                //dongu degiskenini tanımla

   for(numb=1; numb<=10; numb++)            //1'den 10'a kadar tekrarla
      {
      cout << setw(4) << numb;              //1. Sutunu goruntule
      int cube = numb*numb*numb;            //kupleri hesapla
      cout << setw(6) << cube << endl;      //2. Sutunu goruntule
      }
   return 0;
   }
```

Programın çıktısı şöyledir:

```
1     1
2     8
3    27
4    64
5   125
6   216
7   343
8   512
9   729
10 1000
```

Önceki programda kullanılan biçimin değiştirilemez olmadığını vurgulamak için bu programda bir değişiklik yaptık. Döngü değişkenine 0 yerine 1 ilk değer olarak verilir ve döngü değişkeni en son 9 yerine 10 değerini alıyor. Bu, <= (küçüktür veya eşittir) operatöründen kaynaklanıyor. Döngü gövdesi yine 10 kez çalıştırılıyor; ancak, bu kez döngü değişkeni 0'dan 9'a kadar değil, 1'den 10'a kadar ilerliyor.

Önceki programdaki tek satırlık döngü gövdesinin etrafına da küme parantezi koyabileceğinize dikkatinizi çekmek isteriz. Bu şart değildir. Ancak bir çok programcı, döngü gövdesi tek ifade olsun veya olmasın, küme parantezi kullanmanın programın okunabilirliğini artırdığını düşünür.

Bloklar ve Değişkenlerin Erişilebilirliği

Birkaç ifadenin küme parantezleriyle sınırlandığı döngü gövdelerine program *bloğu* denir. Blok içinde tanımlı bir değişkenin blok dışından erişilemez olması blokların önemli bir özelliğidir. (Bu konuyu "Fonksiyonlar" adlı Bölüm 5'te daha ayrıntılı ele alacağız.)

CUBELIST programında, **cube** değişkeni blok içinde tanımlanır:

```
int cube= numb*numb*numb;
```

Bu değişkene blok dışından erişemezsiniz; bu değişken sadece küme parantezleri içindeyken görülebilir. Yani,

```
cube = 10;
```

ifadesini döngü gövdesinden sonra yazarsanız, **cube** değişkeni döngü gövdesi dışında tanımsız olacağından derleyici hata mesajı verir.

Değişkenlerin erişilebilirliğini kısıtlamanın sağladığı bir avantaj, aynı program içindeki farklı bloklarda aynı değişken isimlerinin kullanılabilir olmasıdır. (**CUBELIST**'teki gibi, değişkenleri blok içinde tanımlamak C++'ta çok yaygındır; C'de ise popüler değildir.)

Girintileme ve Döngü Stili

İyi bir programlama stili, döngü gövdesinin, döngü ifadesine (ve programın kalanına) göre girintili, yani sağa doğru kaydırılarak yazılmasını gerektirir. **FORDEMO** örneğinde bir satır girintili yazılmıştır; **CUBELIST** de ise bloğun tamamı, küme parantezleriyle birlikte, girintili yazılmıştır. Girintileme (indentation), programcılara görsel açıdan önemli ölçüde yardımcı olur: Döngü gövdesinin nerede başladığını ve nerede bittiğini görmeyi kolaylaştırır. Derleyici, programın girintili olup olmamasına aldırmaz.

Bu kitaptaki döngülerde kullandığımız ortak bir stil var. Parantezleri düşey olarak aynı hizaya getiriyoruz. Ama bazı programcılar bunun yerine parantezleri döngü ifadesinden hemen sonra kullanmayı tercih eder. Örneğin şöyle:

```
for(numb=1; numb<=10; numb++){
   cout << setw(4) << numb;
   int cube = numb*numb*numb;
   cout << setw(6) << cube << endl;
   }
```

Bu tarz sayesinde kaynak kodunda bir satırlık yer tasarrufu sağlamak mümkün olur. Ancak, açılış parantezi ile kapanış parantezini birbiriyle eşlemek zorlaştığından kaynak kodu daha az okunaklı bir hal alır. Bir başka stile göre de döngünün ana bloğu girintili yazılır, parantezler ise girintisiz kullanılır:

```
for(numb=1; numb<=10; numb++)
{
   cout << setw(4) << numb;
   int cube = numb*numb*numb;
   cout << setw(6) << cube << endl;
}
```


Bu da yaygın bir yazım tarzıdır ama en azından bazı kişiler açısından, parantezlerle döngü bloğu arasındaki irtibatı azalttığı için göze zor gelir. Yine de hemen her türlü farklı yazım tarzına alışmak mümkündür. Önemli olan, hangi tarzı seçerseniz seçin, kendi tarzınızı tutarlı olarak kullanmaktır.

Çalışma Sırasında Hata Ayıklama

Döngü işleminin çalışmasını adım adım izlemek için derleyicinizin hata ayıklama (debug) özelliklerini kullandığınız bir senaryo oluşturabilirsiniz. Burada ihtiyacınız olan temel özellik, kaynak kodunu satır satır çalıştırmak anlamına gelen tek tek adımlama (*single-stepping*) özelliğidir. Derleyiciniz sayesinde bu özelliği kolayca kullanabilirsiniz. Bu amaçla, önce hata ayıklaması yapacağınız program için bir proje dosyası ve kaynak kodunun yer aldığı bir pencere açın. Hata ayıklayıcısını çalıştırmak için yapmanız gereken işlemlerin tam niteliği derleyiciden derleyiciye değişir. Bunun için Ek C "Microsoft C++"a ya da Ek D "Borland C++ Builder"a bakınız. Hata ayıklayıcısının içindeyken belli bir fonksiyon tuşuna basarak kaynak kodunuzu adım adım çalıştırabilirsiniz. Böylece program ilerlerken hangi adımlardan geçildiğini tek tek görebilirsiniz. Bir döngünün içindeyken, döngü bloğundaki ifadelerin adım adım çalıştırıldığını, son ifadeye gelindiğinde (döngünün tamamlanma koşulu henüz sağlanmadıysa) yeniden döngü bloğunun ilk ifadesine geri dönüldüğünü görürsünüz.

Hata ayıklayıcısını bu şekilde adımlatarak, programın her adımında değişkenlerinizin değerlerinin nasıl değiştiğini de gözleyebilirsiniz. Hata ayıklama sürecinde bu çok önemli bir imkandır. **CUBELIST** programında bu şekilde hata ayıklaması yaparken **numb** ve **cube** değişkenlerinizi izleme penceresine *(watch window)* koyduğunuzda, bu değişkenlerin değerlerinin program ilerledikçe nasıl değiştiğini takip edebilirsiniz. Hatırlatalım; bu işlemi kendi derleyicinizle nasıl yapabileceğinizi tam olarak öğrenmek için kitabın sonundaki Ek bölümlerine başvurabilirsiniz.

Programı adım adım işletmek ve Watch penceresi bize son derece güçlü hata ayıklama imkanları sunarlar. Programınız istediğiniz şekilde çalışmıyorsa bu özellikleri kullanarak, program adım adım ilerlerken kilit önemdeki değişkenlerinizin hangi değerleri aldığını takip edebilirsiniz. Bu şekilde yaklaştığınızda genellikle problemin neden kaynaklandığını bulabilirsiniz.

Farklı for Döngü Türleri

Artırım deyimi döngü değişkenini bir adım ileri artırmak zorunda değildir; döngü değişkeni üzerinde istediğiniz işlemi burada yapabilirsiniz. Sıradaki örneğimizde değişken artırılmıyor, *eksiltilir*. **FACTOR** adını verdiğimiz bu programda önce kullanıcının bir sayı girmesi istenir. Daha sonra da bu sayının faktöriyeli hesaplanır. (Bir sayının faktöriyeli, o sayı ile o sayıdan küçük tüm pozitif tam sayıların çarpımıdır. Örneğin 5'in faktöriyeli 5*4*3*2*1 yani 120'dir.)

```
// factor.cpp
// faktoriyel hesabi ve FOR dongusune ornek
#include <iostream>
using namespace std;

int main()
   {
   unsigned int numb;
   unsigned long fact=1;          //buyuk sayilar icin long kullan

   cout << "Enter a number: ";
```

```
    cin >> numb;                    //sayiyi al
    for(int j=numb; j>0; j--)       //1'i
    fact *= j;                      //numb, numb-1, ..., 2, 1 ile carp
    cout << "Factorial is " << fact << endl;
    return 0;
    }
```

Bu örnekte, başlangıçta `j`'nin değeri kullanıcının girdiği sayıya eşitlenmektedir. Koşul sınama deyimine bakarsak, döngünün `j > 0` olduğu sürece çalışacağını görürüz. Artırım deyiminde ise `j` her adımda bir azaltılmaktadır.

Faktöriyel için `unsigned long` tipi kullandık. Çünkü, çok küçük sayıların faktöriyelleri bile çok büyük olabilir. Windows gibi 32 bit sistemlerde `int` ile `long` aynı şeydir. Ama 16 bitlik sistemlerde `long` daha fazla kapasite sağlar. Aşağıdaki program çıktısı, girilen çok küçük bir sayı için bile faktöriyelin ne kadar büyük olabileceğini gösteriyor.

```
Enter a number: 10
Factorial is 3628800
```

Programımıza girebileceğiniz en büyük sayı 12'dir. Daha büyük bir sayı girerseniz, hata mesajı almazsınız. Ama `long` tipinin kapasitesi aşılmış olacağı için alacağınız sonuç yanlış olacaktır.

for İfadelerinde Tanımlanan Değişkenler

Yukarıdaki örnekte dikkat edilmesi gereken bir nokta daha var: `j` değişkeni döngünün içinde tanımlanmıştır:

```
for(int j=numb; j>0; j--)
```

Bu, C++'ta çok yaygın bir yapıdır. Çoğu durumda da döngü değişkenlerinin en uygun kullanım şeklidir. Çünkü bu şekilde değişken, kaynak kodu içinde kullanıldığı yere mümkün olan en yakın noktada tanımlanır. Döngü içinde tanımlanan değişkenler sadece döngü bloğu içinde görünür durumdadırlar. (Microsoft'un derleyicisi, döngü değişkenlerini tanımlandığı noktadan dosya sonuna kadar görünür kılar, ama bu Standard C++'a ait bir özellik değildir.)

Birden Fazla İlk Değer Atama ve Sınama Deyimleri

`for` ifadesinde, birden fazla değişkene ilk değer ataması yapabilirsiniz. Bunun için değişkenleri virgülle ayırmanız yeterlidir. Artırım deyimi de birden fazla olabilir. Ancak koşul sınama deyiminin tek olması gereklidir. Örnek:

```
for(j=0, alpha=100; j<50; j++, beta--)
   {
   // dongu blogu
   }
```

Bu örnekte `j` adlı normal bir döngü değişkeni olmakla birlikte, ayrıca `alpha` adlı bir başka değişkene de başlangıç değeri atanmakta, `beta` adlı üçüncü bir değişkenin de değeri bir azaltılmaktadır. `alpha` ve `beta` değişkenlerinin birbirleriyle ya da `j` ile herhangi bir ilintileri olması gerekmez. Birden fazla değişkene ilk değeri verildiği zaman ya da birden fazla değişkene artırım uygulandığı zaman, değişkenler virgülle birbirinden ayrılmalıdır.

Aslında, `for` ifadesindeki değişkenlerin bazılarını, hatta tümünü kaldırabilirsiniz. Örneğin şu ifade

```
for(;;)
```

koşul sınama deyimi `true` olan bir `while` döngüsü ile aynı işleve sahiptir. `while` döngüleri ile ilgili ayrıntılara aşağıda bakacağız.

Biz bu tür kısaltmalardan kaçınıyoruz. Böyle yaklaşımlar, kaynak kodunu kısa ve öz bir şekle sokabilir ama okunurluğu ve anlaşılırlığı da azaltır. Aynı işi yapacak bağımsız ifadeler ya da farklı bir döngü yapısı bulmak her zaman mümkündür.

while Döngüsü

`for` döngüsü bir işi belli bir sayıda tekrarlamaya yarar. Peki bir işi kaç kere tekrarlayacağımızı döngüye girene kadar bilmiyorsak ne yapacağız? İşte böyle durumlarda başka bir döngü türü olan `while` döngüsü kullanılabilir.

Sıradaki örneğimiz olan `ENDON0`, kullanıcının bir dizi sayı girmesini ister. Girilen sayı 0 ise döngü durur. Dikkat edersenız, programın 0 girilene kadar kaç tane sayı geleceğini bilme imkanı yok. Çünkü bunu kullanıcı belirler.

```
// endon0.cpp
// WHILE dongusu
#include <iostream>
using namespace std;

int main()
   {
   int n = 99;          //n'nin ilk degerinin 0 olmadigindan emin ol

   while(n != 0)        //n, 0 olana kadar tekrar et
   cin >> n;            //n'e bir sayi oku
   cout << endl;
   return 0;
   }
```

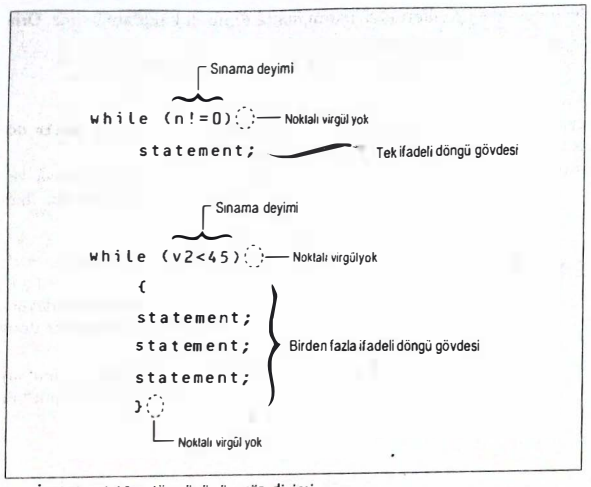
Programın örnek bir çıktısı da aşağıdadır. Kullanıcı bir dizi sayı girmiş ve 0 girildiğinde hem döngü hem de program sona ermiştir.

```
1
27
33
144
9
0
```

`while` döngüsü `for` döngüsünün sadeleştirilmiş bir türüne benzer. Koşul sınama deyimi vardır ama ilk değer atama ya da artırım deyimlerine sahip değildir. Şekil 3.3'te `while` döngüsünün söz dizimi izah edilmiştir.

Koşul sınama deyimi `true` (doğru) oldukça döngü çalışmaya devam eder. `ENDON0` örneğinde, koşul sınama deyimi

```
n != 0
```



ŞEKİL 3.3: `while` döngüsünün söz dizimi.

(yani `n` eşit değildir 0), kullanıcı 0 girene kadar doğru olmaya devam eder.

Şekil 3.4'te `while` döngüsünün işleyişi anlatılmaktadır. `while` döngüsünün basitliği biraz yanıltıcıdır. İlk değer atama ifadesi olmasa da döngü değişkenine (`ENDONO` örneğinde `n`) döngüye girilmeden önce mutlaka bir ilk değer atanmalıdır. Döngü değişkeninin değerini değiştiren bir ifade de mutlaka döngü bloğu içinde yer almalıdır. Aksi halde döngü bir türlü bitmez. Bu ifade `ENDONO` örneğinde şudur:

```
cin>>n;
```

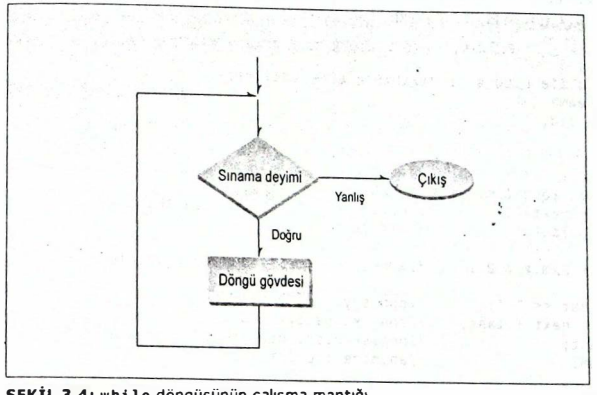
while Döngüsünde Birden Fazla İfade Kullanımı

Sıradaki örneğimiz olan `WHILE4`'da `while` döngüsü içinde birden fazla ifade kullanılmaktadır. Bu örnek, daha önce `for` döngüsünde gördüğümüz `CUBELIST` programının farklı bir şeklinden oluşmaktadır. Ancak bu sefer bir dizi tam sayının küpünü değil de dördüncü kuvvetini hesaplayacağız. Gelin, bu programda sonuçları dört basamaklı bir sütuna sığdırmanın önemli olduğunu da varsayalım. Sonuçların bu genişliğin dışına taşmamasını sağlamak için sonucumuz 9999'u geçmeden önce döngüyü durdurmamız gerekir. Önceden hesap yapmaksızın hangi sayının bu ve bundan büyük sonuç vereceğini bilemeyiz. Bırakalım bunu program halletsin. Böylece, koşul sınama deyimimizi de oluşturmuş oluyoruz: İlgilendiğimiz sayının dördüncü kuvveti 9999'u geçtiğinde döngüden çıkacağız, demektir.

```
// while4.cpp
// sayilarin dorduncu kuvvetini gosterir
#include <iostream>
#include <iomanip>                    //setw icin
using namespace std;
```



```
int main()
   {
   int pow=1;                         //us, baslangicta 1
   int numb=1;                        //numb, 1'den ???'ne kadar gidecek

   while( pow<10000 )                 //us 4 haneyi asmadikca donguye devam et
      {
      cout << setw(2) << numb;        //sayiyi goster
      cout << setw(5) << pow << endl; //sayinin dorduncu kuvvetini goster
      ++numb;                         //bir sonraki usse hazirlik yap
      pow = numb*numb*numb*numb;      //dorduncu kuvveti hesapla
      }
   cout << endl;
   return 0;
   }
```



ŞEKİL 3.4: `while` döngüsünün çalışma mantığı.

`numb`'un dördüncü kuvvetini hesaplamak için sayıyı kendisi ile üç kere çarpmak yeterlidir. Döngüyü her geçişimizde `numb`'ı bir artırıyoruz. Ancak `while` ifadesinin koşul sınama deyiminde `numb`'ı kullanmıyoruz. Döngünün ne zaman sona ereceğine `pow`'un değerine bakarak karar veriyoruz. Programın çıktısı şöyledir:

```
1      1
2     16
3     81
4    256
5    625
6   1296
7   2401
8   4096
9   6561
```

Bir sonraki adımda (10'un dördüncü kuvvetinde) sonuç 10,000 olacaktı. Bu da dört haneli sonuç sütunumuza sığmayacaktı.

Öncelik Sırası: Aritmetik ve İlişkisel Operatörler

Sıradaki programımızda operatörlerin öncelik sırasıyla ilgileneceğiz. Bu program *Fibonacci* adıyla ünlü sayı dizisini üretmeye yarar. İşte, söz konusu dizinin ilk sayıları:

```
1 1 2 3 5 8 13 21 34 55
```

Her sayı, kendinden önceki iki sayının toplamını bularak oluşturulur: 1+1=2, 1+2=3, 2+3=5, 3+5=8, 5+8=13, 8+13=21, 13+21=34, 21+34=55 diye devam eden bir dizi. Fibonacci dizisi bilgisayarlarda sıralama yapmaktan ayçiçeği bitkisindeki spirallerin sayısına kadar hayret edilecek kadar çok farklı alanlarda uygulama bulmaktadır.

Fibonacci dizisinin en ilginç özelliklerinden biri bu dizi ile "altın oran" arasındaki ilişkidir. Altın oranın mimari ve sanattaki ideal oran olduğu varsayılır. Eski Yunan tapınaklarında bu oran kullanılmıştır. Fibonacci dizisi ilerledikçe dizinin son iki sayısı arasındaki oran altın orana yaklaşır. **FIBO.CPP**'nin kaynak kodu şöyledir:

```
// fibo.cpp
// WHILE dongusu ile Fibonacci dizisinin elde edilmesi
#include <iostream>
using namespace std;

int main()
   {                              //en buyuk unsigned long
   const unsigned long limit = 4294967295;
   unsigned long next=0;          //dizideki son sayidan sonraki sayi
   unsigned long last=1;          //dizideki son sayi

   while( next < limit / 2 )      //sonuclarin asiri buyumemesi icin
      {
      cout << last << " ";        //son sayi
      long sum = next + last;     //son iki sayiyi topla
      next = last;                //degiskenleri dizinin bir sonraki
      last = sum;                 //adimina ilerlet
      }
   cout << endl;
   return 0;
. }
```

Programın çıktısı ise şu şekildedir:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
1597 2584 4181 6765 10946 17711 28657 46368 75025 121393
196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986
102334155 165580141 267914296 433494437 701408733 1134903170
1836311903 2971215073
```

Tapınak inşa etmeye meraklı olanlar için; yukarıdaki son iki rakamın oranının altın orana çok yakın bir rakam olduğunu söyleyelim: 0.618033988.

FIBO programımızda **unsigned long** tipini kullandık. Bu tip, en büyük pozitif tamsayıları tutmak için gereklidir. **while** ifadesindeki koşul sınama deyimi, sayılarımızın bu tipin sınırları dışına taşacak kadar büyümesini engeller, oraya gelmeden döngüden çıkar. Bu limit, **const** tipinde tanımlanır, çünkü bu program içinde değişmez. **next** değişkenimiz bu limitin yarısına ulaştığında durmamız gerekir, aksi halde iki değişkenin toplamı limiti aşar.

Koşul sınama deyiminde iki operatör vardır:

```
(next < limit / 2)
```

Amacımız **next**'i **limit/2** ile kıyaslamaktır. Bir başka ifade ile, önce bölme işleminin sonra kıyaslama işleminin gerçekleşmesini istiyoruz. Operatörlerin bu şekilde işlemesini garanti altına almak için parantezlerden yararlanabiliriz.

```
(next < (limit/2) )
```

Ama aslında parantezlere ihtiyacımız da yoktur. Neden? Çünkü aritmetik operatörler (işlemler), ilişkisel operatörlerden daha önceliklidir. Bu nedenle **limit/2**'nin kıyaslamadan daha önce hesaplanacağı garanti altındadır. Operatörlerin önceliği konusunu bu bölümün daha sonraki sayfalarında mantıksal operatörleri işlerken özetleyeceğiz.

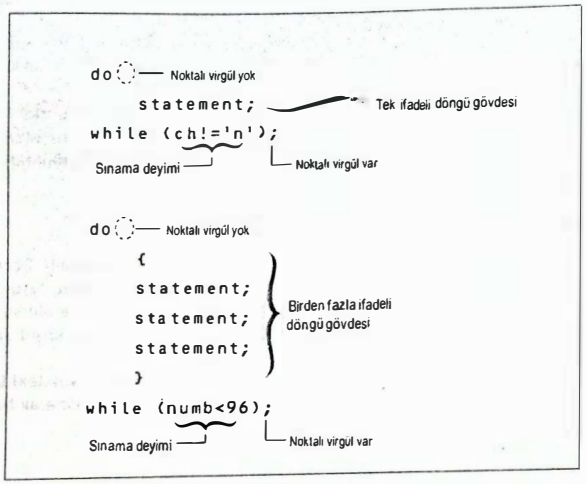
do Döngüsü

while döngüsünde koşul sınama deyiminin değeri döngünün *en başında* hesaplanır. Eğer döngüye girerken bu deyimin değeri yanlış ise, döngü gövdesi hiç çalıştırılmaz. Bazı durumlarda istenilen bu olabilir. Ancak bazı durumlarda koşul sınama deyiminin ilk değeri ne olursa olsun döngü gövdesinin en az bir kez çalıştırılmasını isteyebilirsiniz. İstenilen buysa, koşul sınama deyiminin döngünün *sonunda* yer aldığı **do** döngüsünü kullanmalısınız.

DIVDO örneğimiz kullanıcıyı iki sayı girmeye davet eder: Bölünen (bölme kesrindeki üstteki sayı) ve bölen (bölme kesrindeki alttaki sayı). Program / ve % operatörlerini kullanarak bölümü (yanıtı) ve kalanı hesaplar; sonucu ekrana yazar.

```
// divdo.cpp
// DO dongusu
#include <iostream>
using namespace std;

int main()
   {
   long dividend, divisor;
   char ch;

   do                                        //do dongusunun baslangici
      {                                      //bazi islemler yap
      cout << "Enter dividend: "; cin >> dividend;
      cout << "Enter divisor: "; cin >> divisor;
      cout << "Quotient is " << dividend / divisor;
      cout << ", remainder is " << dividend % divisor;
      cout << "\nDo another?(y/n): ";       //yeniden yapilacak mi?
      cin >> ch;
      }
   while(ch != 'n' ));                       //dongu kosulu
   return 0;
   }
```

Bu programın büyük kısmı `do` döngüsü içinde yer alır. Öncelikle, `do` anahtar kelimesi döngünün başladığı yere işaret eder. Ardından, diğer döngülerde olduğu gibi, küme parantezleri döngü gövdesini sınırlandırır. Son olarak, `while` ifadesi koşul sınama deyimini içerir ve döngüyü sonlandırır. Bu `while` ifadesi, `while` döngüsündeki ile aynı görünür; yalnız, bu ifade döngünün sonunda yer alır ve noktalı virgül ile sona erer (bu, unutulması kolay bir ayrıntıdır!) `do` döngüsünün söz dizimi Şekil 3.5'te gösterilmiştir.



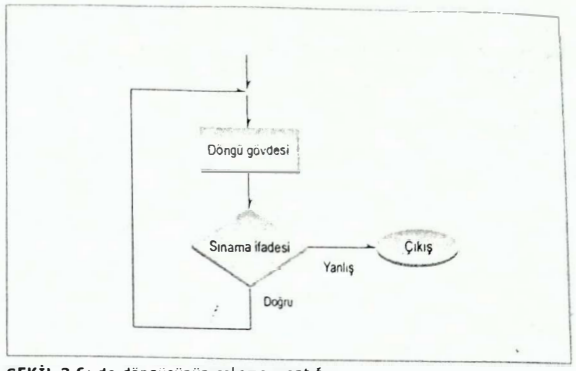
ŞEKİL 3.5: `do` döngüsünün söz dizimi.

Her hesaplamanın ardından `DIVDO` kullanıcıya devam edip etmeyeceğini sorar. Kullanıcı eğer devam etmek isterse, 'y' karakteri girer ve koşul sınama deyimi

```
ch != 'n';
```

true değerini korur. Eğer kullanıcı 'n' girerse, bu deyim false olur ve döngü sonlanır. Şekil 3.6 `do` döngüsünün çalışma mantığını şematik olarak gösterir. `DIVDO`'nun örnek çıktısı şöyledir:

```
Enter dividend: 11
Enter divisor: 3
Quotient is 3, remainder is 2
Do another? (y/n): y
Enter dividend: 222
Enter divisor: 17
Quotient is 13, remainder is 1
Do another?(y/n): n
```

ŞEKİL 3.6: `do` döngüsünün çalışma mantığı.

Hangi Döngüyü Ne Zaman Kullanmalı

Döngülerin nasıl kullanılacağı hakkında genel açıklamalar yapmıştık. `for` döngüsü, döngünün kaç kez çalışacağını önceden bildiğimiz durumlarda uygundur. Döngünün ne zaman duracağını önceden bilmediğimiz durumlarda ise `while` ve `do` döngüleri kullanılır (`while` döngüsü, döngü şartı sağlanmadığı takdirde döngü gövdesinin bir kere bile çalışmasını istemediğiniz koşullarda; `do` döngüsü ise en az bir kez çalışmasından emin olduğunuz koşullarda kullanılır.)

Bu kriterler oldukça keyfidir. Hangi döngü tipini kullanmanız gerektiği sıkı bir kuraldan çok bir stil meselesidir. Aslında hemen hemen her durumda bu döngü tiplerinin her birini çalıştırabilirsiniz. Programınızın takibini en çok kolaylaştıran ve netleştiren tipi tercih etmelisiniz.

Kararlar

Döngülerdeki kararlar her zaman aynı soruya yöneliktir: Bunu (döngü gövdesini) tekrarlamalı mıyız? Biz insanlar, karar verme işlemlerimizde kısıtlanmış olmayı sıkıcı buluruz. Bir yandan bugün işe gidip gitmemeyi düşünürken (döngüye devam ediyoruz), diğer yandan kırmızı veya yeşil renkli bir gömlek alıp almamayı (ya da hiç gömlek almamayı), tatile çıkıp çıkmamayı, eğer çıkacaksak dağa mı, denize mi gitmeyi düşünüp, karar vermek zorundayız.

Programlar, ayrıca, bu bir kerelik kararları da almak zorundadır. Program içindeki bir karar, deyimin değerine bağlı olarak, programın farklı bir bölümüne bir kerelik bir atlayışa neden olur.

C++'ta kararlar birkaç şekilde yapılabilir. Bunların en önemlisi, iki alternatif arasında seçim yapan `if...else` ifadesidir. Bu ifade `else` kısmı olmadan, basit bir `if` ifadesi olarak da kullanılabilir. Diğer karar ifadesi olan `switch`, tek bir değişkenin değerine bağlı olarak, birden fazla alternatif program parçası için seçenekler sunar. Son olarak, özel durumlarda koşul operatörü kullanılır. Bu yapıların tümünü inceleyeceğiz.

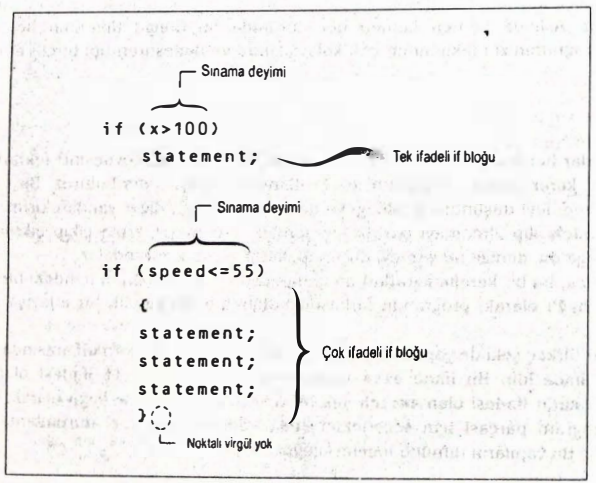
if İfadesi

if ifadesi, karar ifadelerinin en basit olanıdır. Bir sonraki programımız olan **IFDEMO**, konuyla ilgili bir örnek verir:

```
// ifdemo.cpp
// IF ifadesi
#include <iostream>
using namespace std;

int main()
   {
   int x;

   cout << "Enter a number:";
   cin >> x;
   if(x > 100)
      cout << "That number is greater than 100\n";
   return 0;
   }
```

if anahtar kelimesini parantez içinde bir koşul sınama deyimi takip eder. **if** ifadesinin söz dizimi Şekil 3.7'de gösterilmiştir. Gördüğünüz gibi, **if**'in sözdizimi **while**'ınkine çok benzer. İkisi arasındaki fark, koşul sınama deyimi doğru ise, **if**'i takip eden ifadelerin bir kez çalıştırılması; **while**'ı takip eden ifadelerin ise sınama deyimi false olana kadar tekrar tekrar çalıştırılmasıdır. Şekil 3.8 **if** ifadesinin çalışma mantığını gösterir.

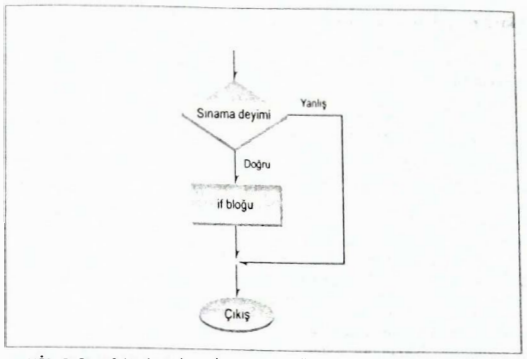
Kullanıcı tarafından 100'den büyük bir değer girildiği zaman **IFDEMO** programının çıktısı şöyledir:



ŞEKİL 3.7: **if** ifadesinin söz dizimi.


```
Enter a number: 2000
That number is greater than 100
```

Eğer girilen sayı 100'den büyük değilse, program ikinci satırı yazmadan sona erer.



ŞEKİL 3.8: **if** ifadesinin çalışma mantığı.

if İfadesi İçinde Birden Fazla İfade Kullanımı

Döngülerde olduğu gibi, **if** gövdesi içindeki kod tek ifadeden oluşabilir – **IFDEMO** örneğinde olduğu gibi – ya da küme parantezleriyle sınırlanmış ifadeler bloğundan oluşabilir. **IF2**, bu varyasyonun **IFDEMO** üzerinde nasıl göründüğünü gösterir.

```
// if2.cpp
// birden fazla ifade iceren IF ifadesi
#include <iostream>
using namespace std;

int main()
   {
   int x;

   cout << "Enter a number: ";
   cin >> x;
   if(x > 100)
      {
      cout << "The number " << x;
      cout << " is greater than 100\n";
      }
   return 0;
   }
```

IF2'nin bir çıktısı şöyledir:

```
Enter a number: 12345
The number 12345 is greater than 100
```

Döngü İçindeki if'ler

Şu ana dek gördüğümüz döngü ve karar yapıları birbiri içine yerleştirilebilir. `if`'leri döngüler içine, döngüleri `if`'lerin içine, `if`'leri `if`'lerin içine vs yerleştirebilirsiniz. `PRIME` örneğinde `for` döngüsü içine bir `if` ifadesi yerleştirilmiştir. Bu örnek, giriş olarak girdiğiniz sayının asal sayı olup olmadığını bildirir. (Asal sayılar yalnızca kendileri ve 1 tarafından tam bölünebilen sayılardır. İlk birkaç asal sayı şunlardır: 2, 3, 5, 7, 11, 13, 17.)

```
// prime.cpp
// asal sayilarla IF ifadesi
#include <iostream>
using namespace std;
#include <process.h>              //exit() icin

int main()
   {
   unsigned long n, j;

   cout << "Enter a number: ";
   cin >> n;                      //test icin sayiyi al
   for(j=2; j <= n/2; j++)        //2'den itibaren her tamsayi ile bol
      if(n%j == 0)                //eger kalan 0 ise,
         {                        //j ile bolunebilir demektir
         cout << "It 's not prime; divisible by " << j << endl;
         exit(0);                 //programdan cikis
         }
   cout << "It 's prime \n ";
   return 0;
   }
```

Bu örnekte, kullanıcının girdiği değer `n` değişkenine atanır. Program daha sonra bir `for` döngüsü kullanarak `n`'yi 2 ile `n/2` arasındaki her sayıya böler. Bölen, döngü değişkeni, `j`'dir. Eğer `n`, `j`'nin herhangi bir değeri ile tam bölünüyorsa, `n` asal sayı değildir. Bir sayı başka bir sayıya tam bölünürse, kalan 0'dır. `if` ifadesi içinde `j`'nin her değeri için bu koşulu test etmek amacıyla kalan operatörü, `%`, kullanılır. Eğer sayı asal sayı değilse, kullanıcıya bu durum bildirilip, programdan çıkılır.

Programın üç ayrı kez çalıştırılması neticesinde elde edilen çıktılar şöyle olabilir:

```
Enter a number: 13
It's a prime
Enter a number: 22229
It's a prime
Enter a number: 22231
It's not prime; divisible by 11
```

Döngü gövdesi etrafında küme parantezi olmadığına dikkat edin. Çünkü `if` ifadesi ve onun içindeki ifadeler tek bir ifade sayılır. Derleyici burada parantez bulunmasına gerek duymaz ama siz arzu ederseniz kaynak kodunu daha okunaklı kılmak için parantez kullanabilirsiniz.

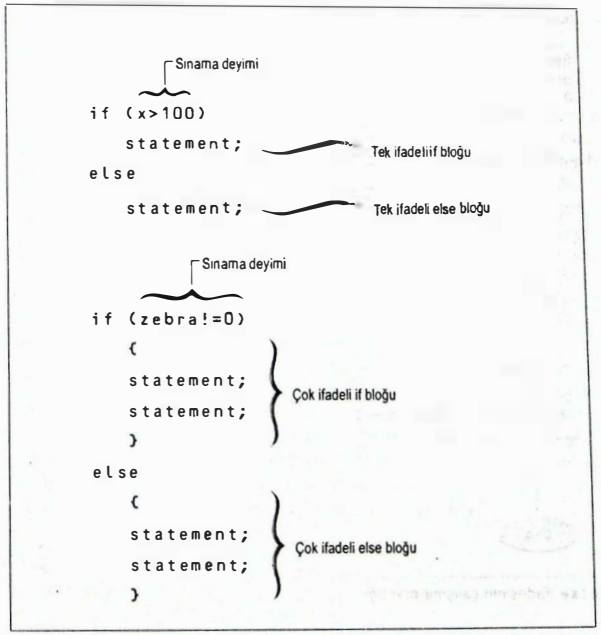
exit() Kütüphane Fonksiyonu

`PRIME`, bir sayının asal sayı olmadığını anlar anlamaz sona erer. Çünkü bir sayının asal olmadığını defalarca kanıtlamaya gerek yoktur. Bu durumda döngüden çıkmak için `exit()` adlı kütüphane fonksiyonu kullanılır. Söz konusu fonksiyon, kodun neresinde olursa olsun, programın

sona ermesine neden olur. `exit()` fonksiyonunun herhangi bir dönüş değeri yoktur. Bizim örneğimizde argüman olarak 0 kullanılmıştır. Bu değer, program sona erince işletim sistemine iletilir. (Söz konusu değer, toplu işlem (batch) dosyalarında işe yarar. Toplu işlem dosyalarında `exit()` fonksiyonunun döndürdüğü değeri okumak için `ERRORLEVEL` parametresine bakabilirsiniz. Normalde, program problemsiz bitmişse 0 döndürülür; başka dönüş değerleri problem olduğu anlamına gelir.)

if...else İfadesi

`if` ifadesi, belli bir koşulun sağlandığı (`true` olduğu) zamanlarda belli bir işi yapmanızı sağlar. Eğer ifade true değilse, hiçbir şey olmaz. Peki ya ifade true olduğunda bir şey, false olduğunda başka bir şey yapmak istersek? İşte o zaman da `if...else` ifadesi devreye girer. `if...else` ifadesi şunlardan oluşur: `if` anahtar kelimesi, bir ifade ya da birkaç ifadeden oluşan blok, `else` anahtar kelimesi, *başka* bir ifade ya da birkaç ifadeden oluşan blok. `if...else` ifadesinin sözdizimi Şekil 3.9'da gösterilmiştir.



ŞEKİL 3.9: `if...else` ifadesinin söz dizimi.

Aşağıdaki örnekte, daha önce kullandığımız `if`'li yapıya bu sefer bir de `else` ekliyoruz:

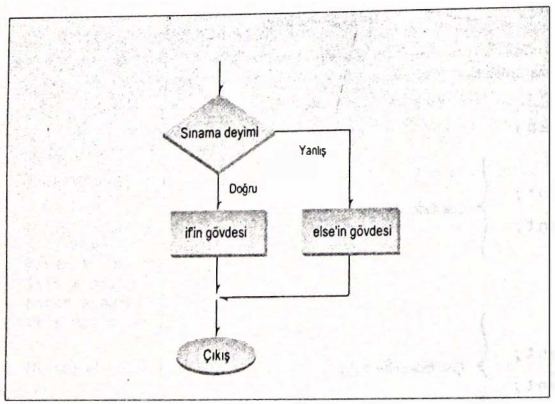
```
// ifelse.cpp
// IF...ELSE ifadesi
#include <iostream>
using namespace std;

int main()
   {
   int x;
   cout << "\nEnter a number:";
   cin >> x;
   if(x > 100)
      cout << "That number is greater than 100\n";
   else
      cout << "That number is not greater than 100\n";
   return 0;
   }
```

if ifadesindeki koşul sınama deyimi true ise program mesajlardan birini, deyim false ise diğerini verir.

Programın iki farklı çalışmasının çıktısı şöyledir:

```
Enter a number:300
That number is greater than 100
Enter a number:3
That number is not greater than 100
```

if...else ifadesinin çalışması Şekil 3.10'da açıklanmıştır.



ŞEKİL 3.10: if...else ifadesinin çalışma mantığı.

getche() Kütüphane Fonksiyonu

Sıradaki örneğimizde bir **while** döngüsü içine yerleştirilmiş bir **if...else** ifadesi vardır. Bu örnekte ayrıca **getche()** adlı yeni bir kütüphane fonksiyonu ile tanışıyoruz. **CHCOUNT** adlı bu program, kullanıcının girdiği bir ifadedeki kelime ve karakterlerin sayısını bulmaya yarar.


```
// chcount.cpp
// girilen kelime ve karakterlerin sayisini bulur
#include <iostream>
using namespace std;
#include <conio.h>      //getche() icin

int main()
   {
   int chcount=0;       //bosluk disindaki karakterlerin sayisi
   int wdcount=1;       //kelimeler arasindaki bosluklarin sayisi
   char ch = 'a';       //karakterin '\r' olmadigini garantilemek icin

   cout << "Enter a phrase: ";
   while( ch != '\r' )  //Enter girilene kadar donguye devam et
      {
      ch = getche();    //bir karakter oku
      if( ch==' ' )     //bosluksa
      wdcount++;        //kelime sayisini artir
      else              //degilse,
      chcount++;        //karakter sayisini artir
      }                 //sonuclari goster
   cout << "\nWords=" << wdcount << endl
        << "Letters=" << (chcount-1) << endl;
   return 0;
   }
```

Şu ana kadar programlarımıza giriş almak için sadece **cin** ve **>>** kullandık. Bu yaklaşım, veri girişi tamamlandığı zaman kullanıcının mutlaka Enter tuşuna basmasını gerektirir. Harfleri tek tek girerken her harfin ardından Enter'a basmak lazımdır. Ama, yukarıdaki örnekte olduğu gibi, pek çok program da daha Enter girilmeden de kullanıcının girdiği verilere ihtiyaç duyar. İşte **getche()** adlı kütüphane fonksiyonu bu işe yarar. Bu fonksiyon her karakteri, girildiği anda okur ve programa iletir. **getche()** fonksiyonunun argümanı yoktur ve bu fonksiyonu kullanmak için **CONIO.H** başlık dosyası gerekir. **CHCOUNT** programında **getche()** fonksiyonunun döndürdüğü karakterin değeri **ch** değişkenine atanır. (**getche()** fonksiyonu, okuduğu karakteri ekrana yazar. **getche** isminin sonundaki **e** harfinin esprisi budur. **getch()** adlı başka bir fonksiyon da aynı işi yapar ama okuduğu karakterleri ekrana yazmaz.)

if...else ifadesi okunan karakterin boşluk karakteri olması durumunda **wdcount** değişkeninin tuttuğu kelime sayısını artırır. Okunan karakter boşluk karakterinden *başka* hangi karakter olursa olsun bu sefer de kelime sayısı değil, **chcount** değişkeninin tuttuğu karakter sayısı artırılır. Kısacası, boşluk karakterinin dışındaki her şeyin bir karakter olduğu varsayılmıştır. (Bu programın çok basit olduğunu göz ardı etmeyin; kelimeler arasına birden fazla boşluk girerek bu programı yanıltmak mümkündür.)

Şimdi de **CHCOUNT**'un örnek bir çalışmasına bakalım:

```
For while and do
Words=4
Letters=13
```

while ifadesindeki koşul sınama deyimi **ch**'nın **'\r'** karakteri, yani Enter tuşu basıldığında klavyenin döndürdüğü karakter olup olmadığına bakar. Eğer bu karakter okunmuşsa hem döngü hem de program sona erer.

Değer Atama Deyimleri

CHCOUNT programı, koddan bir satır kazanmak ve değer atama deyimleri ile öncelik sırası hakkında önemli hususları göstermek için yeniden yazılabilir. Elde edilen yapı biraz tuhaf görünebilir. Ancak bu, C++'ta (ve C'de) yaygın olarak kullanılır.

CHCOUNT'un yeniden yazılmış versiyonu, **CHCNT2**, aşağıdadır:

```
// chcnt2.cpp
// girilen kelime ve karakterlerin sayisini bulur
#include <iostream>
using namespace std;
#include <conio.h>                      //getche()icin

int main()
   {
   int chcount=0;
   int wdcount=1;                       //iki kelime arasindaki bosluk
   char ch;

   while( (ch=getche()) != '\r' ))      //Enter girilene kadar donguye devam et
      {
      if(ch==' ')                       //bosluk karakteriyse
         wdcount++;                     //kelime sayisini artir
      else                              //degilse,
         chcount++;                     //karakter sayisini artir
      }                                 //sonuclari goster
   cout << "\nWords=" << wdcount << endl
        << "Letters=" << chcount << endl;
   return 0;
   }
```

getche() fonksiyonunun döndürdüğü değer, önceki örnekte olduğu gibi, **ch** değişkenine atanır. Ancak, bu atama deyiminin tümü, **while** ifadesindeki koşul sınama deyiminin içine yerleştirilir. Değer atama deyimi, döngünün bilip bitmeyeceğini anlamak için '\r' ile karşılaştırılır. Bu ifade düzgün çalışır, çünkü atama deyiminin tümü, atanan değeri alır. Yani, eğer **getche()** fonksiyonu, **'a'** değerini döndürüyorsa, bu durumda hem **ch** değişkeni, hem de

```
(ch= getche())
```

deyimi **'a'** değerini almış olur. Bu, sonra '\r' ile karşılaşlırılır.

Yani, alama deyimlerinin de bir değeri vardır. Bu gerçek, şu tür ifadelerde de kullanılır:

```
x = y = z = 0;
```

Bu, C++'la tamamen geçerlidir. Önce, **z** değişkeni 0 değerini alır, sonra **z = 0** deyimi 0 değerini alır ve bu değer **y** değişkenine atanır. Daha sonra **y = z = 0** deyimi aynı şekilde 0 değerini alır, bu değer bu kez **x** değişkenine atanır.

```
(ch= getche())
```

atama deyimindeki parantezler şarttır, çünkü değer alama operatörü, **=**, ilişkisel operatör olan **!=** operatöründen daha düşük önceliğe sahiptir. Parantezler olmasaydı, bu deyim şöyle değerlendirilirdi:


```
while( ch = (getche() != '\r') )    //istediğimiz gibi degil
```

Bu ifade **ch** değişkenine **true** veya **false** değeri atar (bu, istediğiniz bir şey değildir.)

CHCNT2'deki **while** ifadesi dar alanda büyük bir güç sağlar. Bu, sadece bir koşul sınama deyimi değildir (**ch** değişkeninin '\r' değerine eşit olup olmadığını kontrol eder); aynı zamanda klavyeden bir karakter alır ve bunu **ch** değişkenine atar. Ayrıca, ilk kez karşılaştığınızda anlaşılması kolay değildir.

Kümelenmiş if...else İfadeleri

Siz muhtemelen, eski karakter modlu MS-DOS sistemlerinde macera oyunlarını hatırlamayacak kadar gençsiniz, ama biz, yine de bu kavramı burada yeniden canlandıralım. "Karakterinizi" hayali bir ortamda gezdirir ve şatoları, büyücüleri, hazineleri vs keşfederdiniz. Üstelik, girdi ve çıktı olarak resim yerine metin kullanarak bunu başarırdınız. Bir sonraki program olan **ADIFELSE**, böyle bir macera oyununun küçük bir bölümünü modeller.

```
// adifelse.cpp
// macera oyunu ile IF...ELSE yapisi
#include <iostream>
using namespace std;
#include <conio.h>                  //getche() icin

int main()
   {
   char dir='a';
   int x=10, y=10;

   cout << "Type Enter to quit\n";
   while( dir != '\r' )             //Enter girilene kadar
      {
      cout << "\nYour location is " << x << "," << y;
      cout << "\nPress direction key (n,s,e,w): ";
      dir = getche();               //karakteri al
      if( dir=='n')                 //kuzeye git
         y--;
      else
         if( dir=='s' )             //guneye git
            y++;
         else
            if( dir=='e' )          //doguya git
               x++;
            else
               if( dir=='w' )       //batiya git
                  x--;
      }     //while'in sonu
   return 0;
   }     //main'in sonu
```

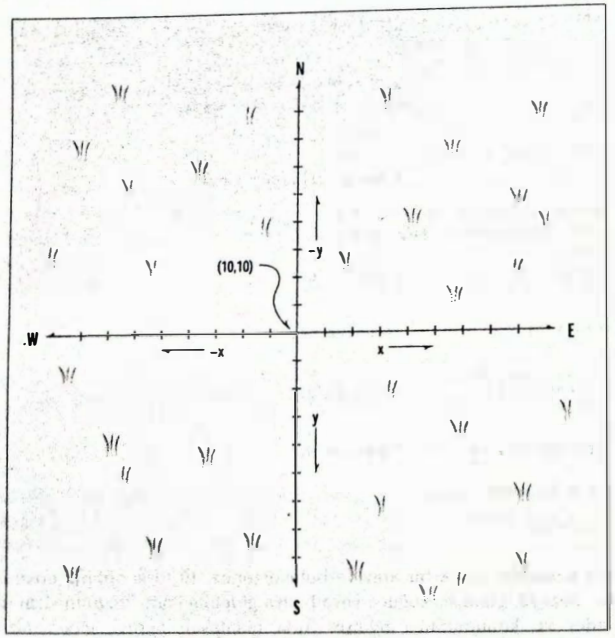
Oyun başladığında kendinizi çorak bir arazide buluyorsunuz. 10,10 koordinat noktasından başlayarak bir "adımı" kuzeye, güneye, doğuya veya batıya gidebilirsiniz. Program sizin nerede olduğunuzu takip eder ve konumunuzu bildirir. Ama maalesef, nereye giderseniz gidin karakterinizin başına heyecan verici bir şeyler gelmez. Arazi, Şekil 3.11'de gösterildiği gibi, sanki sınırsızcasına tüm yönlere doğru uzanır. Bu oyuna daha sonra bir parça daha heyecan kalacağız.

ADIFELSE ile örnek bir etkileşim şöyle olabilir:

```
Your location is 10,10
Press direction key (n,s,e,w):n
Your location is 10,9
Press direction key (n,s,e,w):e
Your location is 11,9
Press direction key (n,s,e,w):
```

Enter tuşuna basarak programdan çıkabilirsiniz.

Bu program görsel açıdan heyecana neden olmayabilir, fakat birden fazla seçeneği ele almanın yollarından biridir. Programda `if...else` ifadesinin içine yerleştirilmiş bir `if...else` ifadesi, bunun içine yerleştirilmiş başka bir `if...else` ifadesi, bunun da içine yerleştirilmiş bir `if` ifadesi kullanılır. Eğer ilk sınama koşulu yanlışsa ikinci incelenir, vs. Tüm dört koşul da kontrol edilene kadar bu işlem sırayla devam eder. Bu koşullardan herhangi biri doğrulanırsa bu koşula karşılık gelen faaliyet gerçekleştirilir – `x` veya `y` koordinatının değeri değiştirilir – ve program iç içe karar ifadelerinin tümünden çıkar. Bu tür iç içe yerleştirilmiş `if...else` ifade grupları *karar ağacı* olarak adlandırılır.



ŞEKİL 3.11: Çorak arazi.

else'i Eşlemek

İç içe `if...else` ifadelerinde potansiyel bir problem vardır: `else`'i kazara yanlış `if` ile eşleyebilirsiniz. BADELSE bununla ilgili bir örnek içeriyor:

```
// badelse.cpp
// yanlis IF ile eslenen ELSE ornegi
#include <iostream>
using namespace std;

int main()
   {
   int a, b, c;
   cout << "Enter three numbers,a,b,and c:\n ";
   cin >> a >> b >> c;
   if( a==b )
      if( b==c )
      cout << "a, b, and c are the same\n";
   else
      cout << "a and b are different\n";
   return 0;
   }
```

Tek bir `cin` ile birden fazla değer kullandık. Girdiğiniz her değerden sonra Enter'a basın. Girilen üç değer `a`, `b` ve `c` değişkenlerine atanır.

Eğer 2, 3 ve 3 girerseniz ne olur? `a` değişkeni 2, `b` değişkeni 3 değerini alır. Bu ikisi farklı olduğu için birinci koşul sınama deyimi false olur. Bu durumda `else`'in çalıştırılacağını ve ekrana *a and b are different* ifadesinin yazılacağını umarsınız. Aslında ekrana hiçbir şey yazılmaz. Neden? Çünkü `else` yanlış `if` ile eşlenmiştir. Girintileme, `else`'in ilk `if`'e ait olduğunu düşünmenize neden olabilir. Fakat `else` aslında, ikinci `if`'e aittir. İşte kural: `else` her zaman, kendi `else`'i olmayan en son `if`'e aittir.

Programın düzeltilmiş versiyonu şöyle olur:

```
if(a==b)
   if(b==c)
      cout << "a, b, and c are the same\n";
   else
      cout <<"b and c are different\n";
```

Girintilemeyi ve `else`'in gövdesindeki yazılı ifadeyi değiştirdik. Şimdi **2, 3, 3** girdiğinizde ekrana hiçbir şey yazılmaz. Fakat **2, 2, 3** girildiğinde

```
b and c are different
```

çıktısı elde edilir. Eğer `else`'i gerçekten önceki `if` ile eşlemek istiyorsanız, içteki `if`'in etrafına küme parantezi kullanabilirsiniz.

```
if(a==b)
   {
   if(b==c)
      cout << "a, b, and c are the same";
   }
else
   cout << "a and b are different";
```

Burada, girintilemeden de anlaşılacağı gibi, **else** ilk **if** ile eşlenmiştir. Parantezler, içteki **if**'i **else**'e karşı görünmez yapar.

else...if Yapısı

ADIFELSE programındaki **if...else** ifadeleri biçimsiz görünür. Ayrıca, insanlar için bu ifadeler açıklaması zor gelir, özellikle gösterildiğinden daha derin biçimde iç içelerse. Bununla birlikte, aynı ifadeleri yazmanın bir başka yolu daha vardır. Bir sonraki örnek olan **ADELSEIF** programını elde etmek için, programı sadece yeniden biçimlendirmemiz gerekir.

```
// adelseif.cpp
// macera programi ile ELSE...IF yapisi
#include <iostream>
using namespace std;
#include <conio.h>              //getche() icin

int main()
   {
   char dir='a';
   int x=10, y=10;

   cout << "Type Enter to quit\n";
   while( dir != '\r' )         //Enter girilene kadar
      {
      cout << "\nYour location is " << x << ", " << y;
      cout << "\nPress direction key (n,s,e,w): ";
      dir = getche();           //karakteri al
      if( dir == 'n')           //kuzeye git
         y--;
      else if( dir=='s' )       //guneye git
         y++;
      else if( dir=='e' )       //doguya git
         x++;
      else if( dir=='w' )       //batiya git
         x--;
      }     //while'in sonu
   return 0;
   }     //main'in sonu
```

Derleyici bunu **ADIFELSE**'nin aynısı olarak görür. Fakat bu programda **if**'leri yeniden düzenledik. böylece, **if**'ler **else**'lerin peşinden gelir. Sonuç sanki yeni bir anahtar kelime, **else...if**, gibi görünüyor. Program, sınama ifadelerinden biri doğrulanana kadar **else...if** merdiveninden aşağı doğru iner. Doğru ifadeye gelince karşılık gelen ifadeyi çalıştırır ve merdiveni terk eder. Bu biçimlendirme, **if...else** yaklaşımına kıyasla daha açık ve takibi daha kolaydır.

switch İfadesi

Eğer büyük bir karar ağacınız varsa ve tüm kararlar aynı değişkenin değerine bağlı ise, **if...else** veya **else...if** yapılarından oluşmuş bir merdiven yerine **switch** ifadesini kullanmayı düşünebilirsiniz. Nostalji düşkünlerine çekici gelebilecek **PLATTERS** adında bir örnek verelim:


```
// platters.cpp
// SWITCH ifadesi
#include <iostream>
using namespace std;

int main()
   {
   int speed;               //pikap hizi

   cout << "\nEnter 33,45,or 78: ";
   cin >> speed;            //kullanici hizi giriyor
   switch(speed)            //hiza bagli secim
      {
      case 33:              //kullanici 33 girerse
         cout << "LP album\n";
         break;
      case 45:              //kullanici 45 girerse
         cout << "Single selection\n";
         break;
      case 78:              //kullanici 78 girerse
         cout << "Obsolete format\n";
         break;
      }
   return 0;
   }
```

Bu program, kullanıcının 33, 45 veya 78 sayılarından birini girmesine bağlı olarak muhtemel üç mesajdan birini ekrana basar. Eskiler hatırlayacaklardır, bir çok şarkı içeren plaklar (LP) 33 rpm (dakikadaki dönüş sayısı) hızında dönerdi. Daha küçük olan 45'liklere ise tek şarkı kaydedilebilirdi. 78'likler ise LP'lerden ve 45'liklerden önceki formattı.

switch anahtar kelimesini parantez içindeki bir **switch** değişkeni takip eder.

```
switch(speed)
```

Birkaç sayıda **case** ifadesini küme parantezleri sınırlandırır. Her **case** anahtar kelimesinin peşinden bir sabit gelir. Bu sabit parantez içinde yazılmaz ve peşinden iki nokta işareti (:) gelir.

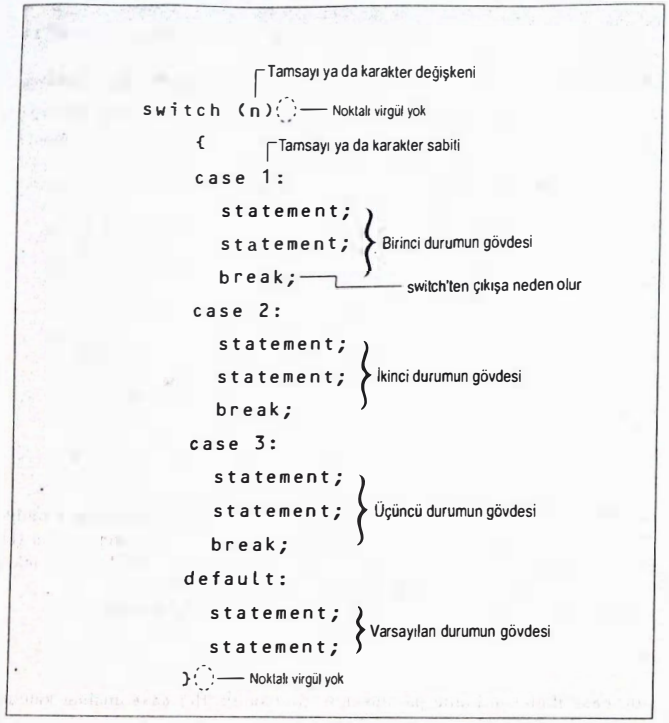
```
case 33:
```

case sabitlerinin veri tipleri **switch** değişkeninin tipiyle eşlenebilir olmalıdır. Şekil 3.12 **switch** ifadesinin söz dizimini gösteriyor.

switch'e girmeden önce program, **switch** değişkenine bir değer atamalıdır. Bu değer genellikle **case** ifadelerindeki sabitlerden biriyle eşlenir. Eğer böyle bir eşleme yapılabiliyorsa, **case** anahtar kelimesinin hemen ardından gelen ifadeler, **break**'e kadar çalıştırılır.

PLATTER'ın bir çıktı örneği şöyledir:

```
Enter 33, 45, or 78: 45
Single selection
```



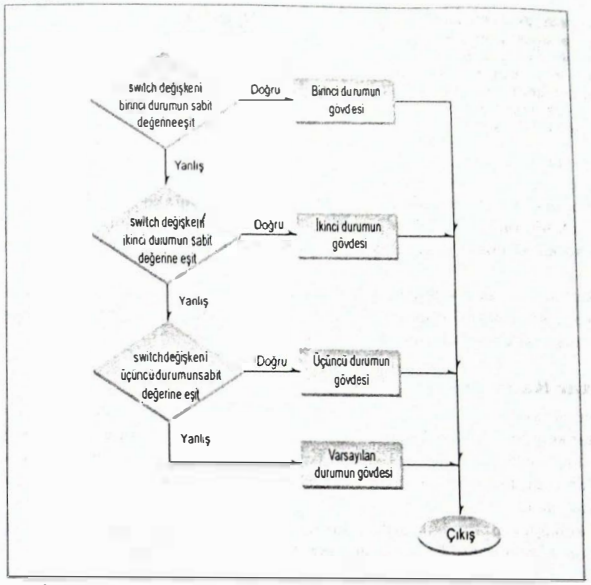
ŞEKİL 3.12: `switch` ifadesinin söz dizimi.

break İfadesi

`PLATTERS`, her `case` bölümünün sonunda `break` ifadesi içerir. `break` anahtar kelimesi, `switch` ifadesinin tümünden çıkmayı sağlar. Böylece kontrol, `switch` yapısının ardından gelen ilk ifadeye geçer. `PLATTERS`'da bu, programın sonudur.

`break` kullanmayı unutmayın. `break` olmadan kontrol, sıradaki `case` ifadesine kayar ki, bu genellikle istenmeyen bir durumdur (gerçi kimi zaman bu durum yararlı olabilir.)

Eğer `switch` değişkeninin değeri `case` sabitlerinden hiç biri ile eşlenmiyorsa, kontrol hiçbir şey yapılmadan `switch`'in sonuna kayar. `switch` ifadesinin çalışma mantığı Şekil 3.13'te gösterilmiştir. `break` anahtar kelimesi, ayrıca, döngülerden çıkmak için de kullanılır. Bu konuyu yakında ele alacağız.



ŞEKİL 3.13: `switch` ifadesinin çalışma mantığı.

Karakter Değişkenleri ile switch İfadesi

`PLATTERS` örneğindeki `switch` ifadesi, `int` tipinde bir değişkene bağlıdır. Burada `char` tipini de kullanabilirsiniz. `ADELSEIF` programımızın `ADSWITCH` adıyla yeniden yazılmış hali şöyledir:

```
// adswitch.cpp
// macera programi ile SWITCH
#include <iostream>
using namespace std;
#include <conio.h>                    //getche() icin

int main()
   {
   char dir='a';
   int x=10, y=10;

   while(dir != '\r' )
      {
      cout << "\nYour location is " << x << ", " << y;
      cout << "\nEnter direction (n,s,e,w): ";
      dir = getche();                              //karakteri al
      switch(dir)  ,                               //karaktere gore dallan
         {
         case 'n': y--; break;                     //kuzeye git
```

```
        case 's': y++; break;                   //guneye git
        case 'e': x++; break;                   //doguya git
        case 'w': x--; break;                   //batiya git
        case '\r': cout << "Exiting\n"; break;  //Enter tusu
        default: cout << "Try again\n";         //bilinmeyen karakter
        }     //switch'in sonu
    }     //while'in sonu
return 0;
}     //main'in sonu
```

`switch` değişkeni olarak `dir` isminde bir karakter değişkeni; karakter sabitleri `'n'`, `'s'` ve diğerleri ise `case` sabitleri olarak kullanılmıştır. (Son iki örnekte olduğu gibi, `switch` değişkeni olarak tamsayıları ve karakterleri kullanabilirsiniz. Ancak kayan noktalı sayıları kullanamazsınız.)

`case` anahtar kelimesini takip eden ifadeler çok kısa oldukları için tek satıra yazılmıştır. Böylece, daha az yer kaplayan bir program listesi elde edilir. Ayrıca programa, Enter tuşlandığında çıkış mesajı veren bir `case` ifadesi daha ekledik.

default Anahtar Kelimesi

`ADSWITCH` programında, `switch`'in en sonunda bir `case` ifadesi görmeyi beklerken, bunun yerine `default` anahtar kelimesi ile karşılaşırsınız. Bu anahtar kelime sayesinde, döngü değişkeni `case` sabitlerinden hiçbiriyle eşlenmezse bile `switch` yapısı bir etkinlikte bulunabilir. Bu örnekte, kullanıcı bilinmeyen bir karakter girdiğinde `Try again` (Yeniden deneyin) ifadesini yazmak için `default` kullanılır.

`switch` ifadesi, kullanıcı tarafından girilen girdiyi analiz etmek için kullanılan yaygın bir yaklaşımdır. Muhtemel karakterlerin her biri, bir `case` ifadesi ile simgelenir.

İhtiyacınız olmadığını düşünseniz bile, tüm `switch` ifadelerinde `default` anahtar kelimesini kullanmak iyi bir fikirdir. Şöyle bir yapı

```
default:
   cout << "Hata: switch girdisi hatali"; break;
```

programcıyı (veya kullanıcıyı) programın çalışmasıyla ilgili bir şeylerin yanlış olduğu konusunda uyarır. Cesur olmak adına böyle bir `default` ifadesini programımıza her zaman dahil etmeyiz. Oysa yapmalısınız. Özellikle ciddi programlarda `default`'u kullanmalısınız.

switch mi, if...else mi?

Ne zaman bir dizi `if...else` veya `else..if` ifadesi; ne zaman `switch` ifadesi kullanırsınız? Bir `else if` yapısında, içinde alakasız değişkenlerin olduğu bir dizi deyim kullanabilirsiniz. Üstelik bu deyimler istediğiniz karmaşıklıkta olabilir. Örneğin,

```
if( SteamPressure*Factor > 56 )
   // ifadeler
else if( VoltageIn + VoltageOut < 23000)
   //ifadeler
else if( day==Thursday )
   //ifadeler
else
   //ifadeler
```


Fakat bir `switch` ifadesinde, seçeneklerin her biri aynı değişkene bağlı olarak seçilir. Bir seçeneği diğerinden ayıran tek özellik bu değişkenin değeridir. `switch`'de şöyle bir ifade kullanamazsınız:

```
case a<3:
   // bir sey yap
   break;
```

`case` sabiti tamsayı veya karakter sabiti olmak zorundadır, `3` veya `'a'` gibi. Ya da sabit bir değere dönüşen bir deyim olabilir, `'a' + 32` gibi.

Bu koşullar sağlandığında `switch` ifadesi çok açıktır – yazması ve anlaşılması kolaydır. Uygun olan her durumda, özellikle karar ağacının birkaç taneden fazla ihtimali olduğunda `switch` ifadesi kullanılmalıdır.

Koşul Operatörü

Şimdi, karar operatörlerinin tuhaf bir türünü görelim. Bu operatör, yaygın olarak kullanılan bir programlama yapısının neticesinde geliştirilmiştir: Eğer bir şey doğruysa, bir değişken bir değer alır; yanlışsa, değişken başka bir değer alır. Örneğin, aşağıdaki `if...else` ifadesi `min` değişkenine `alpha` veya `beta`'dan hangisi küçükse onun değerini verir:

```
if(alpha < beta)
   min = alpha;
else
   min = beta;
```

Bu tip bir yapı o kadar yaygındır ki, C++ tasarımcıları (aslında C tasarımcıları) bunu ifade etmek için sıkıştırılmış bir yöntem geliştirdiler: *Koşul operatörü*. Bu operatör üç operand üzerinde işlem yapan iki sembolden meydana gelir. Bu, C++'taki bu tür tek operatördür; diğer operatörler bir veya iki operand üzerinde işlem yapar. Aynı program parçasının koşul operatörü kullanılarak yazılmış dengi aşağıdadır:

```
min = (alpha<beta) ? alpha : beta;
```

Bu ifadenin, eşittir işaretinin sağ tarafında kalan kısmı *koşullu deyim* (*conditional* expression) olarak adlandırılır.

```
(alpha<beta) ? alpha : beta;  // kosullu deyim
```

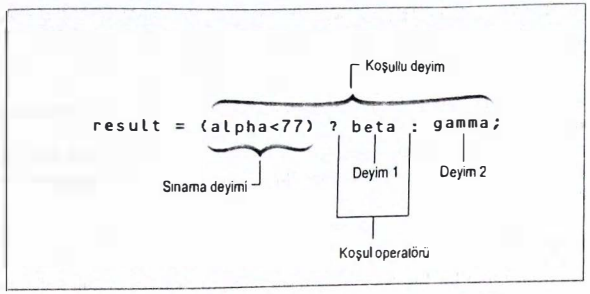
Soru işareti ve iki nokta işareti, koşullu deyimi oluşturur. Soru işaretinden önceki deyim

```
(alpha<beta)
```

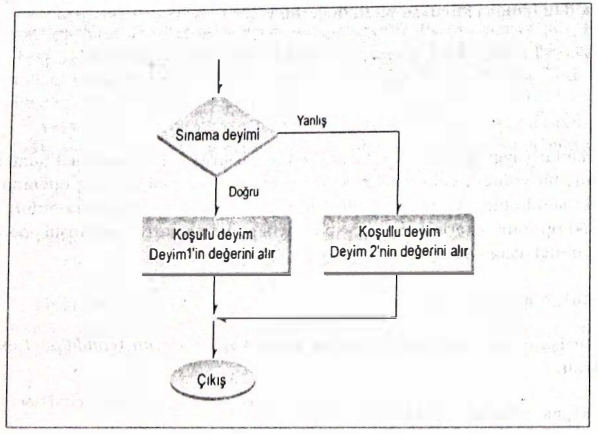
koşul sınama deyimidir. Bu, `alpha` ve `beta`, operatörün üzerinde işlem yaptığı üç operandır.

Eğer koşul sınama deyimi doğruysa, koşullu deyimin tamamı, soru işaretini takip eden operandın değerini, yani bu örnekte, `alpha` değerini alır. Eğer koşul sınama deyimi yanlışsa, koşullu deyim, iki nokta işaretinden sonra gelen operandın değerini, yani `beta` değerini alır. Koşullu deyimin etrafındaki parantezler derleyici için gerekli değildir. Ancak bunlar alışılmıştır;

ifadenin daha kolay okunmasını sağlarlar. Şekil 3.14 koşul ifadesinin sözdizimini; Şekil 3.15 ise bu ifadenin çalışma mantığını gösterir.



ŞEKİL 3.14: Koşul operatörünün söz dizimi.



ŞEKİL 3.15: Koşul operatörünün çalışma mantığı.

Koşullu deyim, başka bir değişkene atanabilir ya da bir değerin kullanılabileceği herhangi bir yerde kullanılabilir. Bizim örneğimizde koşullu deyim, `min` değişkenine atanmıştır.

Bir örnek daha verelim: `n` adlı bir değişkenin mutlak değerini bulmak üzere bir koşul operatörü kullanan bir ifadeyi inceleyelim. (Bir sayının mutlak değeri, sayı pozitifse sayının kendisi, negatifse, önündeki eksi işareti kaldırılmış halidir. Dolayısıyla mutlak değer her zaman pozitiftir.)

```
absvalue = n<0 ? -n : n;
```


`n` sıfırdan küçükse deyimin sonucu `-n`, yani pozitif bir sayı olacaktır. `n` sıfırdan küçük değilse deyimin değeri `n` olarak kalacaktır. Kısacası, bu deyimin sonucu `n`'in mutlak değerine eşittir ve o da `absvalue` değişkenine atanmıştır.

`CONDI.CPP` adını verdiğimiz aşağıdaki program, koşul operatörü kullanarak bir metin satırında sekiz karakterde bir `x` basar. Bu programı kullanarak, örneğin ekranınızdaki sekme aralıklarının yerlerini keşfedebilirsiniz.

```
// condi.cpp
// her sekiz sutunda bir 'x' yazar
// kosullu operatorun kullanimi
#include <iostream>
using namespace std;

int main()
   {
   for(int j=0; j<80; j++)     //her sutunda:
      {                        //o sutunun sekizin kati olup olmadigina bak
      char ch = (j%8) ? ' ' : 'x';  //sekizin kati ise ch'ya 'x' degeri ata
      cout << ch;              // degilse ' ' (bosluk karakteri) degeri ata
      }
   return 0;
   }
```

Sayfamızın eni kısıtlı olduğu için çıktının en sağ tarafını göremeyeceksiniz ama herhalde hayalinizde canlandırabilirsiniz, çıktımız şöyle bir şey olacaktır:

```
x       x       x       x       x       x       x       x       x
```

Burada `j` değişkeni 0'dan 79'a kadar sayarken kalan operatörünün kullanıldığı `(j%8)` deyimi sadece ve sadece `j` sekizin katı olduğunda false (yani 0) değerini alır. Bu nedenle

```
(j%8) ? ' ' : 'x'
```

koşullu ifadesi `j` sekizin katı değilse `' '` değerini (boşluk karakteri) alır. Değişkenimiz, yani `j`, 8'in katı olduğunda ise koşullu ifade `'x'` değerine sahip olur.

Kodumuzun zaten kısa ve öz olduğunu düşünebilirsiniz ama bir adım daha öteye giderek döngü bloğundaki iki satırı birleştirip bu arada `ch` değişkeninden de kurtulabiliriz:

```
cout << ( (j%8) ? ' ' : 'x');
```

Usta C++ (ve C) programcıları bu tür kodlara, yani kısacık bir satıra bol bol işlem sıkıştırmaya, bayılırlar. Ama ille de kısa koda ihtiyacımız yoksa bu yola başvurmanıza gerek yok. Bazen kısa kodlar o kadar anlaşılmaz hale gelir ki, o kodları yazmak için harcanan emeğe yazık olur. Koşullu operatörü kullanmak bile tercihe bağlı bir konudur: Bir `if...else` ifadesi ve birkaç ilave program satırı da aynı işi görür.

Mantıksal Operatörler

Şu ana kadar iki ayrı operatör grubu gördük. (İstisnai bir operatör olan koşullu operatör ayrı olmak üzere.) Birincisi `+`, `-`, `*`, `/` ve `%` operatörlerinden oluşan aritmetik operatörler grubu. İkincisi de `<`, `>`, `<=`, `>=`,`==` ve `!=` operatörlerinden oluşan ilişkisel operatörler grubu.

Şimdi de üçüncü bir operatör grubu olan *mantıksal operatörleri* (*logical operators*) inceleyelim. Bu operatörleri kullanarak Boolean değişkenlerini (yani true ve false değerleri alan `bool` tipi değişkenleri) mantıksal olarak birbiriyle işleme tabi tutmaya yararlar. Örneğin, *bugün hafta içi bir gündür* deyiminin Boolean bir değeri vardır. Çünkü bu ifade ya doğrudur ya da yanlıştır. Bir başka Boolean deyimi de *Maria arabayı aldı* olabilir. Bu iki deyimi mantıksal operatörler ile işleme tabi tutabiliriz: Bugün hafta içi bir günse ve Maria arabayı aldıysa, o zaman ben de otobüsle gitmek zorundayımdır. Burada kullandığımız mantıksal bağlaç *and*'dir ve kullandığımız iki deyimin mantıksal kombinasyonuna bir değer belirlenmesini sağlar. Şöyle ki, sadece ve sadece her iki deyim de doğruysa o zaman benim otobüsle gitmem gerekecektir.

Mantıksal VE (AND) Operatörü

C++'ta mantıksal operatörlerin Boolean deyimleri üzerinde nasıl işlem yaptığına birlikte bakalım. **ADVENAND** adlı programımızda **ADSWITCH** örneğimizde gördüğümüz macera oyununu biraz daha zenginleştirmek amacıyla bir mantıksal operatör kullanılmaktadır. (7,11) koordinatlarına bir hazine gizleyeceğiz ve kullanıcıdan bunu bulmasını isteyeceğiz.

```
// advenand.cpp
// AND mantiksal operatoru
#include <iostream>
using namespace std;
#include <process.h>                    //exit() icin
#include <conio.h>                      //getche() icin

int main()
   {
   char dir='a';
   int x=10, y=10;

   while( dir != '\r' )
      {
      cout << "\nYour location is " << x << "," << y;
      cout << "\nEnter direction (n,s,e,w): ";
      dir = getche();                   //yon bilgisini al
      switch(dir)
         {
         case 'n': y--; break;   //koordinatlari guncelle
         case 's': y++; break;
         case 'e': x++; break;
         case 'w': x--; break;
         }
      if( x==7 && y==11 )               //eger x, 7 ve y, 11 ise
         {
         cout << "\nYou found the treasure!\n";
         exit(D);                       //programdan cik
         }
      }          //switch'in sonu
   return 0;
   }          //main'in sonu
```

Bu programın kilit noktası şu if ifadesidir:

```
if( x==7 && y==11 )
```


Sadece `x` 7'ye eşitse *ve* `y` de 11'e eşitse `test` deyimi true değerini alacaktır. Mantıksal **AND** operatörü (**&&**) iki ilişkisel deyimi bir araya getirerek bu sonucu sağlar. (İlişkisel bir deyim ilişkisel operatör kullanan bir deyimdir.)

İlişkisel deyimlerin etrafındaki parantezlerin şart olmadığına dikkat edin.

```
( (x==7) && (y==11))    //icteki parantezler sart degil
```

Çünkü ilişkisel operatörler mantıksal operatörlerden daha yüksek önceliğe sahiptir.
Kullanıcı (7,10) koordinatlarına geldiğinde programla etkileşim için bir örnek verelim:

```
Your location is 7,10
Enter direction (n,s,e,w):s
You found the treasure!
```

C++'ta üç tane mantıksal operatör mevcuttur:

Operatör	Etkisi
&&	Mantıksal **AND** (ve)
||	Mantıksal **OR** (veya)
!	Mantıksal **NOT** (değil)

C++'ta mantıksal **XOR** (exclusive **OR**) operatörü yoktur.
Şimdi || ve ! operatörlerinin kullanıldığı örneklere bakalım.

Mantıksal VEYA (OR) Operatörü

Varsayalım macera oyununda ejderhalar olmasına karar verdiniz. Kullanıcı doğuya veya batıya doğru çok uzaklaşırsa ejderhalarla karşılaşsın. İşte bir örnek: **ADVENOR**, özgür maceramıza ürkütücülük engeli katmak için mantıksal **OR** (VEYA) operatörünü kullanır. Bu, **ADVENAND** programının bir varyasyonudur.

```
// advenor.cpp
// mantiksal OR operatoru
#include <iostream>
using namespace std;
#include <process.h>                    //exit() icin
#include <conio.h>                      //getche() icin

int main()
   {
   char dir='a';
   int x=10, y=10;

   while( dir != '\r' )                 //Enter tusu ile oyunundan cik
      {
      cout << "\n\nYour location is " << x << ", " << y;

      if( x<5 || x>15 )                 //x, 5'in batisindaysa VEYA 15'in dogusundaysa
         cout << "\nBeware: dragons lurk here";
      cout << "\nEnter direction (n,s,e,w): ";
      dir = getche();                   //yon bilgisini al
      switch(dir)
         {
```

```
            case 'n': y--; break;   //koordinatlari guncelle
            case 's': y++; break;
            case 'e': x++; break;
            case 'w': x--; break;
            }     //switch'in sonu
        }     //while'in sonu
    return 0;
    }     //main()'in sonu
```

Aşağıdaki deyime bir bakalım:

```
x<5 || x>15
```

x, 5'ten küçük olduğunda (oyuncu batı yönünde çok uzaklaşmışsa) veya 15'ten büyük olduğunda (oyuncu doğu yönünde çok uzaklaşmışsa, bu deyimin değeri true olur. Yine, || operatörü, < ve > ilişkisel operatörlerinden daha düşük bir önceliğe sahip. Bu nedenle, bu deyimde parantez kullanmaya gerek yoktur.

Mantıksal DEĞİL (NOT) Operatörü

Mantıksal `NOT` (değil) operatörü, (!), *tekli (unary)* operatördür, yani yalnızca tek bir operand alır. (Şu ana dek gördüğümüz operatörlerin hemen hepsi *ikili* operatörlerdi. İkili operatörler, iki operand alıyorlar. C++'taki tek *üçlü* operatör ise koşul operatörüdür.) ! operatörü, operandının mantıksal değerini tersine çevirir: Bir şey true ise, ! onu false yapar; eğer false ise, ! onu true yapar. (Hayat da bu kadar kolay yönlendirilebilseydi iyi olurdu.)

Örneğin, x, 7'ye eşitse, (x==7) doğrudur, ama !(x==7) deyimi x, 7'ye eşit değilse doğrudur. (Bu örnekte, aynı etkiyi elde etmek için *eşit değildir* ilişkisel operatörünü de kullanabilirdiniz, x != 7)

Her Tamsayı Değişkenine Bir True/False Değeri

Size şu izlenimi vermiş olabiliriz: Bir deyimin bir true/false değerine sahip olması için bir ilişkisel operatör içermesi gerekir. Oysa, her tamsayı deyimin, sadece tek bir değişkenden ibaret olsa bile, aslında bir true/false değeri vardır. x deyimi; x, 0'a eşit olmadığı her durumda true; eşit olduğunda ise false değerindedir. ! operatörünü bu duruma uygularsak, x, 0 olduğunda !x'in true olduğunu görürüz, çünkü ! operatörü x'in gerçek değerini tersine çevirir.

Şimdi bu fikirleri uygulayalım. Diyelim ki, macera oyununuzda, x ve y'nin her ikisinin de 7'nin katı olduğu konumların tümüne birer mantar yerleştirmek istiyorsunuz. (Muhtemelen biliyorsunuzdur, mantarlar oyuncu tarafından toplandığında esrarengiz güçler sağlar.) x'in 7 ile bölümünden kalan – x%7 şeklinde hesaplanabilir – sadece x, 7'nin katı ise 0 olur. Dolayısıyla mantarların konumlarını belirtmek için şöyle yazabiliriz:

```
if( x%7==0 && y%7==0)
       cout << "There 's a mushroom here.\n";
```

Ayrıca, deyimlerin ilişkisel operatörler içermeseler bile true veya false olabileceklerini hatırlayıp, ! operatörünü kullanarak daha kısa ve öz bir biçim elde edebilirsiniz.

```
if( !(x%7) && !(y%7))   // x%7 degilse ve y%7 degilse
```

Bu ifade tamamen aynı etkiye sahiptir.

&& ve || mantıksal operatörlerinin ilişkisel operatörlerden daha düşük önceliği olduğunu söylemiştik. Öyleyse, x%7 ve y%7 deyimlerinin etrafına niçin parantez koyma gereği duyuyoruz? Çünkü, mantıksal bir operatör olmasına rağmen ! operatörü tekli operatördür ve ilişkisel operatörlerde daha yüksek önceliğe sahiptir.

Öncelik Sıralarının Özeti

Şimdi, şu ana kadar gördüğümüz operatörlerin öncelik durumlarını özetleyelim. Listenin üstelerinde yer alan operatörler aşağıdakilerden daha yüksek önceliğe sahiptir. Yüksek önceliğe sahip operatörlerin değeri düşük önceliğe sahip olanlardan daha önce hesaplanır. Aynı satırda bulunan operatörlerin öncelikleri eşittir. Bir deyimi parantez içine alarak, değerinin ilk olarak hesaplanmasını sağlayabilirsiniz.

Ek B'de "C++ Öncelik Sırası Tablosu ve Anahtar Kelimeler" başlığı altında önceliklerin tam listesini bulabilirsiniz.

Operatör tipi	Operatörler	Öncelik Sırası
Tekli (unary)	!, ++, —, +, –	En yüksek
Aritmetik	Çarpma *, /, %	
	Toplama +, –	
İlişkisel	Eşitsizlik <, >, <=, >=	
	Eşitlik ==, !=	
Mantıksal	Ve &&	
	Veya ||	
Koşul	?:	
Değer atama	=, +=, –=, *=, /=, %=	En düşük

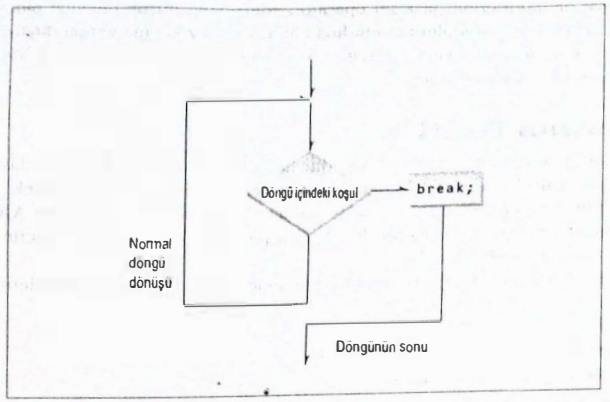
Birden fazla operatörün yer aldığı ilişkisel deyimlerde karışıklık ihtimali varsa, gerekli olsun veya olmasın parantez kullanmanız gerektiğini belirtmeliyiz. Bu parantezlerin bir zararı yoktur. Üstelik, öncelik sırası açısından bir hata yapmış olsanız bile parantezler, deyimin istediğiniz gibi çalışmasını garanti eder. Ayrıca, ne yapmak istediğinizi listeyi okuyan birine açıkça ifade eder.

Diğer Kontrol İfadeleri

C++'ta birkaç tane daha kontrol ifadesi vardır. Bunlardan birini (`break`), `switch` ifadesinde görmüştük. `break` başka yerlerde de kullanılabilir. Diğer bir ifade olan `continue`, sadece döngülerde kullanılır; üçüncü bir ifade olan `goto` ise hiç kullanılmamalıdır. Şimdi bu ifadelere sırasıyla göz atalım.

break İfadesi

Tıpkı `switch` ifadesinde olduğu gibi, `break` bir döngüde kullanıldığında döngüden çıkmayı sağlar. `break`'ten sonra çalıştırılacak ilk ifade döngü ifadesini takip eden ifadedir. Şekil 3.16 `break` ifadesinin çalışma mantığını gösterir.



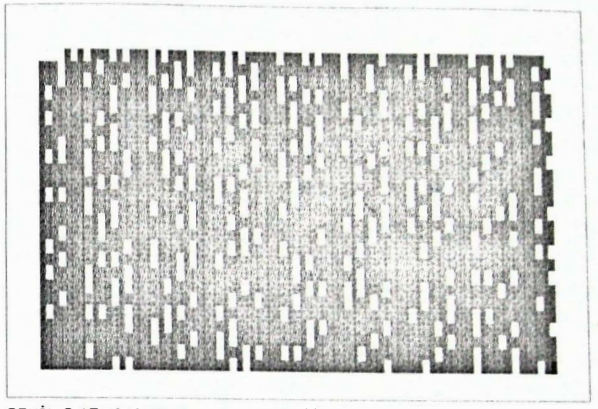
ŞEKİL 3.16: `break` ifadesinin çalışma mantığı.

`break`'i tanıtmak için işte size bir program. `SHOWPRIM`, asal sayıların dağılımını grafik formunda gösterir:

```
// showprim.cpp
// asal sayilarin dagilimi
#include <iostream>
using namespace std;
#include <conio.h>                      //getche()icin

int main()
   {
   const unsigned char WHITE = 219;     //beyaz renk (asal sayilar)
   const unsigned char GRAY  = 176;     //gri (asal olmayan sayilar)
   unsigned char ch;
                                        //her ekran konumu icin
   for(int count=0; count<80*25-1; count++)
      {
      ch = WHITE;                       //asal oldugunu varsayalim
      for(int j=2; j<count; j++)        //2'den baslayip, her tamsayiya bol
         if(count%j == 0)               //eger kalan, 0 ise
            {
            ch = GRAY;                  //sayi asal degildir
            break;                      //icteki donguden cik
            }
      cout << ch;                       //karakteri goruntule
      }
   getch();                     //herhangi bir tusa basilana kadar ekrani dondur
   return 0;
   }
```

25 satır – 80 sütunluk bir konsol ekranındaki konumlar 0'dan başlayarak 1999'a (yani, 80*25-1) kadar numaralanmıştır. Belirli bir konumdaki sayı asal ise, bu konum beyaz olarak; değilse gri olarak renklendirilir.

Şekil 3.17 elde edilecek ekran görüntüsünü gösteriyor. Tam anlamıyla söylersek, 0 ve 1 asal sayı olmamalarına rağmen, programı karmaşıklaştırmamak için beyaz olarak gösterilirler. En üstteki 0'dan 79'a kadar numaralandırılmış sütunlara bir bakın. Dikkat ederseniz, hiçbir asal sayı (2 hariç) çift sayılı sütunda yer almaz; çünkü bu sütunlardaki tüm sayılar 2 ile bölünebilir. Diğer sayılar için de bir kural geçerli midir? Eğer verilen bir sayının asal olup olmadığını anlayan bir model geliştirebilirseniz matematik dünyasını ayağa kaldırırsınız.



ŞEKİL 3.17: `SHOWPRIM` programının çıktısı.

İçteki `for` döngüsü sayının asal olmadığını belirlerse, `ch` karakterine `GRAY` (gri) değerini atar ve `break`'i çalıştırarak içteki döngüden çıkar. (`PRIME` örneğinde olduğu gibi programın tümünden çıkmak istemiyoruz, çünkü daha üzerinde çalışacağımız bir dizi sayı var.)

Dikkat ederseniz, `break` sizi sadece en içteki döngüden çıkarır. Bu, hangi yapıların birbiri içinde yer aldığından bağımsız olarak, her zaman doğrudur: `break` sizi sadece kendisinin içinde yer aldığı yapıdan dışarı çıkarır. Eğer döngünün içinde bir `switch` olsaydı, `break` sizi döngüden değil, `switch`'den dışarı çıkarmış olacaktı.

Sondaki `cout` ifadesi grafik karakterini ekrana basar; sonra döngü, sıradaki sayının asallığını test etmek için dönmeye devam eder.

ASCII Genişletilmiş Karakter Seti

Bu program *genişletilmiş ASCII karakter setinden* iki karakter kullanıyor. 128 ile 255 arasındaki sayılarla simgelen karakterler genişletilmiş ASCII karakter setini oluşturur. Genişletilmiş ASCII karakter seti Ek A'da "ASCII Tablosu" başlığı altında gösterilmiştir. 219 sayı değeri dolu bir bloğu simgeler (siyah-beyaz monitörlerde bu renk beyaza karşılık gelir); 176 ise gri bloğu simgeler.

`SHOWPRIM` örneğinde, program sona erdiği zaman ekranın yukarı kaymasını önlemek için programın son satırında `getch()` fonksiyonu kullanılmıştır. `getch()`, herhangi bir tuşa basılana kadar ekranı dondurur.

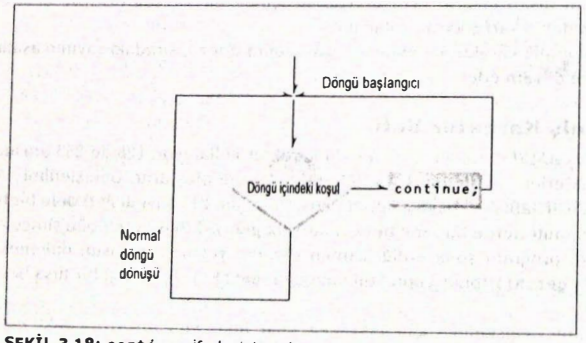
`SHOWPRIM`'deki karakter değişkenleri için `unsigned char` tipini kullanıyoruz, çünkü karakter değerleri 255'e kadar çıkabilir. `char` tipi sadece 127'ye kadar çıkabilir.

continue İfadesi

break ifadesi döngüden çıkıp, döngünün sonuna gitmeyi sağlar. Fakat, kimi zaman beklenmedik bir durum olduğunda döngünün başına dönmek isteyebilirsiniz. **continue** böyle bir etkiye sahiptir. (Daha açık bir anlatımla, **continue** kontrolü döngünün kapalı küme parantezine taşır, buradan da döngünün başına atlayabilirsiniz.) Şekil 3.18 **continue**'nun çalışma mantığını gösteriyor.

İşte size **DIVDO** örneğinin bir varyasyonu. Bu bölümde daha önce gördüğümüz bu program bölme işlemi yapar. Ancak, bu programın tehlikeli bir kusuru vardır: Eğer kullanıcı bölen olarak 0 girerse, program bir facianın içine girer ve *Divide Error* (*Hatalı Bölme*) hata mesajı ile sona erer. Bu programın yeni versiyonu olan **DIVDO2**, bu durumu daha zarif biçimde ele alır.

```
// divdo2.cpp
// CONTINUE ifadesi
#include <iostream>
using namespace std;

int main()
   {
   long dividend, divisor;
   char ch;

   do {
      cout << "Enter dividend: "; cin >> dividend;
      cout << "Enter divisor: "; cin >> divisor;
      if(divisor==0)                        //0 ile bolme
         {                                  //girisimi olursa,
         cout << "Illegal divisor\n";       //mesaji goruntule
         continue;                          //dongunun basina don
         }
      cout << "Quotient is " << dividend / divisor;
      cout << ",remainder is " << dividend % divisor;

      cout << "\nDo another? (y/n): ";
      cin >> ch;
      } while( ch != 'n' );
   return 0;
   }
```



ŞEKİL 3.18: **continue** ifadesinin çalışma mantığı.

Kullanıcı bölen için 0 değerini girerse, program bir hata mesajı verir ve kullanıcıdan diğer girdileri almak için **continue** ifadesini kullanarak döngünün başına döner.

```
Enter dividend:10
Enter divisor:0
Illegal divisor
Enter dividend:
```

Bu durumda **break** ifadesi kullanmak **do** döngüsünden ve programdan çıkmaya neden olacaktı –ki, bu gereksiz yere sert bir tepki olurdu.

do döngüsünün formatını bir parça daha sıkıştırdık. **do** açma paranteziyle aynı satırda, **while** ise kapama paranteziyle aynı satırda yer alıyor.

goto İfadesi

Burada **goto** ifadesinden sadece konuyu tamamlamak amacıyla bahsediyoruz, yoksa bu ifadeyi kullanmak iyi bir fikir olduğu için değil. Yapısal programlama prensiplerine bir parça yakınlığınız varsa, **goto**'ların çabucak, anlaması ve hata gidermesi zor, karmaşık kodlara yol açtığını bilirsiniz. Bu kitaptaki program örneklerinde yer almamasının işaret ettiği gibi, **goto** kullanmaya neredeyse hiçbir zaman gerek yoktur.

Bu kadar bilgiden sonra, işte **goto**'nun söz dizimi. Programınızda **goto** için belirlediğiniz yere bir etiket yerleştirirsiniz. Bu etiket her zaman iki nokta işareti ile sona erer. Program içinde peşinde bu etiket olan **goto** anahtar kelimesi, sizi bu etiketin olduğu satıra götürür. Aşağıdaki kod parçası bu yaklaşımı gösterir.

```
goto SystemCrash;
// diger ifadeler
SystemCrash:
// kontrol goto'dan sonra buradan devam eder
```

Özet

İlişkisel operatörler iki değerin eşit olup olmadığını, birinin diğerinden küçük olup olmadığını vs anlamak için iki değeri karşılaştırır. Sonuç, true (doğru) veya false (yanlış) değerine sahip mantıksal veya Boolean (**bool** tipinde) bir değerdir. false 0 ile, true 1 veya sıfırdan farklı herhangi bir sayı ile belirtilir.

C++'ta üç çeşit döngü vardır. **for** döngüsü, döngünün kaç kez tekrarlanacağını önceden bildiğiniz durumlarda daha sık kullanılır. **while** ve **do** döngüleri, döngünün sona ermesine neden olacak koşulun döngü içinde ortaya çıktığı durumlarda kullanılır. **while** döngüsü koşula bağlı olarak hiç tekrarlanmayabilirken, **do** döngüsü her zaman en az bir kez çalışır.

Döngü gövdesi tek bir ifadeden ibaret olabileceği gibi, küme parantezleriyle sınırlanmış birden fazla ifadenin oluşturduğu bir ifade bloğunu da içerebilir. Bir blok içinde tanımlanmış bir değişkene sadece bu blok içinde erişilebilir.

Dört çeşit karar verme ifadesi mevcuttur. **if** ifadesi, koşul sınama deyimi doğru ise, bir iş yapar. **if...else** ifadesi, koşul sınama ifadesi doğru ise bir iş yapar, değilse başka bir iş yapar. **else...if** yapısı, iç içe **if...else** ifadelerini daha okunur kılmak için, bu ifadeleri yeniden yazma yollarından biridir. **switch** ifadesi, tek bir değişkenin değerine bağlı olarak birden fazla program parçasına dallanmayı sağlar. Koşul operatörü, bir koşul sınama deyimi doğru ise bir değer, değilse başka bir değer döndürme işlemini kolaylaştırır.

Mantısal AND (VE) ve OR (VEYA) operatörleri iki Boolean deyimi birleştirerek yeni bir Boolean değer üretir. Mantıksal NOT (DEĞİL) operatörü bir Boolean değişkeni true'dan false'a veya false'dan true'ya çevirir.

`break` ifadesi kontrolü, en içteki döngünün sonuna veya içinde bulunduğu `switch`'in sonuna taşır. `continue` ifadesi kontrolü, içinde bulunduğu döngünün başına taşır. `goto` ifadesi kontrolü, etiketin bulunduğu satıra taşır.

Öncelik sırası hangi tür işlemlerin önce yapılacağını belirler. Öncelik sırası birli operatörler, aritmetik operatörleri, ilişkisel operatörler, mantıksal operatörler, koşul operatörü ve değer atama operatörleri şeklindedir.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Bir ilişkisel operatör
   a. bir operandın değerini diğerine atar.
   b. Boolean bir değer döndürür.
   c. iki operandı karşılaştırır.
   d. iki operandı mantıksal olarak karşılaştırır.
2. Bir ilişkisel operatör kullanan bir deyim yazın. Bu operatör, `george` değişkeninin değeri `sally`'ye eşit değilse true değerini döndürsün.
3. –1 true mudur, false mu?
4. Bir `for` ifadesinde kullanılan üç deyimin neler olduğunu belirtin, ne amaçla kullanıldıklarını açıklayın.
5. Döngü gövdesinde birden fazla ifade içeren bir `for` döngüsünde, noktalı virgül _____ peşinden gelir.
   a. `for` ifadesinin kendisinin
   b. birden fazla ifade içeren döngü gövdesinin kapanış parantezinin
   c. döngü gövdesindeki her ifadenin
   d. koşul sınama ifadesinin
6. Doğru/Yanlış: `for` döngüsü içindeki artırma deyimi döngü değişkenini azaltabilir.
7. 100 ile 110 arasındaki sayıları ekranda gösterecek bir `for` döngüsü yazın.
8. Bir program bloğu _____ ile sınırlanır.
9. Bir blok içinde tanımlanmış bir değişkenin erişilebilirliği hakkında hangisi doğrudur?
   a. Tanımlandığı noktadan programın sonuna kadar erişilebilirdir.
   b. Tanımlandığı noktadan fonksiyonun sonuna kadar erişilebilirdir.
   c. Tanımlandığı noktadan blok sonuna kadar erişilebilirdir.
   d. Fonksiyon boyunca erişilebilirdir.
10. 100 ile 110 arasındaki sayıları ekranda gösteren bir `while` döngüsü yazın.
11. Doğru/Yanlış: İlişkisel operatörler aritmetik operatörlerden daha yüksek önceliğe sahiptir.
12. Bir `do` döngüsü içinde döngü gövdesi kaç kez çalıştırılır?
13. 100 ile 110 arasındaki sayıları ekranda gösteren bir `do` döngüsü yazın.
14. `age` isimli değişkenin değeri 21'den büyükse ekrana `Yes` yazan bir `if` ifadesi yazın.
15. `exit()` kütüphane fonksiyonu aşağıda belirtilen durumların hangisinden çıkış sağlar?

   a. İçinde bulunduğu döngüden.
   b. İçinde bulunduğu bloktan.
   c. İçinde bulunduğu fonksiyondan.
   d. İçinde bulunduğu programdan.
16. `age` isimli değişkenin değeri 21'den büyükse ekrana `Yes` yazan, değilse `No` yazan bir `if...else` ifadesi yazın.
17. `getche()` kütüphane fonksiyonu
   a. herhangi bir tuş basıldığında bir karakter döndürür.
   b. Enter basıldığında bir karakter döndürür.
   c. herhangi bir tuş basıldığında ekranda bir karakter görüntüler.
   d. ekranda bir karakter görüntülemez.
18. Kullanıcı Enter'a basarsa `cin` ile elde edilen karakter nedir?
19. `else` her zaman _____ `if` ile eşlenir, eğer bu `if` _____ ile sınırlanmamışsa.
20. `else...if` yapısı, iç içe yerleştirilmiş `if...else` ifadesinin _____ ile elde edilir.
21. `ch` isimli değişkenin değeri `'e'` ise ekrana Evet yazan, `'h'` ise Hayır yazan, aksi halde `Bilinmeyen Yanit` yazan bir `switch` ifadesi yazın.
22. `speed` değişkeninin değeri 55'ten büyükse `ticket`'değişkenine 1; değilse, `ticket`'e 0 değeri atayan ve koşul operatörü kullanan bir ifade yazın.
23. `&&` ve `||` operatörleri
   a. iki nümerik değeri karşılaştırır.
   b. iki nümerik değeri birleştirir.
   c. iki Boolean değeri karşılaştırır.
   d. iki Boolean değeri birleştirir.
24. `limit`, 55 ise ve `speed`, 55'ten büyükse true değeri alan ve bir mantıksal operatör içeren bir deyim yazın.
25. Aşağıdaki operatör çeşitlerini öncelik sırasına göre (en yüksek önceliği olan ilk önce gelecek şekilde) düzenleyin:
26. Mantıksal, birli, aritmetik, değer atama, ilişkisel, koşul.
27. `break` ifadesi aşağıdakilerin hangisinden çıkış sağlar?
   a. Sadece en içteki döngüden.
   b. Sadece en içteki `switch`'den.
   c. Tüm döngü ve `switch`'leıden.
   d. En içteki döngü veya `switch`'ten.
28. Bir döngü içinde `continue` operatörü çalıştırılırsa kontrol _____ geçer.
29. `goto` ifadesi kontrolün aşağıdakilerden hangisine taşınmasını sağlar?
   a. Bir operatöre.
   b. Bir etikete.
   c. Bir değişkene.
   d. Bir fonksiyona.

Alıştırmalar

Yıldızlı soruların cevaplarını Ek G'de bulabilirsiniz.

1. Verilen herhangi bir sayının katlarını gösteren bir tablo üretmek istediğinizi varsayalım. Böyle bir tablo üreten bir program yazın. Kullanıcı bir sayı girince, programınız bu sayının katlarını 10 sütun-20 satırlık bir tablo şeklinde ekranda göstersin. Programınızla etkileşim şu şekilde olmalıdır (burada sadece ilk üç satır gösterilmiştir):*

```
Enter a number: 7
  7  14  21  28  35  42  49  56  63  70
 77  84  91  98 105 112 119 126 133 140
147 154 161 168 175 182 189 196 203 210
```

2. Bir sıcaklık dönüşüm programı yazın. Programınız kullanıcıya Fahrenheit'tan Celsius'a, Celsius'tan Fahrenheit'a çevirme seçeneği sunmalı; ardından dönüşümü yapmalı. Kayan noktalı sayılar kullanın. Programınızla etkileşim şu şekilde olabilir:*

```
Type 1 to convert Fahrenheit to Celsius,
     2 to convert Celsius to Fahrenheit: 1
Enter temperature in Fahrenheit: 70
In Celsius that 's 21.111111
```

3. Klavyeden girdi okuyan >> operatörü gibi operatörler, bir dizi rakamı bir sayıya çevirme becerisine sahiptir. Aynı işi gören bir program yazın. Programınız kullanıcının en fazla 6 basamağa kadar rakam girmesini sağlamalı, elde edilen sayıyı **long** tamsayı tipinde ekranda göstermeli. Rakamlar, **getche()** kullanılarak, tek tek karakter olarak okunmalı. Mevcut sayıyı 10'la çarpıp yeni rakamı ekleyerek sayıyı elde edebilirsiniz. (İpucu: ASCII'den nümerik rakama ulaşmak için 48 veya '0' değerini çıkarmalısınız.)*
İşte örnek bir etkileşim:

```
Enter a number: 123456
Number is: 123456
```

4. Dört fonksiyonu olan bir hesap makinesinin aynını yapın. Program, kullanıcının bir sayı, bir operatör ve bir sayı daha girmesini isteyecektir. (Kayan noktalı tip kullanın.) Program daha sonra belirtilen işlemi yapacaktır: İki sayıyı toplama, çıkarma, çarpma veya bölme. İşlemi seçmek için **switch** ifadesi kullanın. En son olarak, sonucu ekranda gösterin.*
Program, hesaplamayı bitirince kullanıcıya bir işlem daha yapmak isteyip istemediğini sormalıdır. Yanıt '**y**' veya '**n**' olabilir. Programla etkileşim aşağıdaki gibi olabilir:

```
Enter first number,operator,second number: 10 / 3
Answer = 3.333333
Do another (y/n)? y
Enter first number,operator,second number: 12 + 100
Answer = 112
Do another (y/n)? n
```

5. **for** döngüleri kullanarak ekranda **X**'lerden oluşan bir piramit görüntüleyen bir program yazın. Piramit şu şekilde görünmeli:

```
   X
  XXX
 XXXXX
```



```
 XXXXXXX
XXXXXXXXX
```

Yalnız piramit, burada gösterildiğinden farklı olarak 5 satır yerine, 20 satır yüksekliğinde olmalı. Bunu yapmanın bir yolu, iki tane iç içe döngüyü (biri boşluk, diğeri **X**'leri basmak için) dıştaki bir döngünün içine yerleştirmektir. Dıştaki döngü ekranda satır satır ilerlemek için kullanılabilir.

6. Bu bölümdeki **FACTOR** programını değiştirerek, kullanıcıya tekrar tekrar bir sayı soran ve bu sayının faktöriyelini hesaplayan şekle dönüştürün. Kullanıcı 0 girerse program sona ermeli. Bunu yapabilmek için **FACTOR**'deki ilgili ifadeleri **while** veya **do** döngüsü içine yerleştirebilirsiniz.
7. Bir miktar paranızı, yıllık sabit faiz oranıyla yatırırsanız ne kadar para kazanacağınızı hesaplayan bir program yazın. Başlangıçtaki para miktarını, paranın faizde kalacağı yıl sayısını ve faiz oranını (yüzde olarak) kullanıcının girmesini sağlayın. Programla etkileşim şu şekilde olabilir:

```
Enter initial amount: 3000
Enter number of years: 10
Enter interest rate (percent per year): 5.5
At the end of 10 years,you will have 5124.43 dollars.
```

İlk yılın sonunda 3000 + (3000*0.55)= 3165 dolarınız olacak. İkinci yılın sonunda 3165 + (3165*0.55)= 3339.08 dolarınız olacak. Bunu kaç yıl varsa hepsi için tekrarlayın. **for** döngüsü hesaplamayı kolaylaştıracaktır.

8. Kullanıcıya sürekli olarak iki para miktarı girmesini isteyen bir program yazın. Paralar, eski stil Britanya para biriminde olduğu gibi pound, shilling ve pence cinsinden ifade edilmelidir. ("C++ Programlama Temelleri" adlı Bölüm 2'deki Alıştırma 10 ve 12'ye bakın.) Program iki miktarı toplayıp, cevabı yine pound, shilling ve pence cinsinden vermelidir. Kullanıcıya programın sona erip ermediğini soran bir **do** döngüsü kullanın. Tipik bir etkileşim şöyle olabilir:

```
Enter first amount: £5.10.6
Enter second amount: £3.2.6
Total is £8.13.0
Do you wish to continue (y/n)?
```

İki miktarı toplarken pence değeri 11'den büyük olduğunda 1 shilling taşımak zorunda kalacaksınız. 19 shilling'den fazlası varsa 1 pound taşımalısınız.

9. Diyelim ki 6 konuğa bir akşam yemeği daveti vermek istiyorsunuz, ama masanız 4 kişilik. Bu 6 konuktan 4'ü kaç değişik şekilde masaya oturabilir? 6 konuktan herhangi biri ilk sandalyeye oturur. Kalan 5 kişiden biri ikinci sandalyeye oturur. Kalan 4'ünden biri üçüncü sandalyeye oturur ve kalan üç kişiden biri de dördüncü sandalyeye oturur. (Kalan son iki kişi ayakta beklemek zorunda kalacaklar.) Yani, 6 konuğun 4 sandalyeye oturma düzeni 6*5*4*3= 360 değişik şekilde olabilir. Herhangi sayıda konuk ve sandalye için muhtemel oturma düzenini hesaplayan bir program yazın. (Konuk sayısının sandalye sayısından hiçbir zaman az olmayacağını varsayın.) Bunu çok karmaşıklaştırmayın. Basit bir **for** döngüsü yeterli olacaktır.
10. Alıştırma 7'deki programın bir başka versiyonunu yazın. Yatırımınızın son miktarını programın hesaplaması yerine, programa son miktarı siz girin, program yıllık sabit faiz

oranıyla kaç yılda bu miktara ulaşılacağını hesaplasın. Bu problem için hangi döngü uygun? (Kesirli yıllar için endişe etmeyin; yıl için tamsayı değer kullanın.)

11. Eski stil İngiliz para birimi için üç fonksiyonlu bir hesap makinesi yapın. Eski stil İngiliz para biriminde para miktarı pound, shilling ve pence cinsinden belirtilirdi. (Bölüm 2'deki Alıştırma 10 ve 12'ye bakın.) Hesap makinesi, kullanıcının iki para miktarını toplamasına veya para miktarını kayan noktalı bir sayı ile çarpmasına imkan vermeli. (İki para miktarını çarpmak mantıklı değil; paranın karesi diye bir ifade yok. Bölmeyi de ihmal edeceğiz. Bu bölümde Alıştırma 4'teki dört fonksiyonlu sıradan hesap makinesinin genel stilini kullanın.)
12. Kesirli sayılar için dört fonksiyonlu bir hesap makinesi yapın. (Bölüm 2'deki Alıştırma 9'a ve bu bölümdeki Alıştırma 4'e bakın.) Kesirli sayılara uygulanan aritmetik işlemlerin formülleri şöyledir:

Toplama: `a/b + c/d= (a*d + b*c)/(b*d)`
Çıkarma: `a/b - c/d= (a*d - b*c)/(b*d)`
Çarpma: `a/b * c/d= (a*c)/(b*d)`
Bölme: `a/b / c/d= (a*d)/(b*c)`

Kullanıcı önce birinci kesri, sonra operatörü ve ikinci kesri girmelidir. Program, sonucu ekranda göstermeli ve kullanıcıya devam etmek isteyip istemediğini sormalıdır.

DÖRDÜNCÜ BÖLÜM

YAPILAR

Yapılar
Numaralandırmalar (Enumerations)

`float`, `char` ve `int` gibi temel veri tiplerini gördük. Bu tipteki değişkenler tek bir bilgi simgeleyebilir: Yükseklik, miktar, sayı vs. Fakat, nasıl ki sebzeler poşetler içinde, çalışanlar bölümler içinde, sözcükler cümleler içinde düzenleniyorsa, temel veri yapılarını daha karmaşık bir bütün içinde düzenlemek genellikle uygun olur. C++'daki *yapı* (*structure*) adı verilen kavram bunu gerçekleştirme yollarından biridir.

Bu bölümün ilk kısmı yapılara ayrıldı. İkinci kısımda ise bununla ilgili başka bir konuya göz atacağız: Numaralandırmalar (enumerations).

Yapılar

Bir yapı, temel değişkenlerin bir araya getirilmesidir. Bir yapı içindeki değişkenler değişik tiplerde olabilir: Bazıları `int` tipinde, bazıları `float` tipinde ya da diğer tiplerde olabilir. (Yapılar, yakında tanışacağımız dizilerden farklıdır; dizilerde tüm değişkenler aynı tipte olmak zorundadır.) Bir yapı içindeki verilere yapının *üyeleri* denir.

C programlama ile ilgili kitaplarda yapılar ileri seviyede bir özellik olarak ele alınır ve kitabın sonuna yakın bir yerde anlatılır. Ancak, C++ programcıları için nesneleri ve sınıfları anlamada yapılar, iki temel bloktan biridir. Aslında bir yapının sözdizimi, bir sınıfınkiyle hemen hemen aynıdır. Bir yapı (tipik olarak kullanıldığı şekliyle) verilerin bir araya getirilmesidir; sınıf ise hem veri hem de fonksiyonların bir araya getirilmesidir. Dolayısıyla, yapıları öğrenerek sınıfları ve nesneleri anlamada yol almış oluruz. C++'taki (ve C'deki) yapılar, Pascal gibi diğer bazı dillerdeki *kayıtlarla* (*record*) aynı amaca hizmet ediyor.

Basit Bir Yapı

Üç tane değişken içeren bir yapı ile başlayalım: Değişkenler iki tamsayı ve bir tane kayan noktalı sayı olsun. Bu yapı, bir şirketin yedek parça envanterindeki bir parçayı simgelesin. Bu yapı, tek bir parça için hangi bilgilerin gerekli olduğunu belirten bir çeşit taslaktır. Şirket birkaç çeşit mal üretir; yani, malın model numarası (kodu) yapının ilk üyesidir. Parçanın numarası sonraki üyedir. Son üye ise parçanın maliyetidir. (Parça numarasını enteresan bulmayanlar ticaretin romantizmine gözlerini çevirmeliler.)

`PARTS` programında `part` adında bir yapı ve `part` tipinde bir yapı değişkeni olan `part1`, tanımlanır. Program `part1`'in üyelerine değer atar ve bu değerleri ekranda görüntüler.

```
// parts.cpp
// yapilari tanitmak icin yedek parca envanterini ornek olarak kullanir
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct part                    //bir yapi deklare et
   {
   int modelnumber;            //parcanin model numarasi
   int partnumber;             //parca numarasi
   float cost;                 //parcanin maliyeti
   };
////////////////////////////////////////////////////////////////
int main()
   {
   part part1;                 //yapi degiskeni tanimla

   part1.modelnumber = 6244;  //yapi uyelerine deger ver
   part1.partnumber = 373;
```



```
   part1.cost = 217.55F;
                                   //yapinin uyelerini goruntule
   cout << "Model " << part1.modelnumber;
   cout << ",part " << part1.partnumber;
   cout << ",costs $" << part1.cost << endl;
   return 0;
   }
```

Programın çıktısı şuna benzer:

```
Model 6244, part 373, costs $217.55
```

`PARTS` programının üç temel parçası vardır: Yapının tanımlanması, yapı değişkeninin tanımlanması ve yapı üyelerine erişim. Şimdi bunların her birini inceleyelim.

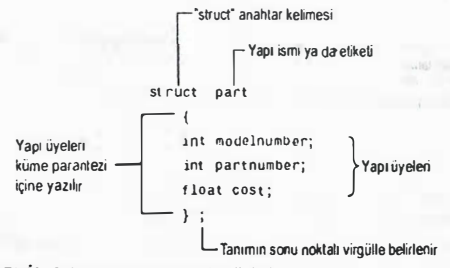
Yapıyı Tanımlamak

Yapının tanımı, yapının nasıl organize edildiğini gösterir: Yapının üyelerinin neler olacağını belirtir. Aşağıdaki örneği inceleyin:

```
struct part
   {
   int modelnumber;
   int partnumber;
   float cost;
   };
```

Yapı Tanımının Söz Dizimi

`struct` anahtar kelimesi yapının tanımını başlatır. Ardından *yapının ismi* veya *etiketi* gelir, yani `part`. Yapının üyelerinin – `modelnumber`, `partnumber` ve `cost` – deklarasyonları küme parantezi içinde yer alır. Kapanış parantezinden sonra noktalı virgül gelir ve yapının tümü sona erer. Noktalı virgülün yapılardaki bu kullanımının, bir program bloğundaki kullanımından farklı olduğuna dikkat edin. Önceden gördüğümüz gibi, döngülerde, kararlarda ve fonksiyonlarda kullanılan program blokları da küme parantezleriyle sınırlanır. Ancak, bu ifadelerde en sondaki parantezden sonra noktalı virgül kullanılmaz. Şekil 4.1 yapı tanımının söz dizimini gösterir.



ŞEKİL 4.1: Yapı tanımının söz dizimi.

Yapı Tanımının Kullanımı

Yapı tanımı, `part` tipindeki değişkenlerin tanımlanmasında sadece bir taslak görevi görür. Yapı tanımı ile yapı değişkeni tanımlanmış olmaz; yani bu tanımlama bellekte yer ayırmaz, hatta değişkenleri isimlendirmez. Bu, basit bir değişkenin tanımından farklıdır. Basit bir değişken tanımlandığında, bu değişken için bellekte yer ayrılır. Yapı tanımı ise sadece, yapı değişkenlerinin tanımlandıkları zaman nasıl görüneceklerini gösteren bir spesifikasyondan ibarettir. Bu, Şekil 4.2'de gösteriliyor.

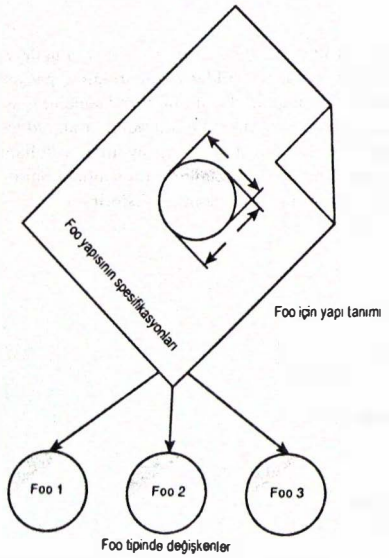
Bu tanımlamanın, Bölüm 1'de "Genel Görünüm" başlığı altında bahsettiğimiz, sınıflar ve nesneler arasındaki farklılığa benzer görünmesi tesadüf değildir. İleride göreceğimiz gibi, bir nesnenin kendi sınıfı ile arasında olan ilişki, yapı tipindeki bir değişkenin bu yapının tanımı ile arasıda olan ilişkinin aynısıdır.

Bir Yapı Değişkeninin Tanımlanması

`main()` içindeki ilk ifade,

```
part part1;
```

`part` yapısı tipinde, `part1` adında bir değişken tanımlar. Bu tanım, bellekte `part1` için yer ayırır. Peki ne kadar bir yer? `part1`'in tüm üyelerini – yani `modelnumber`, `partnumber` ve `cost` – saklamaya yetecek kadar. Bu durumda, iki `int` tipindeki değişkenin her biri için 4 byte (sistemin 32 bit olduğunu varsayarak) ve `float` içinde 4 byte demektir. Şekil 4.3 `part1`'in bellekte nasıl göründüğünü gösteriyor. (Şekil, 2 byte'lık tamsayıları gösterir.)

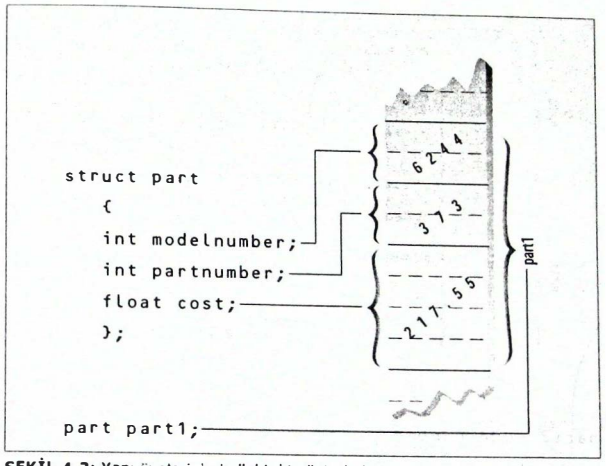


ŞEKİL 4.2: Yapılar ve yapı değişkenleri.

Bazı açılardan, `part` yapısını yeni bir veri değişkeni için bir spesifikasyon olarak da düşünebiliriz. Bu, konular ilerledikçe daha netleşecektir; fakat, bir yapı değişkeni tanımında kullanılan biçimin temel standart veri tipinde kullanılan ile (mesela `int`) aynı olduğuna dikkat edin:

```
part part1;
int var1;
```

Bu benzerlik rastlantısal değildir. C++'ın amaçlarından biri, kullanıcının tanımladığı veri tiplerinin söz dizimini ve işlevini mümkün olduğunca standart veri tiplerininkine benzetmektir. (C'de `struct` anahtar kelimesini yapı tanımlarında kullanmanız gerekir, `struct part part1` gibi; C++'ta `struct` anahtar kelimesi şart değildir.)



ŞEKİL 4.3: Yapı üyelerinin bellekteki gösterimi.

Yapı Üyelerine Erişim

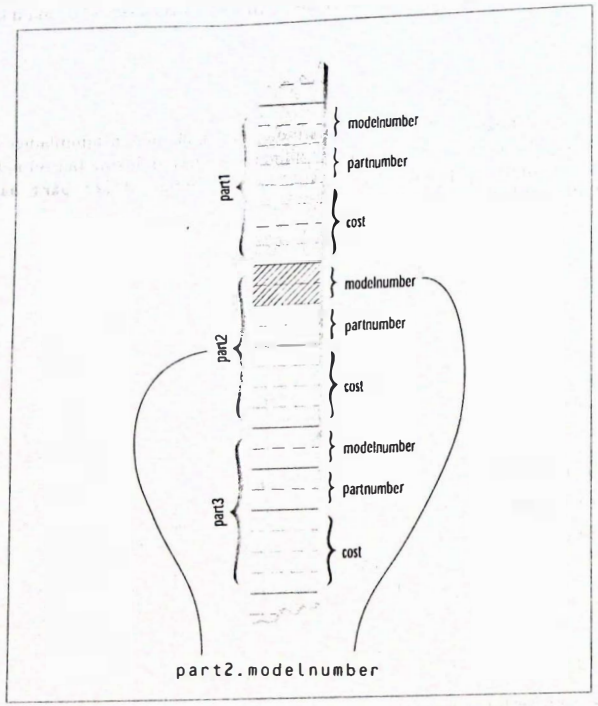
Bir yapı değişkeni tanımlandığı andan itibaren, bu değişkenin üyeleri *nokta operatörü* denilen bir operatör kullanılarak erişilebilir. İlk üyeye değerinin verilişi şöyle olabilir:

```
part1.modelnumber = 6244;
```

Bir yapı üyesinin yazımı üç parçadan oluşur: Yapı değişkeninin ismi (`part1`); bir noktadan ibaret nokta operatörü (.) ve üyenin ismi (`modelnumber`). Bu tanım, "`part1`'in `modelnumber` isimli üyesi" anlamına gelir. Nokta operatörünün asıl ismi *üye erişim operatörüdür*, fakat bu kadar uzun bir terimi elbette kimse kullanmak istemez.

Şunu aklınızdan çıkarmayın: Nokta operatörünün kullanıldığı deyimin ilk bileşeni, spesifik yapı değişkeninin ismidir (bu örnekte `part1`), yapı tanımında kullanılan isim (`part`) değildir.

Değişken ismi bir değişkeni diğerinden ayırt etmek için kullanılır. Örneğin, Şekil 4.4 'te gösterildiği gibi `part1`, `part2` vs.



ŞEKİL 4.4: Nokta operatörü.

Yapı üyeleri tıpkı diğer değişkenler gibi ele alınır. `part1.modelnumber= 6244;` ifadesinde üye, normal bir değer atama operatörü kullanılarak 6244 değerine sahip olur. Program ayrıca `cout` ifadelerinde üyelerin nasıl kullanıldığını da gösterir. Örneğin,

```
cout << "\nModel " << part1.modelnumber;
```

Bu ifadeler, yapı üyelerinin değerlerinin çıktısını alınak için kullanılır.

Diğer Yapı Özellikleri

Yapılar şaşırtıcı derecede çok yönlüdür. Şimdi, yapı söz diziminin ve kullanımının diğer ilave özelliklerine göz atalım.

Yapı Üyelerine İlk Değer Atanması

Aşağıdaki örnek, yapı değişkeni tanımlandığında yapı üyelerine nasıl ilk değer atanabileceğini gösteriyor. Bu örnek ayrıca belirli bir yapı tipinde birden fazla değişken tanımlanabileceğini belirtiyor (şimdiye kadar zaten bundan şüphelenmiş olduğunuzu umuyoruz).

PARTINIT için program listesi şöyledir:

```
// partinit.cpp
// yapi degiskenlerine ilk deger atamasi
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct part                    //yapiyi belirtin
   {
   int modelnumber;            //parcanin model numarasi
   int partnumber;             //parca numarasi
   float cost;                 //parcanin maliyeti
   };
////////////////////////////////////////////////////////////////
int main()
   {                           //degiskene ilk degerini ata
   part part1 = {6244, 373, 217.55F };
   part part2;                 //degisken tanimla
                               //ilk degiskeni ekranda goster
   cout << "Model " << part1.modelnumber;
   cout << ",part " << part1.partnumber;
   cout << ",costs $" << part1.cost << endl;

   part2 = part1;              //ilk degiskeni ikinciye ata
                               //ikinci degiskeni ekranda goster
   cout << "Model " << part2.modelnumber;
   cout << ",part " << part2.partnumber;
   cout << ",costs $" << part2.cost << endl;
   return 0;
   }
```

Bu program `part` tipinde iki değişken tanımlar: `part1` ve `part2`. Program, `part1`'e ilk değerini verir, `part1`'in üyelerinin değerlerini ekranda görüntüler, `part1`'i `part2`'ye atar ve `part2`'nin üyelerinin değerlerini ekranda görüntüler.

İşte programın çıktısı:

```
Model 6244, part 373, costs $217.55
Model 6244, part 373, costs $217.55
```

Sonuç hiç de şaşırtıcı değildir. Bir değişken diğerine eşitlendiği için aynı çıktı ekranda tekrarlanarak gösterilir.

`part1` yapı değişkeninin üyelerinin değerleri, değişken tanımlandığı sırada verilir:

```
part part1= { 6244, 373, 217.55 };
```

Yapı üyelerine atanacak değerler küme parantezi içine yazılır ve virgül ile birbirinden ayrılır. Listedeki ilk değer, birinci üyeye atanır; ikinci ikinciye atanır ve bu şekilde devam eder.

Değer Atama İfadelerinde Yapı Değişkenleri

PARTINIT programında görülebileceği gibi, bir yapı değişkeni diğerine atanabilir:

```
part2 = part1;
```

part1'in üyelerinin her birinin değeri **part2**'nin karşılık gelen üyesine atanır. Büyük bir yapının düzinelerce üyesi olabileceği için böyle bir ifade, bilgisayarın hatırı sayılır miktarda iş yapmasını gerektirebilir.

Bir yapı değişkeninin diğerine atanabilmesi için her ikisinin de aynı yapı tipinde olması gerektiğine dikkat edin. Eğer bir yapı tipindeki bir değişkeni başka bir yapı tipindeki değişkene atamaya çalışırsanız, derleyici bu durumdan şikayetçi olacaktır.

Bir Ölçüm Örneği

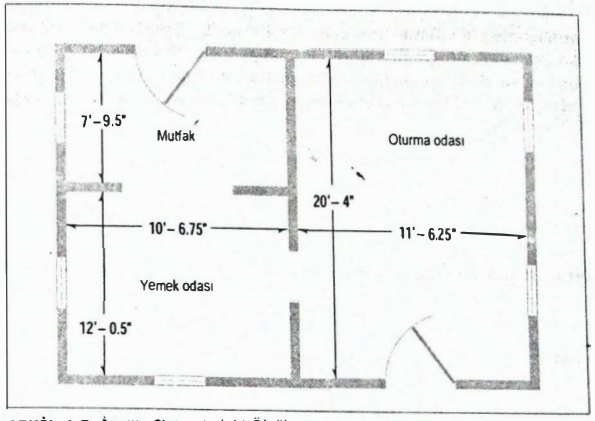
Şimdi, değişik türde bilgileri gruplamak için bir yapının nasıl kullanılabileceğini görelim. Hayatınızda bir mimari çizime bir kez bakmışsanız, mesafelerin (en azından Birleşik Devlet-lerde) ayak (feet) ve inç cinsinden olduğunu biliyorsunuzdur. (Büyük ihtimalle, bir ayağın 12 inç olduğunu da biliyorsunuzdur.) Oturma odasının uzunluğu, söz gelişi, 15'-8" olarak verilebilir; bu, 15 ayak artı 8 inç demektir. Tire, negatif sayı işareti değildir, sadece ayak olarak ifade edilen değeri inç değerinden ayırır. Bu, İngiliz ölçme sisteminin bir parçasıdır. (Burada, İngiliz sistemi ve metrik sistemin faziletleriyle ilgili bir yorum yapmayacağız.) Şekil 4.5 İngiliz sistemindeki tipik uzunluk ölçülerini gösteriyor.

Diyelim ki, İngiliz sistemini kullanan bir çizim ya da mimari programı geliştirmek istiyorsunuz. Mesafeleri ayak ve inç ile simgelenen iki sayı halinde saklamak uygun olacaktır. Sıradaki örnek olan **ENGLSTRC**, bu işin yapı kullanılarak nasıl yapılabileceği hakkında bir fikir verir. Bu program, **Distance** tipindeki iki ölçümün birbiriyle nasıl toplanabileceğini de gösterir.

```
// englstrc.cpp
// Ingiliz olculerini kullanarak yapilar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct Distance              //Ingiliz olculeri cinsinden uzaklik
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance d1, d3;          //iki uzunlugu tanimla
   Distance d2 = {11, 6.25 }; //bir uzunlugu tanimla ve degerini ata
                             //d1 uzunlugunu kullanicidan al
   cout << "\nEnter feet: "; cin >> d1.feet;
   cout << "Enter inches: "; cin >> d1.inches;
                             //d3'u elde etmek icin d1 ve d2 uzunluklarini topla
   d3.inches = d1.inches + d2.inches; //inches'leri topla
   d3.feet = 0;              //(muhtemel bir elde degeri icin)
   if(d3.inches >= 12.0)     //toplam 12.0'yi gecerse,
      {                      //inches'i 12.0 azalt
      d3.inches -= 12.0;     //ve
      d3.feet++;             //feet'i 1 artir
```



```
   }
d3.feet += d1.feet + d2.feet;    //feet'leri topla
                                 //tum uzunluklari ekranda goster
cout << d1.feet << "\'-" << d1.inches << "\" +";
cout << d2.feet << "\'-" << d2.inches << "\" =";
cout << d3.feet << "\'-" << d3.inches << "\"\n";
return 0;
}
```



ŞEKİL 4.5: İngiliz Sistemindeki Ölçüler.

Burada, **Distance** yapısı iki üyeye sahiptir: **feet** ve **inches**. **inches** değişkeni ondalık sayı içerebilir. Bu nedenle, bu değişken için **float** tipini kullanacağız. **feet** her zaman tamsayıdır, bu nedenle, bu değişkenler için **int** tipini kullanacağız.

Bu şekilde iki uzaklık tanımlayacağız: **d1** ve **d3**. Bu değişkenlere başlangıç değeri vermeyeceğiz. Bir başka değişken olan **d2**'ye ise 11'-6.24" başlangıç değerini atayacağız. Program, kullanıcıya ayak ve inç cinsinden bir uzaklık ölçüsü girmesini ister ve kullanıcının girdiği bu uzaklığı **d1** değişkenine atar. (İnç olarak girilen değer 12'den küçük olmalıdır.) Daha sonra program **d1** ve **d2**'yi toplayarak sonucu, toplam mesafeyi ifade eden **d3** değişkenine atar. Program en son olarak ise, başta girilen iki uzaklığı ve toplam uzaklığı ekrana basar. Örnek bir çıktı şu şekilde olabilir:

```
Enter feet: 10
Enter inches: 6.75
10'-6.75" + 11'-6.25"= 22'-1"
```

İki uzaklık değişkenini aşağıdaki gibi bir program ifadesi ile eklemenin mümkün olmadığına dikkat edin:

```
d3 = d1 + d2; //ENGLSTRC'de bu mumkun degil
```

Bu neden mümkün değil? Çünkü C++'ta `Distance` tipindeki değişkenleri eklemeyi bilen standart bir fonksiyon yok. Bizim bildiğimiz + operatörü, `float` gibi tiplere ait değişkenler üzerinde işlem yapabilir ama kendi tanımladığımız, `Distance` gibi türler üzerinde çalışamaz. (Ancak Bölüm 8'de göreceğimiz gibi, sınıfları kullanmanın yararlarından biri de, kullanıcının tanımladığı veri tipleri üzerinde çalışabilen işlemleri tanımlayabilmek ve bunları kullanabilmektir.)

İç İçe Yapılar

Yapıları başka yapıların içinde kümelemek mümkündür. Bunun bir örneğini `ENGLSTRC`'nin değişik bir versiyonu olan aşağıdaki programda görebiliriz. Bu programda tipik bir odanın ebatlarını, yani enini ve boyunu, kaydedeceğimiz bir veri yapısı oluşturmayı istiyoruz. Şu anda İngiliz uzunluk sistemi ile uğraştığımız için odanın enini ve boyunu kaydetmek amacıyla `Distance` tipinde iki değişken kullanacağız.

```
struct Room
   {
   Distance length;
   Distance width;
   }
```

Aşağıdaki `ENGLAREA` adlı program, bir odayı temsil etmek için `Room` adlı yapıdan yararlanır.

```
// englarea.cpp
// ic ice yapilar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct Distance                  //Ingiliz uzunluk sistemi
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////////
struct Room                      //dikdortgen alan
   {
   Distance length;              //dikdortgenin boyu
   Distance width;               //dikdortgenin eni
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Room dining;                  //bir oda tanimla

   dining.length.feet = 13;      //odanin ebatlarini ata
   dining.length.inches = 6.5;
   dining.width.feet = 10;
   dining.width.inches = 0.0;
                                 //en ve boyu donustur
   float l = dining.length.feet + dining.length.inches/12;
   float w = dining.width.feet + dining.width.inches/12;
                                 //alanini bul ve ekrana yaz
   cout << "Dining room area is " << l * w
        << " square feet\n" ;
   return 0;
   }
```


Bu program, `Room` tipinde tek bir değişken olan `dining`'i şu satırda tanımlar:

```
Room dining; //Room tipinden dining degiskeni
```

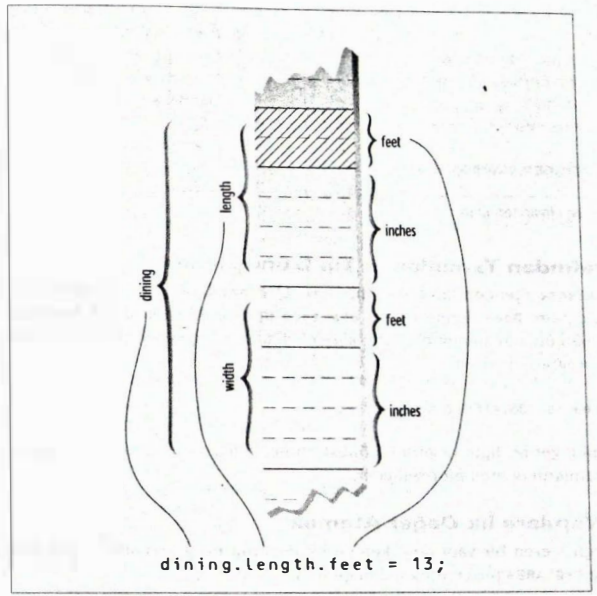
Program daha sonra da tanımladığımız yapının çeşitli üyelerine değer atamaları yapar.

Kümelenmiş Yapıların Üyelerine Erişim

Yapılar kümelendiğinde, üyelere erişmek için nokta operatörünü iki kez kullanmak gerekir.

```
dining.length.feet= 13
```

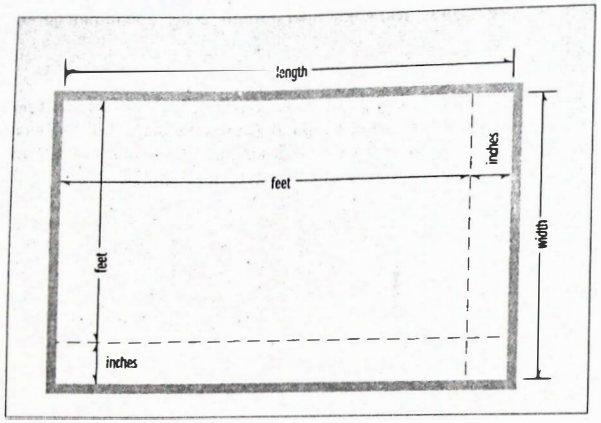
Bu ifadede `dining`, daha önceden de bildiğimiz üzere, yapı değişkeninin adıdır; `length` bir dış yapının (`Room`'un) bir üyesidir; `feet` ise iç yapının (`Distance`'ın) üyesidir. Dolayısıyla, yukarıdaki ifade şu anlama gelmektedir: "`dining` değişkeninin üyesi olan `length`'in üyesi `feet`'i al ve ona 13 değerini ata." Şekil 4.6 bu işlemin nasıl gerçekleştiğini açıklar.



ŞEKİL 4.6: Nokta operatörü ve kümelenmiş yapılar.

`dining`'in üyelerine değer ataması yapıldıktan sonra program, odanın taban alanını Şekil 4.7'de anlatıldığı gibi hesaplar.

Taban alanını bulmak için, program **Distance** tipinden olan değişkenlere kaydedilmiş en ve boy rakamlarını **float** tipinden olan ve uzunlukları ayak cinsinden ifade eden **w** ve **l** değişkenlerine dönüştürür. Söz konusu **w** ve **l** değişkenleri, **Distance**'ın **feet** üyesi ile **inches**'i ekleyip sonucu 12'ye bölmek suretiyle hesaplanır. **feet** üyesi toplama işleminden önce **float** tipine otomatik olarak dönüştürülür ve toplama işleminin sonucu da **float** olur. Daha sonra **w** ve **l** değişkenleri çarpılarak taban alanı bulunur.



ŞEKİL 4.7: Ayak ve inç cinsinden alan.

Kullanıcı Tarafından Tanımlanan Tip Dönüşümleri

Programımızın **Distance** tipinden iki değişkeni (**w** ve **l**) **float** tipine çevirdiğine dikkat edin. Aslında böylece program, **Room** tipinin (**Room**, **Distance** tipinde iki yapı olarak tanımlanır) bir yapısı olarak saklanan odanın alanını da, ayak kare cinsinden simgeleyen tek bir kayan noktalı sayıya çevirir. İşte programın çıktısı:

```
Dining room area is 135.416672 square feet
```

Bir tipin değerini diğer bir tipin değerine dönüştürmek, kullanıcı tarafından tanımlanan veri tipleri içeren programların önemli bir özelliğidir.

Kümelenmiş Yapılara İlk Değer Atamak

Kendi içinde bir yapı içeren bir yapı değişkenine ilk değerini nasıl atarsınız? Aşağıdaki ifade **dining** değişkenine **ENGLAREA** programındaki değerlerin aynısını atar:

```
Room dining= { {13, 6.5}, {10, 0.0} };
```

Room içinde yer alan **Distance** tipindeki her yapıya ayrı ayrı değer atanır. Bunun için değerleri küme parantezleri içine alıp, virgülle ayırmak gerektiğini hatırlayın. İlk **Distance**'a aşağıdaki değer atanır:


```
{13, 6.5}
```

ikinciye ise

```
{10, 0.0}
```

değeri atanır. Bu iki **Distance** değeri daha sonra **Room** değişkenine değer atamak için kullanılır. Değerler yine küme parantezi içinde ve virgülle ayrılmış olarak yazılır.

Kümelenme Derinliği

Teorik olarak yapıların kümelenmesi herhangi bir derinlikte olabilir. Apartmanları tasarlayan bir programda kendinizi şu tür ifadelerle karşı karşıya bulabilirsiniz:

```
apartment1.laundry_room.washing_machine.width.feet
```

İskambil Oyunu Örneği

Şimdi değişik türde bir örneği inceleyelim. Bu örnek, iskambil kağıdı oyununu modellemek için bir yapı kullanır. Bu program karnavallarda profesyonel kumarbazlar tarafından oynan bir oyunu örnek alır. Kumarbaz size üç kağıt gösterir, sonra bunları tersini çevirerek masaya koyar ve yerlerini kendi aralarında birkaç kez değiştirir. Belli bir kağıdın nerede olduğunu doğru tahmin ederseniz, kazanırsınız. Her şey göz önünde olur, ancak kumarbaz kağıtları öylesine hızlı ve şaşırtıcı şekilde değiştirir ki oyuncu hemen her zaman kartın ucunu kaçırır ve oyunu kaybeder. Elbette parasını da.

Bir iskambil kağıdını simgelemek için programın kullandığı yapı şöyledir:

```
struct card
   {
   int number;
   int suit;
   };
```

Bu yapı, kartın üzerindeki sayıyı ve kartın ait olduğu takımı saklamak için ayrı üyeler kullanır. **number** 2'den 14'e kadar değişir. 11, 12, 13 ve 14 sırasıyla vale, kız, papaz ve ası simgeler (bu, pokerde kullanılan sıradır.) **suit** 0 ile 3 arasında değişiyor; bu dört sayı maça, karo, kupa ve sineği simgeler.

CARDS'ın program listesi şöyledir:

```
// cards.cpp
// iskambil kartlarini kullanarak yapilar
#include <iostream>
using namespace std;

const int clubs = 0;              //kart takimlari
const int diamonds = 1;
const int hearts = 2;
const int spades = 3;
const int jack = 11;              //resimli kartlar
const int queen = 12;
const int king = 13;
const int ace = 14;
```

```
//////////////////////////////////////////////////////////////
struct card
   {
   int number;                    //2'den 10'a, vale, kiz, papaz, as
   int suit;                      //maca, karo, kupa, sinek
   };
//////////////////////////////////////////////////////////////
int main()
   {
   card temp, chosen, prize;        //kartlari tanimla
   int position;

   card card1 = { 7, clubs };       //card1'e deger ata
   cout << "Card 1 is the 7 of clubs\n";

   card card2 = { jack, hearts };   //card2'ye deger ata
   cout << "Card 2 is the jack of hearts\n";

   card card3 = { ace, spades };    //card3'e deger ata
   cout << "Card 3 is the ace of spades\n";

   prize = card3;                   //bu karti kopyala, hatirlamak icin

   cout << "I 'm swapping card 1 and card 3\n";
   temp = card3; card3 = card1; card1 = temp;

   cout << "I 'm swapping card 2 and card 3\n";
   temp = card3; card3 = card2; card2 = temp;

   cout << "I 'm swapping card 1 and card 2\n";
   temp = card2; card2 = card1; card1 = temp;

   cout << "Now, where (1, 2, or 3) is the ace of spades? ";
   cin >> position;

   switch (position)
      {
      case 1: chosen = card1; break;
      case 2: chosen = card2; break;
      case 3: chosen = card3; break;
      }
   if(chosen.number == prize.number &&     //kartlari karsilastir
        chosen.suit == prize.suit)
      cout << "That 's right! You win!\n";
   else
      cout << "Sorry. You lose.\n";
   return 0;
   }
```

Programın bir çalışma örneği aşağıdaki gibidir:

```
Card 1 is the 7 of clubs
Card 2 is the jack of hearts
Card 3 is the ace of spades
I 'm swapping card 1 and card 3
I 'm swapping card 2 and card 3
I 'm swapping card 1 and card 2
Now, where (1, 2, or 3) is the ace of spades? 3
Sorry. You lose.
```


Bu örnekte talihsiz oyuncu yanlış kartı seçti (doğru yanıt 2 idi.)

Program, resimli kartlar ve kart takımları için `const int` tipinde birkaç tane değişken tanımlayarak başlar. (Programda bu değişkenlerin hepsi kullanılmaz, ancak programın bütünlüğü açısından değişkenlerin tümü dahil edildi.) Sonra `card` yapısı belirlenir. Program, bunun ardından, üç tane ilk değeri atanmamış `card` tipinde değişken tanımlar: `temp`, `chosen` ve `prize`. Program ayrıca, üç rasgele değer atanmış üç tane kart daha tanımlar – `card1`, `card2` ve `card3`. Program kullanıcıya bilgi vermek için bu kartların değerlerini ekrana yazar. Sonra bu kart değerlerinden birini, hatırlamak amacıyla, `prize` isimli `card` değişkenine atar. Oyunun sonunda oyuncunun konumunu tahmin etmesi gereken kart, işte bu kart.

Daha sonra program kartları yeniden düzenler. Program birinci ve üçüncünün, ikinci ve üçüncünün, birinci ve ikincinin yerlerini değiştirir. Her seferinde ne yaptığını kullanıcıya anlatır. (Eğer programı çok kolay bulduysanız, kartları biraz daha karıştırmak için bu tür ifadelerden daha fazla ekleyebilirsiniz. İfadeleri ekranda kısa bir süre için yakıp söndürmek teşvik edici olacaktır.)

En son olarak, program kullanıcıya söz konusu kartın hangi konumda olduğunu sorar. Program, `chosen` isminde bir `card` değişkenine bu konumdaki kartın değerini atar ve `chosen` ile `prize` kartını karşılaştırır. İkisi eşleşiyorsa, oyuncu kazanmıştır, değilse oyun kaybedilmiştir.

Kartları değiştirmek ne kadar kolay, dikkat edin:

```
temp = card3; card3 = card1; card1 = temp;
```

Kartlar yapıları simgeliyor olsalar bile, yapılar ile işlem yapan değer atama operatörünün (=) becerisi sayesinde kartlarla çok doğal biçimde istenildiği gibi oynanabilir.

Ama ne yazık ki, yapılar aritmetik olarak toplanamadıkları gibi, birbirleriyle karşılaştırılamazlar. Şöyle yazamazsınız:

```
if ( chosen == prize)      //henuz gecerli degil
```

== operatörüne, `card` yapısını tanıyabileceği bir rutin eklenmiş olmadığı için bu ifade kurallara uygun değildir. Yine de, toplamada olduğu gibi bu problem, operatörler aşırı yüklenerek çözülebilir. Bunu daha sonra göreceğiz.

Yapılar ve Sınıflar

Yapıların becerileri hakkında sizi bir parça yanlış yönlendirdiğimizi itiraf etmeliyiz. Yapıların genellikle verileri saklamak için, sınıfların ise hem verileri hem de fonksiyonları saklamak için kullanıldıkları doğrudur. Ancak, C++'ta yapılar, aslında veri ve fonksiyonları saklayabilir. (C'de sadece veri saklayabilirler.) C++'ta yapılar ve sınıflar arasındaki söz dizimsel fark asgari düzeydedir. Bu nedenle, teorik olarak, neredeyse biri diğerinin yerine kullanılabilir. Fakat birçok C++ programcısı, bu bölümde olduğu gibi, yapıları sadece veriler için kullanır. Bölüm 6'da, "Nesneler ve Sınıflar" başlığı altında da göreceğimiz gibi, sınıflar genellikle verileri ve fonksiyonları saklamak için kullanılır.

Numaralandırmalar (Enumerations)

Yapılar, önceden de gördüğümüz gibi, kullanıcı tarafından tanımlanan veri tiplerini sunma yollarından biri olarak ele alınabilir. Kendi veri tipinizi tanımlamanın bir başka yolu ise *numaralandırmadır* (*enumeration*). C++'ın bu özelliği yapılar kadar kritik öneme sahip değil-

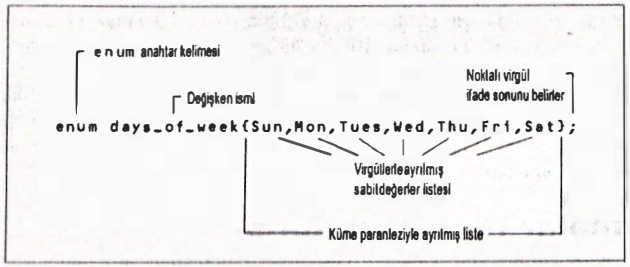
dir. Numaralandırma hakkında hiçbir şey bilmeden de C++'ta kusursuz nesne yönelimli programlar yazabilirsiniz. Yine de, numaralandırmalar C++'ın ruhuna çok uygundur, çünkü size kendi veri tiplerinizi tanımlama imkanı verdiği için programlama işleminizi kolaylaştırıp, netleştirebilir.

Haftanın Günleri

Numaralandırılmış tipler, bir veri tipinin alabileceği sınırlı (genellikle kısa) değerler listesini önceden bildiğiniz durumlarda işe yararlar. Aşağıda görülen **DAYENUM** isimli program örneği haftanın günleri için numaralandırma kullanır:

```
// dayenum.cpp
// enum tipleri
#include <iostream>
using namespace std;

                                //enum tipini belirle
enum days_of_week {Sun, Mon, Tue, Wed, Thu, Fri, Sat };

int main()
   {
   days_of_week day1, day2;     //days_of_week tipinde
                                //degiskenleri tanimla
   day1 = Mon;                  //degiskenlere
   day2 = Thu;                  //deger ata

   int diff = day2 - day1;      //tamsayi aritmetigi yapabilirsiniz
   cout << "Days between = " << diff << endl;

   if(day1 < day2)              //karsilastirma yapabilirsiniz
      cout << "day1 comes before day2\n";
   return 0;
   }
```

enum deklarasyonu, bu tipin izin verilen değerlerinin isimlerinin tamamının yer aldığı, bir isimler kümesi tanımlar. Bu izin verilen değerlere *enümeratörler* denir. **enum** tipindeki **days_of_week**'in yedi tane enümeratörü vardır: Sun, Mon, Tue vs. Sat'a kadar. Şekil 4.8'de bir **enum** deklarasyonunun söz dizimi görülüyor.



ŞEKİL 4.8: **enum** deklarasyonunun söz dizimi.

Bir numaralandırma, muhtemel değerlerin tümünün bir listesidir. Bu, mesela bir **int** spesifikasyonundan farklıdır. **int** spesifikasyonu, değerler aralığı cinsinden ifade edilir. **enum** da ise muhtemel her değere belirli bir isim vermeniz gereklidir. Şekil 4.9 **int** ve **enum** arasındaki farkı gösteriyor.

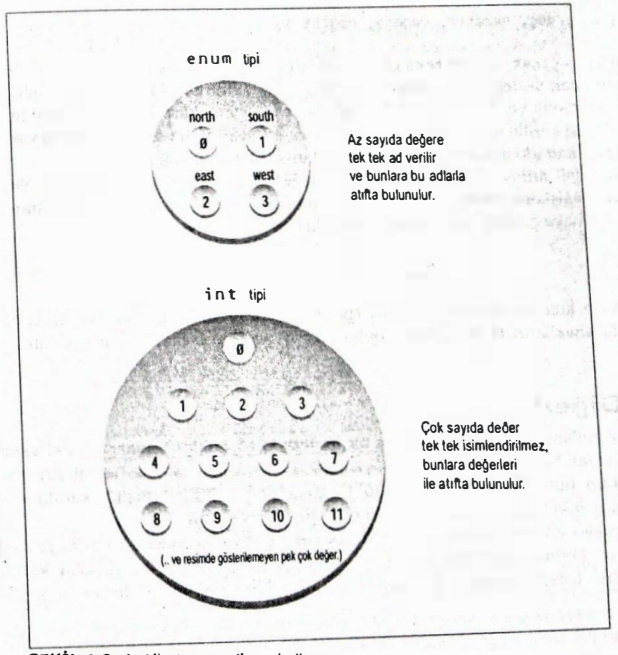
days_of_week'i **enum** tipinde gösterildiği gibi tanımlar tanımlamaz, artık bu tipte değişkenler tanımlayabilirsiniz. **DAYENUM** böyle iki değişkene sahiptir, **day1** ve **day2**. Bu değişkenler şu ifade ile tanımlanır:

```
days_of_week day1, day2;
```

(C'de tip isminden önce **enum** anahtar kelimesini kullanmanız gerekir. Şu şekilde:

```
enum days_of_week day1, day2;
```

C++'ta bu, şart değildir.)



ŞEKİL 4.9: **int**'lerin **enum**'ların kullanımı.

day1 ve **day2** gibi numaralandırılmış tipteki değişkenlere, **enum** deklarasyonunda listelenen değerlerden herhangi biri değer olarak verilebilir. Örnekte bu değişkenlere **Mon** ve **Tue** değerlerini veriyoruz. Deklarasyonda listelenmeyen değerleri kullanamazsınız. Şu tür ifadeler kurala uymaz:

```
day1 = halloween;
```

enum tipleri üzerinde standart aritmetik operatörleri kullanabilirsiniz. Programda iki değeri birbirinden çıkarıyoruz. Ayrıca, programda gösterdiğimiz gibi, karşılaştırma operatörlerini de kullanabilirsiniz. Programın çıktısı şöyledir:

```
Days between = 3
day1 comes before day2
```

Aritmetik veya ilişkisel operatörleri **enum** tipleri ile kullanmanın fazla bir esprisi yoktur. Örneğin, aşağıdaki gibi bir deklarasyonda

```
enum pets { cat, dog, hamster, canary, ocelot };
```

dog + canary veya **(cat < hamster)** gibi deyimlerin ne anlama geldiği açık olmayabilir.

Numaralandırmalar, derleyici tarafından tamsayı gibi ele alınırlar. Bu, bu tipteki değişkenler üzerinde neden aritmetik ve ilişkisel işlemler yapabileceğinizi açıklar. Alışılmış biçimde, listedeki ilk isme 0 değeri verilir, ikinciye 1 değeri verilir ve bu şekilde devam eder. **DAYENUM** örneğinde **Sun**'dan **Sat**'a kadarki değerler 0-6 arasındaki tamsayılar şeklinde saklanır.

enum tipleri ile ilgili aritmetik işlemler tamsayı değerler üzerinde gerçekleştirilir. Yine de, derleyiciniz **enum** değişkenlerinizin aslında birer tamsayı olduğunu bilmesine rağmen, bu avantajdan yararlanmaya çalıştığınızda dikkatli olmalısınız. Eğer

```
day1 = 5;
```

şeklinde bir ifade kullanıyorsanız, derleyici (programı derleyecektir ama) bir uyarı mesajı verecektir. Aslında **enum**'ların birer tamsayı olduğunu – mümkün olduğunca – unutmak en iyisidir.

Biri veya Diğeri

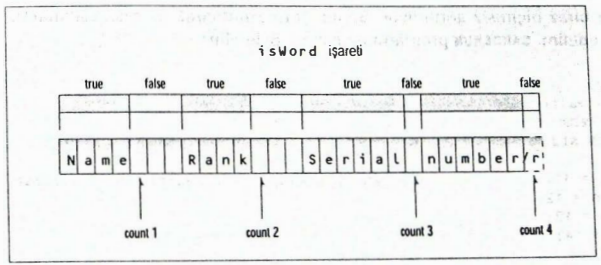
Sıradaki örneğimiz kullanıcı tarafından yazılan bir ifadedeki sözcükleri sayar. Önceki **CHCOUNT** örneğinden farklı olarak bu program, sözcük sayısını belirlemek için sadece boşluk karakterlerini saymaz. Program bunun yerine, Şekil 4.10'da gösterilen şekliyle, boşluk karakterinden farklı bir karakter dizisinin boşluk karakterine dönüştüğü yerleri sayar.

Bu şekilde sözcükler arasında birden fazla boşluk karakteri varsa sözcük sayısını yanlış hesaplamış olmazsınız. (Yalnız, program sekmeleri ve diğer boşlukları hâlâ hesaba katmaz.) **WDCOUNT**'un program listesi aşağıda görülüyor. Bu örnekte, numaralandırma sadece iki enümeratör içerir.

```
// wdcount.cpp
// enum'lari tanitir, bir ifadedeki sozcukleri sayar
#include <iostream>
using namespace std;
```



```
#include <conio.h>              //getche() icin

enum itsaWord {NO, YES };       //NO=0, YES=1

int main()
   {
   itsaWord isWord = NO;        //YES sozcuk icinde iken,
                                //NO bosluk uzerinde iken
   char ch = 'a';               //klavyeden okunan karakter
   int wordcount = 0;           //okunan sozcuk sayisi

   cout << "Enter a phrase:\n";
   do {
      ch = getche();            //karakteri al
      if(ch==' ' || ch=='\r') //bosluksa
         {
         if( isWord == YES )    //ve bir sozcuk uzerindeyse,
            {                   //o zaman sozcuk sonuna gelinmistir
            wordcount++;        //sozcugu say
            isWord = NO;        //isareti sifirla
            }
         }                      //aksi halde, bu
      else                      //normal bir karakterdir
         if(isWord == NO)       //eger sozcuk baslangiciysa,
            isWord = YES;       //isareti ayarla
      } while( ch != '\r' );  //Enter tusu ile cik
   cout << "\n---Word count is " << wordcount << "---\n";
   return 0;
   }
```



ŞEKİL 4.10: **WDCOUNT** programının çalışma mantığı.

Program, klavyeden karakterleri okurken bir **do** döngüsü içinde döner. Bir boşluk karakteri bulana kadar boşluk olmayan karakterleri atlar. Bu sırada bir sözcük saymış olur. Sonra bir karakter bulana kadar boşluk karakterlerini atlar ve tekrar boşluk karakteri bulana kadar karakterleri sayar. Bu işlemi yapması için programın, sözcüğün ortasında mı veya boşluk dizisinin ortasında mı olduğunu hatırlaması gerekir. Program bunu **isWord** isimli **enum** değişkeni ile hatırlar. Bu değişken **itsaWord** tipinde olacak şekilde tanımlanır. Bu tipin şu ifadeyle belirtilir:

```
enum itsaWord { NO, YES };
```

`itsaWord` tipindeki değişkenlerin sadece iki değeri olabilir: `NO` veya `YES`. Dikkat ederseniz liste `NO` ile başlar. Dolayısıyla, `NO`'ya – bir değerin false olduğunu işaret eden – 0 değeri verilir. (Aynı amaç için `bool` tipinde bir değişken de kullanabilirdik.)

Program çalışmaya başladığında `isWord` değişkenine `NO` değeri atanır. Program boşluktan farklı ilk karakterle karşılaşınca, sözcüğün ortasında olduğunu işaret etmesi için `isWord`'e `YES` değerini atar. Program bu değeri bir sonraki boşluğa kadar saklar; boşluğa gelindiğinde bu değişkenin değeri tekrar `NO`'ya çevrilir. Sahnenin gerisinde ise, `NO`'nun değeri 0 ve `YES`'in değeri 1'dir; ancak, bu gerçeği kullanmaktan kaçındık. `if(isWord == YES)` yerine `if(isWord)` ve `if(isWord == NO)` yerine `if(!isWord)` ifadelerini kullanabilirdik, fakat bu, iyi bir programlama tarzı olmazdı.

Ayrıca programdaki ikinci `if` ifadesi etrafına bir çift ekstra küme parantezi kullanmaya ihtiyaç duyduk, böylece `else` ilk `if` ile eşlenebilir.

`WDCOUNT`'taki gibi evet/hayır içeren bir durumda bir başka yaklaşım, `bool` tipinde bir değişken kullanmaktır. Duruma bağlı olarak bu, biraz daha az dolambaçlı bir yol olabilir.

Kartları Düzenlemek

İşte `enum` tipleri ile ilgili en son örneğimiz. Bu bölüm içinde daha önceden gördüğümüz `CARDS` programında, hatırlarsanız, iskambil kartlarının takımlarını simgelemek için `const int` tipinde bir grup sabit tanımlamıştık.

```
const int clubs = 0;
const int diamonds = 1;
const int hearts = 2;
const int spades = 3;
```

Bu tür bir liste biraz biçimsiz görünüyor. Bunun yerine numaralandırma kullanarak `CARDS` programını revize edelim. `CARDENUM` programının listesi şöyle olur:

```
// cardenum.cpp
// numaralandirmalar
#include <iostream>
using namespace std;

const int jack = 11;              //2'den 10'a kadar isimlendirilmemis sayilar
const int queen = 12;
const int king = 13;
const int ace = 14;

enum Suit {clubs, diamonds, hearts, spades };
////////////////////////////////////////////////////////////////
struct card
   {
   int number;    //2'den 10'a kadar, vale, kiz, papaz, as
   Suit suit;     //maca, karo, kupa, sinek
   };
////////////////////////////////////////////////////////////////
int main()
   {
   card temp, chosen, prize;          //kartlari tanimla
   int position;

   card card1 = {7, clubs };          //card1'e ilk deger ata
```



```
cout << "Card 1 is the seven of clubs\n";

card card2 = {jack, hearts };       //card2'ye ilk deger ata
cout << "Card 2 is the jack of hearts\n";

card card3 = {ace, spades };        //card3'e ilk deger ata
cout << "Card 3 is the ace of spades\n";

prize = card3;                      //bu karti kopyala, hatirlamak icin

cout << "I'm swapping card 1 and card 3\n";
temp = card3; card3 = card1; card1 = temp;

cout << "I'm swapping card 2 and card 3\n";
temp = card3; card3 = card2; card2 = temp;

cout << "I'm swapping card 1 and card 2\n";
temp = card2; card2 = card1; card1 = temp;

cout << "Now, where (1, 2, or 3)is the ace of spades? ";
cin >> position;

switch (position)
   {
   case 1: chosen = card1; break;
   case 2: chosen = card2; break;
   case 3: chosen = card3; break;
   }
if(chosen.number == prize.number &&     //kartlari karsilastir
      chosen.suit == prize.suit)
   cout << "That's right! You win!\n";
else
   cout << "Sorry. You lose.\n";
return 0;
}
```

`CARDS` programında kart takımları için kullanılan bir dizi tanımlama bu programda `enum` deklarasyonu ile değiştirilmiştir:

```
enum Suit ( clubs, diamonds, hearts, spades };
```

Bu, `const` değişkenleri kullanmaktan daha temiz bir yaklaşımdır. `suit`'in olası tüm değerlerini tam olarak biliyoruz. Bu yüzden, başka değer kullanmaya yönelik girişimler, mesela

```
card1.suit = 5;
```

derleyicinin uyarı mesajı vermesine neden olur.

Tamsayı Değerleri Açıkça Belirtmek

Bir `enum` deklarasyonunda ilk enümeratörün değerinin 0, ikincinin 1 vs olduğunu belirtmiştik. Başlangıç noktasının 0'dan farklı olduğunu belirtmek için eşittir işareti kullanılarak numaralandırma değiştirilebilir. Mesela, kart takımının 0 yerine 1'den başlamasını istiyorsanız, şöyle diyebilirsiniz:

```
enum Suit { clubs = 1, diamonds, hearts, spades };
```

Sonraki isimler de bu noktadan başlayarak değer alır; yani, **diamonds = 2, hearts = 3** ve **spades = 4** olur. Aslında eşittir işaretini herhangi bir enümeratöre belirli bir değer vermek için kullanabilirsiniz.

Kusursuz Değil

enum tipinin bir can sıkıcı özelliği, bu tipin C++ giriş/çıkış (I/O) ifadeleri tarafından fark edilmemesidir. Örnek olarak, sizce aşağıdaki kod parçası ekranda ne gösterir?

```
enum direction {north, south, east, west };
direction dir1 = south;
cout << dir1;
```

Çıktının **south** olduğunu mu düşündünüz? Bu hoş olurdu, fakat C++ I/O **enum** tipindeki değişkenleri tamsayı olarak ele aldığı için çıktı 1 olur.

Diğer Örnekler

Bu tipin muhtemel kullanımları hakkında bir fikir vermesi için işte size numaralandırılmış veri deklarasyonlarından birkaç örnek:

```
enum months {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };

enum switch {off, on };

enum meridian {am, pm };

enum chess {pawn, knight, bishop, rook, queen, king };

enum coins {penny, nickel, dime, quarter, half-dollar, dollar };
```

Daha sonraki programlarda başka örnekler de göreceğiz.

Özet

Bu bölümde iki konu işledik: Yapılar ve numaralandırmalar (enumerations). *Yapılar* C++'ın önemli bir bileşenidir, çünkü yapıların sözdizimi sınıflarınkinin aynısıdır. Aslında bir sınıf (en azından sözdizimsel açıdan) fonksiyon içeren bir yapıdan başka bir şey değildir. Yapılar genellikle tek bir bütün oluşturmak amacıyla birkaç tane veriyi gruplamak amacıyla kullanılırlar. Yapının tanımı, bu yapıyı oluşturan değişkenleri listeler. Sonraki diğer tanımlar, yapı değişkenleri için bellekte yer ayırır. Yapı değişkenleri bazı durumlarda bölünmez birimler olarak ele alınır (mesela, bir yapı değişkenini diğerine eşitlerken), fakat diğer durumlarda yapı değişkeninin üyeleri (genellikle nokta operatörü kullanılarak) tek tek erişilebilir.

Numaralandırma, sabit uzunluktaki değerler listesi ile sınırlı olan ve kullanıcı tarafından tanımlanan bir tiptir. Tip deklarasyonu, tipe bir isim verir ve tipin alabileceği değerleri belirtir. Bu değerlere *enümeratör* denir. Daha sonraki tanımlamalar, bu tipte değişkenler oluşturur. Derleyici kendi içinde, numaralandırma değişkenlerini tamsayı olarak ele alır.

Yapılar numaralandırmalarla karıştırılmamalıdır. Yapılar, çok çeşitli verileri tek bir bütün içinde güçlü ve esnek olarak gruplama yollarından biridir. Numaralandırma ise, tip deklarasyonunda listelenen (numaralandırılan) sabit sayıdaki değerleri, değer olarak alabilen değişkenlerin tanımlanmasına imkan verir.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Bir yapı aşağıdakilerden hangisini bir araya getirmek için kullanılır?
   a. Aynı veri tipine ait parçaları.
   b. Birbiriyle ilgili verileri.
   c. Kullanıcı tarafından isimlendirilmiş tamsayıları.
   d. Değişkenleri.
2. Doğru/Yanlış: Bir yapı ile bir sınıf aynı sözdizimini kullanırlar.
3. Bir yapının kapanış küme parantezinin peşinden _____ gelir.
4. Her birinin **int** tipinde üç değişken içeren bir yapı spesifikasyonu yazın. Değişkenlerin isimleri **hrs**, **min** ve **secs** olsun. Bu yapıya **time** ismini verin.
5. Doğru/Yanlış: Bir yapı tanımı, bir değişken için bellekte yer ayırır.
6. Yapının bir üyesine erişirken, nokta operatörünün solunda kalan, aşağıdakilerden hangisinin ismidir?
   a. Yapının üyesinin.
   b. Yapının etiketinin.
   c. Yapı değişkeninin.
   d. **struct** anahtar kelimesinin.
7. **time2** yapı değişkeninin **hrs** üyesine 11 değerini atayan bir ifade yazın.
8. **struct time** tipinde tanımlanmış üç değişkeniniz varsa ve bu yapı üç tane **int** tipinde üye içeriyorsa, bu değişkenlerin tümü toplam kaç byte bellek gerektirir?
9. Soru 4'te tanımlandığı gibi **struct time** tipinde bir değişken olan **time1**'in üyelerine şu başlangıç değerlerini atayan bir tanım yazın. **hrs = 11, mins = 10, secs = 59**.
10. Doğru/Yanlış: Aynı tipte olmaları şartıyla, bir yapı değişkenini diğerine atayabilirsiniz.
11. **temp** değişkenini, **fido** değişkeninin **dogs** üyesinin **paw** üyesine eşitleyen bir ifade yazın.
12. Bir numaralandırma aşağıdakilerden hangisini bir araya getirir?
    a. Değişik veri tiplerindeki parçaları.
    b. İlgili veri değişkenlerini.
    c. Kullanıcı tarafından isimlendirilen tamsayıları.
    d. Sabit değerleri.
13. **players** isminde ve B1, B2, SS, B3, RF, CF, LF, P ve C değerlerini içeren bir numaralandırma deklarasyonu ifadesi yazın.
14. Soru 13'te tanımlanan **players** tipinde iki değişkenin, **joe** ve **tom** olduğunu varsayarak, bu değişkenlere sırasıyla LF ve P değerlerini atayın.
15. Soru 13 ve 14'teki ifadelerin doğru olduğunu varsayarak, aşağıdakilerden hangileri geçerli hangileri geçersizdir:
    a. **joe= QB;**
    b. **tom= SS;**
    c. **LF= tom;**
    d. **Difference= joe - tom;**

16. Bir `enum` tipinin ilk üç enumeratörü normal olarak _____, _____ ve _____ ile temsil edilir.
17. Enumeratörleri `obsolete`, `single` ve `album` olan `speeds` adlı bir numaralandırma tanımlayan bir ifade yazın. Bu üç isme tamsayı olarak 78, 45 ve 33 değerlerini atayın.
18. Şu ifadenin:

```
enum isWord {NO, YES};
```

aşağıdaki ifadeden neden daha iyi olduğunu açıklayın.

```
enum isWord {YES, NO};
```

Alıştırmalar

Yıldızla işaretli alıştırmaların cevaplarını Ek G'de bulabilirsiniz.

1. (212) 767 8900 gibi bir telefon numarasını üç parça halinde düşünmek mümkündür: alan kodu (212), santral kodu (767) ve numara (8900). Bir telefon numarasını bu şekilde ayrı ayrı üç parça değerlendiren bir yapı kullanan bir program yazın. Bu yapıya `phone` adını verin. Sonra `phone` tipinden iki yapı değişkeni tanımlayın. Bunlardan birine başlangıç değeri atayın, diğerini kullanıcının girmesini sağlayın. Her iki numarayı da ekrana yazın. Programa veri girişi ve programın çıktısı şu şekilde olabilir:*

```
Enter your area code, exchange, and number: 415 555 1212
My number is (212) 767-8900
Your number is (415) 555-1212
```

2. İki boyutlu bir düzlem üzerindeki bir nokta iki sayı ile simgelenebilir: x ve y koordinatı. Örneğin, (4,5) düşey eksenden 4 birim sağda, yatay eksenin 5 birim yukarısında yer alan bir noktayı simgeler. İki noktanın toplamı yeni bir nokta olarak tanımlanabilir. Öyle ki, bu yeni noktanın x koordinatı, iki noktanın x koordinatlarının toplamı; y koordinatı y koordinatlarının toplamıdır.
   Bir noktayı modellemek için `point` isminde bir yapı kullanan bir program yazın. Üç nokta tanımlayın. Bu noktalardan iki tanesinin değerlerini kullanıcının girmesini isteyin. Üçüncü noktanın değeri diğer ikisinin toplamı olsun. Sonra yeni noktanın değerini ekranda gösterin. Programla etkileşim şu şekilde görünebilir:

```
Enter coordinates for p1: 3 4
Enter coordinates for p2: 5 7
Coordinates of p1+p2 are: 8, 11
```

3. Bir odanın hacmini modellemek için `Distance` tipinde (ENGLSTRC örneğindeki gibi) üç değişken kullanan `Volume` isimli bir yapı geliştirin. `Volume` tipinde bir değişkene belirli bir boyutu başlangıç değeri olarak atayın, bunun simgelediği hacmi hesaplayın ve sonucu ekrana yazın. Hacmi hesaplamak için, `Distance` değişkeninin her boyutunu `float` tipinde başka bir değişkene atayın. `float` tipindeki bu değişken ayak cinsinden ondalık sayılarla simgelenir. Son olarak bu üç sayıyı birbiriyle çarpın.
4. İki üyesi olan `employee` adında bir yapın geliştirin. Bu yapının üyeleri şunlar olmalı: Çalışanın (personelin) numarası (`int` tipinde) ve çalışanın tazminatı (dolar olarak; `float` tipinde). Kullanıcıdan üç çalışan için bu bilgileri doldurmasını isteyin, bu bilgileri `struct employee` tipinde üç değişken içinde saklayın ve her çalışan için karşılık gelen bilgileri ekranda gösterin.

5. Üç tane üyesi olan `date` tipinde bir yapı tanımlayın. Üyeler şunlar olmalı: Ay, ayın günü ve yıl, hepsi `int` tipinde. (Ya da, tercih ederseniz gün-ay-yıl numaralandırmasını da kullanabilirsiniz.) Kullanıcının 12/31/2001 formatında bir tarih girmesini sağlayın, bunu `struct date` tipinde bir değişkende saklayın, sonra değişkenden değerleri alın ve aynı formatta ekrana yazın.
6. Daha önce, C++ I/O ifadelerinin numaralandırılmış veri tipini otomatik olarak anlamadıklarını belirtmiştik. (`>>`) ve (`<<`) operatörleri, bu tür değişkenleri basit birer tamsayı olarak ele alır. Bir `switch` ifadesi kullanarak bu kısıtlamanın üstesinden gelebilirsiniz. `switch` ifadesi, numaralandırılmış değişkenleri kullanıcının ifade ettiği şekli ile bu değişkenlerin asıl değerleri arasında çevirmek için kullanılacaktır. Örneğin, bir kurumdaki çalışanların tipini belirtmek için değerler içeren bir numaralandırılmış tip hayal edin:

```
enum etype {laborer, secretary, manager, accountant, executive, researcher };
```

   Kullanıcının, çalışanın tipini, baş harfini girerek belirtmesine imkan veren bir program yazın (`'l'`, `'s'`, `'m'` vs.) Program, seçilen tipi, `enum etype` tipinde bir değişkende değer olarak saklamalı, son olarak bu tipe karşılık gelen kelimenin tamamını ekranda göstermeli.

```
Enter employee type (first letter only)
laborer, secretary, manager,
accountant, executive, researcher): a
Employee type is accountant.
```

7. Alıştırma 4'teki `employee` sınıfına `enum etype` tipinde bir değişken (Alıştırma 6'ya bakın) ve `struct date` tipinde (Alıştırma 5'e bakın) başka bir değişken ekleyin. Programı, kullanıcı üç çalışanın her biri için dört parça bilgi girecek şekilde yeniden düzenleyin: Çalışanın numarası, çalışanın tazminatı, çalışanın tipi ve işe giriş tarihi. Program bu bilgileri `employee` tipindeki üç değişkende saklamalı ve bu değişkenlerin içeriğini ekranda göstermelidir.
8. "C++ Programlama Temelleri" adlı Bölüm 2'de, Alıştırma 9'daki kesirli toplama yapan programla başlayın. Bu program, toplama yapmadan önce iki kesrin payını ve paydasını saklar. Ayrıca, yine bir kesirli sayı olan sonucu da saklayabilir. Programı şu şekilde değiştirin: Tüm kesirli sayılar `struct fraction` tipinde değişkenlerde saklansın. Bu yapının iki üyesi olsun: Kesirli sayının payı ve paydası (her ikisi de `int` tipinde). Kesirli sayılarla bağlantılı tüm veriler bu tip yapılar içinde saklanmalıdır.
9. `time` olarak adlandırılan bir yapı oluşturun. Bu yapının, her biri `int` tipinde üç üyesi, `hours`, `minutes` ve `seconds` olsun. Kullanıcının saat, dakika ve saniye cinsinden zamanı girmesini isteyen bir program yazın. Girdi 12:59:59 formatında olabilir veya her sayı kullanıcıdan tek tek alınabilir ("Saati giriniz:", "Dakikayı giriniz:" vs.). Program daha sonra zamanı `struct time` tipinde bir değişkende saklamalı ve son olarak, girilen süre ile simgelenen toplam saniye sayısını ekrana yazmalıdır:

```
long totalsecs = t1.hours*3600 + t1.minutes*60 + t1.seconds
```

10. "Döngüler ve Kararlar" adlı Bölüm 3'deki Alıştırma 8 ve 11'de bahsedilen eski stil İngiliz sistemine göre para miktarlarını saklayan `sterling` isminde bir yapı oluşturun. Bu yapının üyelerine `pounds`, `shillings` ve `pence` ismini verebilirsiniz. Üyelerin hepsi `int` tipinde olmalı. Program kullanıcıdan yeni stil ondalık pound biçiminde (`double` tipinde) para miktarını girmesini istemeli, bunu eski stile çevirmeli, `struct sterling` tipinde bir değişkende saklamalı ve ekranda bu miktarı pound-shilling-pence biçiminde göstermelidir.
11. Alıştırma 9'daki `time` yapısını kullanarak şöyle bir program yazın: Program, kullanıcıdan 12:59:59 formatında iki `time` değeri alsın. Bunları `struct time` değişkenlerinde saklasın. Her birini saniyeye çevirsin (`int` tipinde). Bu değerleri toplasın. Sonucu tekrar saat-dakika-saniye formatına çevirsin. Bu sonucu bir `time` yapısı içinde saklasın ve son olarak da sonucu 12:59:59 biçiminde ekranda göstersin.
12. Bölüm 13'de Alıştırma 12'de anlatılan kesirli sayılarla işlem yapabilen dört fonksiyonlu hesap makinesini şu şekilde geliştirin: Her kesir dahili olarak, bu bölümde Alıştırma 8'de bahsedilen `struct fraction` tipinde bir değişkende saklansın.

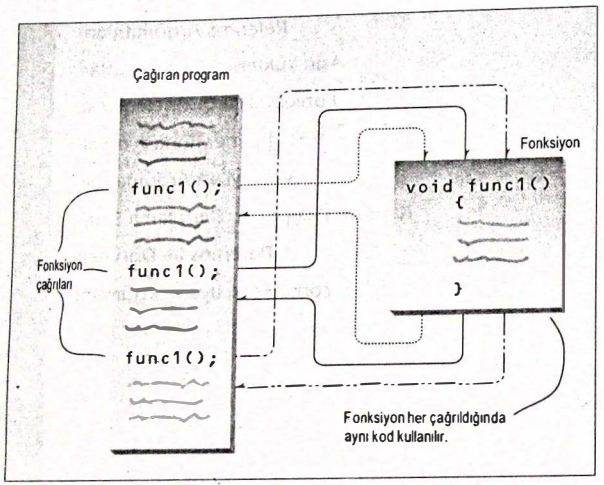
BEŞİNCİ BÖLÜM

FONKSİYONLAR

Bir fonksiyon birkaç tane program ifadesini tek bir birim içinde gruplar ve buna bir isim verir. Bu birim daha sonra programın diğer bölümleri tarafından çağrılabilir.

Fonksiyonları kullanmanın en önemli sebebi, bir programın kavramsal olarak düzenlenmesine yardımcı olmaktır. "Genel Görünüm" başlıklı Bölüm 1'de ele alındığı gibi, bir programı fonksiyonlara bölmek yapısal programlamanın en temel prensiplerinden biridir. (Bununla birlikte, nesne yönelimli programlama, programları organize etmek için daha güçlü ilave yöntemler sunar.)

Fonksiyon kullanmanın diğer bir sebebi (ve uzun zaman önce icat edilmelerindeki neden), fonksiyonların programın boyutunu küçültmeleridir. Bir program içinde birden fazla yerde ortaya çıkan herhangi bir komut dizisi fonksiyon içine alınmaya adaydır. Fonksiyon, program boyunca pek çok kere çalıştırılıyor olsa bile, fonksiyonun kodu bellekte tek bir yerde saklanır. Şekil 5.1 bir fonksiyonun, programın değişik bölümleri tarafından nasıl çağrıldığını gösterir.



ŞEKİL 5.1: Bir fonksiyona kontrol akışı.

C++'taki (ve C'deki) fonksiyonlar diğer çeşitli dillerdeki alt rutin ve prosedürlerle aynıdır.

Basit Fonksiyonlar

İlk örneğimizde, amacı ekrana 45 asteriksten oluşan bir satır basmak olan basit bir fonksiyon gösterilmiştir. Örnek program bir tablo üretir ve asteriklerden oluşan satırlar, tabloyu daha okunaklı hale getirmek için kullanılır. **TABLE** programının listesi şöyledir:

```
// table.cpp
// basit bir fonksiyon
#include <iostream>
using namespace std;
```



```
void starline();                        //fonksiyon tanimi
                                        //   (prototip)
int main()
   {
   starline();                          //fonksiyona cagri
   cout << "Data type
Range" << endl;
   starline();                          //fonksiyona cagri
   cout << "char
        -128 to 127" << endl
        << "short
        -32,768 to 32,767" << endl
        <<" int
        System dependent" << endl
        <<" long
        -2,147,483,648 to 2,147,483,647" << endl;
   starline();                          //fonksiyona cagri
   return 0;
   }
//--------------------------------------------------------------
// starline()
// fonksiyon tanimi
void starline()                         //fonksiyon deklaratoru
   {
   for(int j=0; j<45; j++)              //fonksiyon govdesi
      cout << '*';
   cout << endl;
   }
```

Programın çıktısı şöyle görünür:

```
*********************************************
Data type   Range
*********************************************
char       -128 to 127
short     -32,768 to 32,767
int       System dependent
long    -2,147,483,648 to 2,147,483,647
*********************************************
```

Bu program iki fonksiyondan oluşur: **main()** ve **starline()**. Şimdiden, yalnızca **main()** kullanan birçok program gördünüz. Bir fonksiyonu bir programa dahil etmek için başka hangi bileşenler gerekli? Üç tane bileşen vardır: Fonksiyonun *deklarasyonu*, fonksiyona yapılan *çağrılar* ve fonksiyonun *tanımı*.

Fonksiyonun Deklare Edilmesi

Nasıl ki, bir değişkeni derleyiciye bildirmeden önce kullanamıyorsunuz, bir fonksiyonu da derleyiciye bildirmeden kullanamazsınız. Bunu yapmanın iki yolu vardır. Burada göstereceğimiz yaklaşım, fonksiyonu çağrılmadan önce *deklare etmektir*. (Diğer yaklaşım, fonksiyonu çağrılmadan önce *tanımlamaktır*; bunu, bundan sonra göreceğiz.) **TABLE** programında, **starline()** fonksiyonu şu satırda deklare edilir:

```
void starline();
```

Bu deklarasyon derleyiciye, daha sonraki bir noktada **starline** isminde bir fonksiyon sunmayı planladığımızı anlatır. **void** anahtar kelimesi fonksiyonun döndürdüğü bir değer olmadığını, boş parantezler ise argüman almadığını belirtir. (Fonksiyonun argüman almadığını belirtmek için C'de sıkça kullanıldığı şekliyle, parantezlerin içinde **void** anahtar kelimesini

kullanabilirsiniz. Fakat parantezlerin içini boş bırakmak C++'ta daha yaygın olan bir uygulama şeklidir.) Argümanlar ve fonksiyonların döndürdüğü değerlerle ilgili olarak, birazdan daha ayrıntılı konuşacağız.

Fonksiyon deklarasyonunun noktalı virgül ile sona ediğine dikkat edin. Bu kendi başına komple bir ifadedir.

Fonksiyon deklarasyonlarına ayrıca *prototip* de denir, çünkü fonksiyon deklarasyonları fonksiyonların bir modelini veya taslağını sunar. Fonksiyon deklarasyonları derleyiciye, "Programın içinde buna benzer bir fonksiyon birazdan ortaya çıkacak; bu nedenle, fonksiyonun kendisini görmeden onunla ilgili referansları görürsen, bir problem yok," derler. Deklarasyonda kullanılan bilgiler (fonksiyondan dönen değerin tipi ve argümanların tipi ve sayısı) kimi zaman fonksiyonun *imzası* (*signature*) olarak adlandırılır.

Fonksiyonu Çağırmak

Fonksiyon `main()`'den üç kez *çağrılır*(veya *çalıştırılıyor*). Bu üç çağrının her biri şu şekilde görünür:

```
starline();
```

Fonksiyonu çağırmak için yapmamız gereken işin hepsi bu kadar: Fonksiyon ismi ve peşinden gelen bir çift parantez. Çağrının söz dizimi, fonksiyon deklarasyonunda kullanılan ifade ile aynıdır; tek bir fark vardır, çağrıda fonksiyondan dönen değer yer almaz. Çağrı, noktalı virgül ile sona erer. Çağrı ifadesini çalıştırmak fonksiyonu çalıştırmak demektir; yani kontrol fonksiyona geçer, fonksiyonun tanımındaki ifadeler (bir dakika içinde bunları göreceğiz) çalıştırılır, sonra kontrol fonksiyon çağrısını takip eden ifadeye geri döner.

Fonksiyonun Tanımı

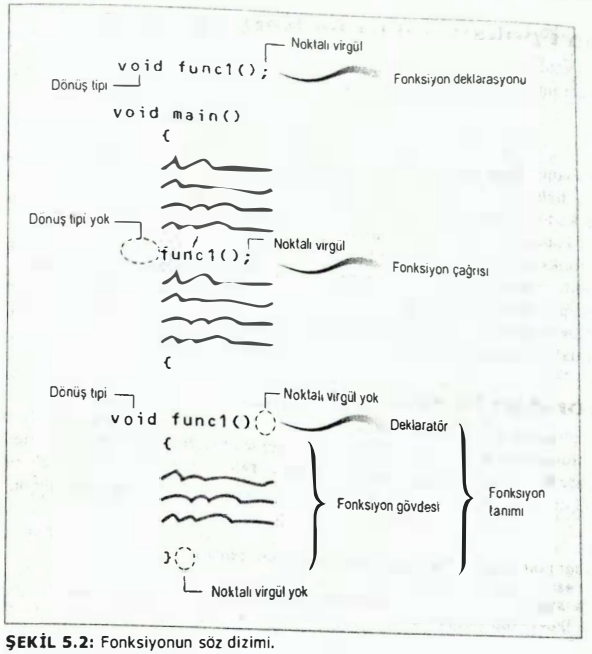
Nihayet fonksiyonun kendisine gelebildik. Buna fonksiyonun *tanımı* denir. Tanım, fonksiyonun asıl kodunu içerir. `starline()`'ın tanımı şöyledir:

```
void starline()                 //deklarator
   {
   for(int j=0; j<45; j++)      //fonksiyonun govdesi
   cout << '*';
   cout << endl;
   }
```

Tanım, *deklaratör* olarak adlandırılan bir satırdan ve bunu takip eden fonksiyon gövdesinden ibarettir. Fonksiyon gövdesi, küme parantezleriyle sınırlandırılmış, fonksiyonu oluşturan ifadelerden meydana gelir.

Deklaratör, deklarasyonla uyuşmak zorundadır: Aynı fonksiyon ismini kullanmalı, aynı tipte ve aynı sırada argüman içermeli (eğer argümanı varsa) ve döndürdüğü değer aynı tipte olmalıdır.

Deklaratörün noktalı virgül ile sona *ermediğine* dikkat edin. Şekil 5.2 fonksiyon deklarasyonunun, fonksiyon çağrısının ve fonksiyon tanımının söz dizimini gösterir.



ŞEKİL 5.2: Fonksiyonun söz dizimi.

Bir fonksiyon çağrıldığında kontrol, fonksiyon gövdesindeki ilk ifadeye aktarılır. Sonra fonksiyon gövdesindeki diğer ifadeler çalıştırılır. Kapanan parantez ile karşılaşınca kontrol, fonksiyonu çağıran programa geri döner.

Tablo 5.1 değişik fonksiyon bileşenlerini özetler.

TABLO 5.1: Fonksiyon Bileşenleri

Bileşen	Amaç	Örnek
Deklarasyon (Prototip)	Fonksiyon ismini, argüman tiplerini, ve fonksiyondan dönen değeri belirtir. Derleyiciyi (ve programcıyı) birazdan bir fonksiyonla karşılaşacağı konusunda uyarır.	`void func();`
Çağrı	Fonksiyonun çalıştırılmasını sağlar.	`func();`

Tanım	Fonksiyonun kendisidir. Fonksiyonu oluşturan kod satırlarını içerir.	`void func();`
		`{`
		`  //kod satirlari`

Deklaratör	Tanımın ilk satırıdır.	`void func();`

Kütüphane Fonksiyonları ile Karşılaştırma

Şimdiye kadar bazı kütüphane fonksiyonlarını kullanmayı öğrendik. Program kodunun içine kütüphane fonksiyonlarının çağrılarını dahil ettik. Örneğin,

```
ch= getche();
```

Peki, bu kütüphane fonksiyonunun deklarasyonu ve tanımı nerededir? Deklarasyonu, programın başında belirtilen başlık dosyası içindedir (**getche()** için **CONIO.H**). Tanımı ise (derlenip, makine koduna çevrilmiş olarak), programınız çalıştığında programınıza otomatik olarak bağlanan bir kütüphane dosyası içindedir.

Bir kütüphane fonksiyonu kullandığımız zaman bu fonksiyonun deklarasyonunu veya tanımını yazmak zorunda değiliz. Ancak kendi fonksiyonlarımızı yazarken deklarasyon ve tanım, **TABLE** örneğinde de gösterdiğimiz gibi, kaynak dosyamızın bir parçasıdır. (Bölüm 13'te "Birden Fazla Dosya Kullanan Programlar" başlığı altında göreceğimiz gibi, birden fazla dosya kullanan programlarda işler daha da karmaşıklaşır.)

Deklarasyondan Kurtulmak

Bir fonksiyonu bir program içine yerleştirmek için ikinci bir yaklaşım, fonksiyon deklarasyonunu kaldırmak ve fonksiyonun tanımını (fonksiyonun kendisini) program listesi içinde fonksiyona yapılan ilk çağrıdan önce yerleştirmektir. Örneğin, **TABLE** programını **TABLE2**'i üretmek için yeniden yazabiliriz. **TABLE2**'de ilk önce **starline()** fonksiyonunun tanımı görülüyor.

```
// table2.cpp
// fonksiyon cagrilarindan once yer alan fonksiyon tanimi
#include <iostream>
using namespace std;                    //burada fonksiyon deklarasyonu yok
//-------------------------------------------------------------
//starline()                            //fonksiyon tanimi
void starline()
   {
   for(int j=0; j<45; j++)
      cout << '*';
   cout << endl;
   }
//-------------------------------------------------------------
int main()                              //main() fonksiyondan sonra gelir
   {
   starline();                          //fonksiyon cagrisi
   cout << "Data type  Range " << endl;
   starline();                          //fonksiyon cagrisi
   cout << "char       -128 to 127" << endl
        << "short      -32,768 to 32,767" << endl
        << "int        System dependent" << endl
        << "long       -2,147,483,648 to 2,147,483,647" << endl;
   starline();                          //fonksiyon cagrisi
   return 0;
   }
```

Deklarasyonu ortadan kaldırdığından dolayı, kısa programlar için bu yaklaşım daha basit olabilir; fakat daha az esnektir. Birkaç taneden fazla fonksiyon olduğunda, bu tekniği kullanmak için programcının fonksiyonları düzenleme işine hatırı sayılır ölçüde emek vermesi gerekir. Öyle ki, her fonksiyon kendisini çağıran diğer fonksiyonlardan önce yer almalıdır. Kimi zaman bu imkansızdır. Ayrıca birçok programcı, **main()**'i program listesinde en başa yerleştirmeyi tercih eder, çünkü program çalışmaya bu noktadan başlar. Genel olarak biz, deklarasyon kullanarak ve listeye **main()** ile başlayarak, birinci yaklaşıma bağlı kalacağız.

Fonksiyonlara Argüman Aktarmak

Argüman, bir programdan bir fonksiyona aktarılan veri parçasıdır (mesela, bir **int** değeri). Argümanlar bir fonksiyonun değişik değerlerle çalışmasına imkan verir, hatta onu çağıran programın isteklerine bağlı olarak değişik işler bile yapabilir.

Sabitleri Aktarmak

Örnek olarak, son örnekteki **starline()** fonksiyonunun çok katı olduğuna karar verdiğimizi varsayalım. Her zaman 45 asteriks basan bir fonksiyon yerine, herhangi bir karakteri herhangi sayıda basacak bir fonksiyon istiyoruz.

İşte sadece böyle bir fonksiyon içeren bir program, **TABLEARG**. Ekrana basılacak karakteri ve kaç kez basılacağını fonksiyona argüman olarak aktarırız.

```
// tablearg.cpp
// fonksiyon argumanlari
#include <iostream>
using namespace std;
void repchar(char, int);                //fonksiyon deklarasyonu

int main()
   {
   repchar('-', 43);                    //fonksiyon cagrisi
   cout << "Data type  Range" << endl;
   repchar('=', 23);                    //fonksiyon cagrisi
   cout << "char       -128 to 127" << endl
        << "short      -32,768 to 32,767" << endl
        << "int        System dependent" << endl
        << "double     -2,147,483,648 to 2,147,483,647" << endl;
   repchar('-', 43);                    //fonksiyon cagrisi
   return 0;
   }
//-------------------------------------------------------------
//repchar()
//fonksiyon tanimi
void repchar(char ch, int n)            //fonksiyon deklaratoru
   {
   for(int j=0; j<n; j++)               //fonksiyon govdesi
      cout << ch;
   cout << endl;
   }
```

Yeni fonksiyonun ismi **repchar()**. Bu fonksiyonun deklarasyonu şöyle görünür:

```
void repchar(char, int);                //deklarasyonda veri tipleri belirtiliyor
```

Parantez içindeki parçalar **repchar()**'a gönderilecek argümanların veri tipleridir: **char** ve **int**.

Fonksiyon çağrısında, spesifik değerler – bu örnekte sabitler – parantez içinde uygun yerlere yerleştirilir:

```
repchar('-', 43); //fonksiyon cagrisi gercek degerleri acikca belirtir
```

Bu ifade, **repchar()**'a 43 tireden oluşan bir satır basması için talimat verir. Çağrıda yer alan değerlerin tipleri, deklarasyonda belirtilen tipler olmalıdır: Birinci argüman olan '-', **char** tipinde; ikinci argüman olan 43, **int** tipinde olmalıdır. Deklarasyonda ve tanımda yer alan tipler de, ayrıca uyuşmalıdır.

repchar()'a yapılan ikinci çağrı

```
repchar('=', 23);
```

fonksiyona, 23 eşittir işaretinden oluşan bir satır basmasını bildirir. Üçüncü çağrı ile yine 43 tire basılır. **TABLEARG**'ın çıktısı şöyle olur:

```
-----------------------------------------
Data type   Range
==========================
char       -128 to 127
short      -32,768 to 32,767
int       System dependent
long       -2,147,483,648 to 2,147,483,647
-----------------------------------------
```

Fonksiyonu çağıran program, fonksiyona *argümanları*, yani, '-' ve 43'ü tedarik eder. Fonksiyon içinde bu argümanları saklamak için kullanılan değişkenlere *parametre* denir. **repchar()**'da bu parametreler **ch** ve **n**. (Birçok programcı argüman ve parametre terimlerini birbirinin yerine kullanıyorlar; buna dikkat etmeliyiz.) Fonksiyon tanımındaki deklaratör bu parametrelerin hem veri tiplerini hem de isimlerini açıkça belirtir:

```
void repchar(char ch, int n);    //deklarator parametre isimlerini ve
                                 //tiplerini acikca belirtir
```

Bu parametre isimleri, **ch** ve **n**, fonksiyon içinde sanki normal değişkenlermiş gibi kullanılır. Bunları deklaratörün içine yerleştirmek aşağıdaki gibi ifadeler kullanarak tanımlaya denktir:

```
char ch;
int n;
```

Fonksiyon çağrıldığında, fonksiyonun parametrelerine, fonksiyonu çağıran programdan gelen değerler otomatik olarak atanır.

Değişkenleri Aktarmak

TABLEARG örneğinde, argümanlar birer sabitti: '-', 43 vs. Şimdi, sabitler yerine değişkenlerin argüman olarak aktarıldığı bir örneğe bakalım. **VARGARG** isimli bu program, **TABLEARG**'daki **repchar()** fonksiyonunun aynını içerir, fakat karakteri ve karakterin kaç kez tekrarlanması gerektiğini kullanıcının belirlemesine izin verir.


```
// vararg.cpp
// degisken argumanlar
#include <iostream>
using namespace std;
void repchar(char,int);              //fonksiyon deklarasyonu
int main()
   {
   char chin;
   int nin;

   cout << "Enter a character: ";
   cin >> chin;
   cout << "Enter number of times to repeat it: ";
   cin >> nin;
   repchar(chin, nin);
   return 0;
   }
//--------------------------------------------------------------
// repchar()
// fonksiyon tanimi
void repchar(char ch, int n)     //fonksiyon deklaratoru
   {
   for(int j=0; j<n; j++)        //fonksiyon govdesi
      cout << ch;
   cout << endl;
   }
```

VARGARG ile örnek bir etkileşim şöyle olabilir:

```
Enter a character: +
Enter number of times to repeat it: 20
++++++++++++++++++++
```

Bu örnekte, **main()**'deki **chin** ve **nin**, **repchar()**'a aktarılan argümanlar olarak kullanılır.

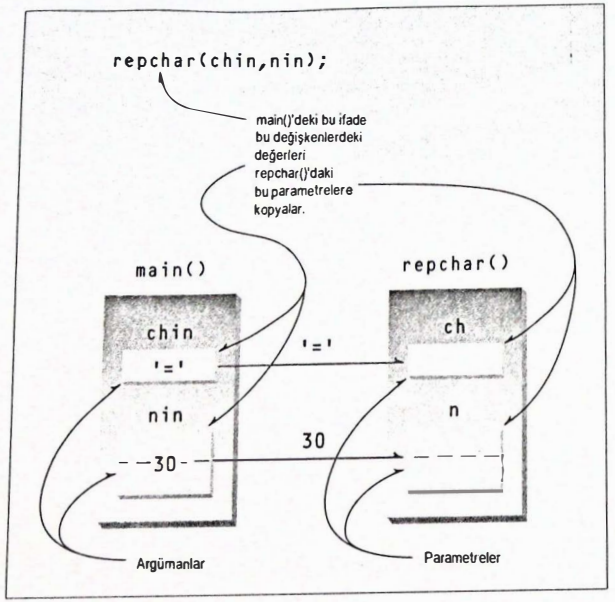
```
repchar(chin, nin);  //fonksiyon cagrisi
```

Argüman olarak kullanılan değişkenlerin veri tipleri, tıpkı sabitlerde olduğu gibi, fonksiyon deklarasyonunda ve tanımında belirtilen tiplerle eşlenmelidir. Yani, **chin** bir **char** olmalıdır, **nin** ise bir **int**.

Argümanı Değer Olarak Aktarmak

VARGARG'da **chin** ve **nin**'in sahip olduğu belli değerler, fonksiyon çağrısı yapıldığı zaman fonksiyona aktarılır. Sabitlerin aktarılmasında olduğu gibi fonksiyon, bu değişken argümanlarının değerlerini saklamak için yeni değişkenler tanımlar. Fonksiyon bu yeni değişkenlere deklaratörde belirtilen parametre isimlerini ve veri tiplerini verir: **char** tipinde **ch** ve **int** tipinde **n**. Bunlara daha sonra fonksiyon gövdesindeki ifadeler tarafından diğer değişkenler gibi erişilir.

Argümanların bu şekilde aktarılmasına, yani, fonksiyonun kendisine gelen argümanların kopyalarını çıkardığı bu aktarım şekline *değer olarak aktarmak* denir. Bu bölüm içinde, daha sonra, bir başka yaklaşım olan *referans olarak aktarmayı* da inceleyeceğiz. Şekil 5.3 argümanlar değer olarak aktarıldığında, fonksiyon içinde yeni değişkenlerin nasıl ortaya çıktığını gösterir.



ŞEKİL 5.3: Değer olarak aktarmak.

Argüman Olarak Yapılar

Yapıların tümü fonksiyonlara argüman olarak aktarılabilir. İki örnek göstereceğiz. Biri, **Distance** yapısını, diğeri ise bir grafik şeklini simgeleyen bir yapıyı kullanır.

Bir Distance Yapısını Aktarmak

Bu örnek, **Distance** tipinde bir yapıyı argüman olarak kullanan bir fonksiyona dikkat çekiyor. **Distance**, "Yapılar" adlı Bölüm 4 içindeki birçok programda karşımıza çıkan yapı tipinin aynısıdır. İşte **ENGLDISP** programının listesi:

```
// engldisp.cpp
// arguman olarak yapilarin aktarilmasi
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
struct Distance                  //Ingiliz sistemine gore uzaklik
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////
```



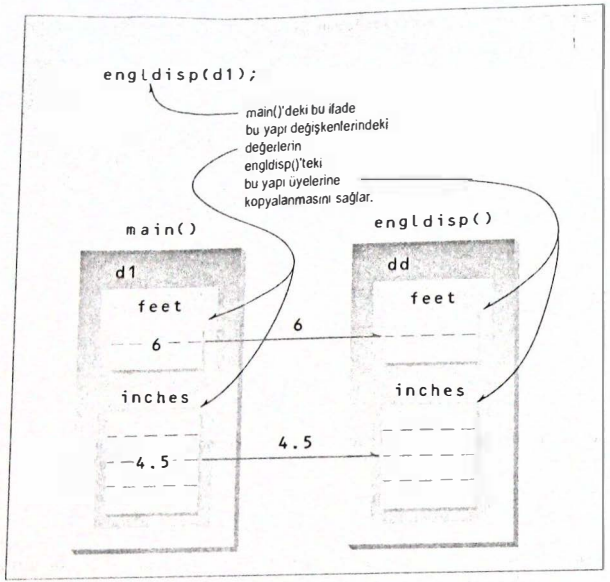
```
void engldisp(Distance);          //deklarasyon

int main()
   {
   Distance d1, d2;               //iki uzaklik tanimla

                                  //d1 uzakligini kullanicidan al
   cout << "Enter feet: "; cin >> d1.feet;
   cout << "Enter inches: "; cin >> d1.inches;

                                  //d2 uzakligini kullanicidan al
   cout << "\nEnter feet: "; cin >> d2.feet;
   cout << "Enter inches: "; cin >> d2.inches;

   cout << "\nd1 = ";
   engldisp(d1);                  //1. uzakligi ekranda goruntule
   cout << "\nd2 = ";
   engldisp(d2);                  //2. uzakligi ekranda goruntule
   cout << endl;
   return 0;
   }
//--------------------------------------------------------------
// engldisp()
// feet ve inc cinsinden Distance tipindeki yapiyi ekranda goruntule
void engldisp( Distance dd )      //Distance tipinde dd isimli parametre
   {
   cout << dd.feet << "\'-" << dd.inches << "\"";
   }
```

Bu programın **main()** bölümü kullanıcıdan ayak ve inç biçiminde iki uzaklık değeri alır ve bu değerleri **d1** ve **d2** isimli iki yapının içine yerleştirir. Program daha sonra, **Distance** yapısını argüman olarak alan **engldisp()** isminde bir fonksiyonu çağırır. Bu fonksiyonun amacı, kendisine aktarılan uzaklığı standart biçimde, mesela 10'-2.25" gibi, ekranda görüntülemektir. Programla etkileşim şöyle olabilir:

```
Enter feet: 6
Enter inches: 4
Enter feet: 5
Enter inches: 4.25

d1 = 6'-4"
d2 = 5'-4.25"
```

main() içindeki fonksiyon deklarasyonları ve fonksiyon çağrıları, ayrıca fonksiyon gövdesindeki deklaratör, yapı değişkenlerini, argüman olarak kullanılan diğer değişkenlerle aynı biçimde ele alırlar. Bu örnekte fonksiyon tarafından ele alınan değişken **char** veya **int** gibi temel bir tip yerine **Distance** tipindedir.

main() içinde **engldisp()** fonksiyonu iki kez çağrılır. İlk çağrı ile **d1** yapısı, ikinci ile **d2** yapısı fonksiyona aktarılır. **engldisp()** fonksiyonu, **dd** ismini verdiği **Distance** tipinde bir yapıyı parametre olarak kullanır. Basit değişkenlerle olduğu gibi, **main()**'den aktarılan yapının değeri, parametre olarak kullanılan bu yapı değişkenine otomatik olarak atanır. **engldisp()**'in içindeki ifadeler, daha sonra **dd**'nin üyelerine, **dd.feet** ve **dd.inches** deyimleri ile her zamanki gibi erişirler. Şekil 5.4'te bir fonksiyona bir yapının argüman olarak aktarılması görülüyor.



ŞEKİL 5.4: Bir yapının argüman olarak aktarılması.

Basit değişkenlerle olduğu gibi, `engldisp()`'deki `dd` isimli yapı tipindeki parametre, fonksiyona geçirilen argümanlarla (`d1` ve `d2`) aynı değildir. Böylece, `engldisp()` (gerçi burada böyle yapmıyor olsa da) `d1` ve `d2`'yi etkilemeden `dd`'yi değiştirebilir. Yani, eğer `engldisp()` aşağıdaki gibi ifadeler içermiş olsaydı, bu ifadelerin `main()`'in içindeki `d1` ve `d2` üzerinde hiçbir etkisi olmayacaktı.

```
dd.feet= 2;
dd.inches= 3.25;
```

Bir circle Yapısını Aktarmak

Bir fonksiyona bir yapıyı aktarmakla ilgili sıradaki örneğimiz Console Graphics Lite fonksiyonlarından yararlanıyor. Bu fonksiyonlar için kaynak ve başlık dosyaları Ek E'de "Console Graphics Lite" başlığı altında gösterilmiştir; ayrıca Giriş bölümünde bahsedildiği gibi yayıncının Web sitesinden de indirilebilir. Uygun olan başlık dosyalarını (derleyicinize bağlı olarak `MSOFTCON.H` veya `BORLACON.H`) programınıza dahil etmeniz ve kaynak dosyayı (`MSOFTCON.CPP` veya `BORLACON.CPP`) projenize eklemeniz gerekir. Console Graphics Lite fonksiyonları Ek E'de anlatılmıştır; projelere dosya ekleme prosedürü ise Ek C'de "Microsoft Visual C++" başlığı altında ve Ek D'de "Borland C++Builder" başlığı altında açıklanmıştır.

Bu örnekte, `circle` isminde bir yapı dairesel bir şekli simgeler. Daireler konsol ekranında belirli yerlere yerleştirilir ve her biri belirli bir yarıçapa sahiptir. Ayrıca dairelerin renkleri ve

dolgu desenleri de vardır. Renklerin ve dolgu desenlerinin olası değerlerini Ek E'de bulabilirsiniz. İşte `CIRCSTRC` için program listesi:

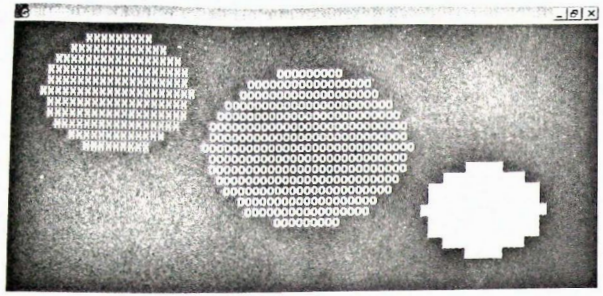
```
// circstrc.cpp
// grafik nesneleri olarak daireler
#include "msoftcon.h "              //grafik fonksiyonlari icin
////////////////////////////////////////////////////////////////
struct circle                       //grafik olarak daire
   {
   int xCo, yCo;                    //merkezin koordinatlari
   int radius;
   color fillcolor;                 //renk
   fstyle fillstyle;                //dolgu deseni
   };
////////////////////////////////////////////////////////////////
void circ_draw(circle c)
   {
   set_Color(c.fillcolor);                  //rengi ayarla
   set_fill_style(c.fillstyle);             //dolgu desenini ayarla
   draw_circle(c.xCo, c.yCo, c.radius);     //ici dolu bir daire ciz
   }
//---------------------------------------------------------------
int main()
   {
   init_graphics();                 //grafik sistemini ilk kullanima hazirla
                                    //daireleri olustur
   circle c1={ 15, 7, 5, cBLUE, X_FILL };
   circle c2={ 41, 12, 7, cRED, O_FILL };
   circle c3={ 65, 18, 4, cGREEN, MEDIUM_FILL };

   circ_draw(c1);                   //daireleri ciz
   circ_draw(c2);
   circ_draw(c3);
   set_cursor_pos(1, 25);           //imleci sol alt koseye tasi
   return 0;
   }
```

`circle` tipindeki değişkenlere, yani `c1`, `c2` ve `c3`'e, farklı farklı başlangıç değerleri verilir. `c1` için bu ifadenin nasıl göründüğüne bakalım:

```
circle c1= { 15, 7, 5, cBLUE, X_FILL };
```

Konsol ekranınızın 80 sütun ve 25 satır olduğunu varsayıyoruz. Bu tanımdaki ilk değer, 15, dairenin merkezinin yerleşeceği *sütun numarasını* (`x` koordinatını), 7 ise *satır numarasını* (ekranın en üstünden başlayarak aşağıya doğru `y` koordinatını) belirtir. 5, dairenin yarıçapını; `CBLUE`, rengini ve `X_FILL` sabiti ise dairenin içinin `X` ile doldurulacağını gösterir. Diğer iki daireye de aynı şekilde ilk değerleri atanır.

Tüm daireleri oluşturup, ilk değerlerini verir vermez, `circ_draw()` fonksiyonu her daire için bir kez olmak üzere toplam üç kez çağrılır ve daireler çizilir. Şekil 5.5 `CIRCSTRC` programının çıktısını gösterir. İtiraf etmeliyiz ki, daireler çok düzgün görünmüyor. Bu, konsol modundaki grafiklerde sınırlı sayıda piksel kullanılmasından kaynaklanır.



ŞEKİL 5.5: `CIRCSTRC` programının çıktısı.

Programda kullanılan yapının, dairelerin özelliklerini nasıl sakladığına dikkat edin. `circ_draw()` fonksiyonu ise dairelerin gerçekten bir iş yapmalarına neden olur (daireler kendilerini çizerler). "Nesneler ve Sınıflar" adlı Bölüm 6'da göreceğimiz gibi, nesneler, belli özellikler taşıyan ve iş yapan bir bütün ortaya çıkarmak amacıyla yapıların ve fonksiyonların bir araya getirilmesiyle oluşturulurlar.

Deklarasyonda Yer Alan İsimler

İşte size, fonksiyon deklarasyonlarınızın anlaşılırlığını artırmanın bir yolu. Bu fikir, deklarasyonlarda anlamlı isimler ve veri tipleri kullanmaya dayanıyor. Mesela, ekranda bir nokta görüntüleyen bir fonksiyon kullandığınızı varsayalım. Sadece veri tipleri içeren bir deklarasyon kullanabilirsiniz:

```
void display_point(int, int);       //deklarasyon
```

Fakat daha iyi bir yaklaşım şöyle olabilir:

```
void display_point(int horiz, int vert);   //deklarasyon
```

Bu iki deklarasyon, derleyiciye tam olarak aynı şeyi ifade eder. Ancak, ilk yaklaşımda `(int,int)` ifadesi hangi argümanın düşey, hangisinin yatay koordinat için olduğu hakkında hiç ipucu içermez. İkinci yaklaşımın avantajı, programcı için daha anlaşılır olmasıdır: Bu deklarasyonu gören birinin, fonksiyonu çağırırken doğru argümanları kullanması daha muhtemeldir.

Dikkat ederseniz, deklarasyonda kullanılan isimlerin, fonksiyonu çağırırken kullanılan isimler üzerinde hiçbir etkisi yoktur. Hangi argüman ismini kullanmak istiyorsanız, bunda tamamen özgürsünüz:

```
display_point(x, y);        //fonksiyon cagrisi
```

Program listesinin daha anlaşılır olması gerektiği durumlarda biz, bu isim artı veri tipi yaklaşımını kullanacağız.

Fonksiyonlardan Değer Döndürmek

Bir fonksiyon çalışmasını tamamladıktan sonra, kendisini çağıran programa tek bir değer döndürebilir. Genellikle döndürülen bu değer, fonksiyonun çözdüğü problemin cevabını içerir. Sıradaki örnek, pound cinsinden verilen ağırlığı kilogram cinsine çeviren bir fonksiyonu gösterir. `CONVERT` programının listesi şöyledir:

```
// convert.cpp
// donus degerleri ornegi, pound'u kilograma cevirir
#include <iostream>
using namespace std;
float lbstokg(float);           //deklarasyon

int main()
   {
   float lbs, kgs;

   cout << "\nEnter your weight in pounds: ";
   cin >> lbs;
   kgs = lbstokg(lbs);
   cout << "Your weight in kilograms is " << kgs << endl;
   return 0;
   }
//-------------------------------------------------------------
//lbstokg()
//pound'u kilograma ceviriyor
float lbstokg(float pounds)
   {
   float kilograms = 0.453592 * pounds;
   return kilograms;
   }
```

Bu programla etkileşim şöyle olabilir:

```
Enter your weight in pounds: 182
Your weight in kilograms is 82.553741
```

Bir fonksiyon bir değer döndürecekse, bu değerin veri tipi açıkça belirtilmelidir. Fonksiyon deklarasyonu bunu, veri tipini (bu örnekte `float`), fonksiyon deklarasyonunda ve tanımında fonksiyon isminden önce yerleştirerek gerçekleştirir. Önceki program örneklerindeki fonksiyonlar bir değer döndürmüyordu; bu nedenle, döndürülen değer tipi için `void` kullanılmıştı. `CONVERT` programında `lbstokg()` fonksiyonu `float` tipinde bir değer döndürür. Dolayısıyla, fonksiyon deklarasyonu şu şekilde olur:

```
float lbstokg(float);
```

Birinci `float` döndürülen değerin tipini belirtir. Parantez içindeki `float` ise `lbstokg()`'a aktarılacak argümanın tipinin de `float` olduğunu ifade eder.

Bir fonksiyon bir değer döndürdüğünde, fonksiyona yapılan çağrı bir deyim olarak ele alınır:

```
lbstokg(lbs);
```

Bu deyimin değeri, fonksiyonun döndürdüğü değer olur. Bu deyimi herhangi bir değişken gibi ele alabiliriz; örneğimizde bunu, bir değer atama ifadesinde kullanıyoruz:

```
kgs = lbstokg(lbs);
```

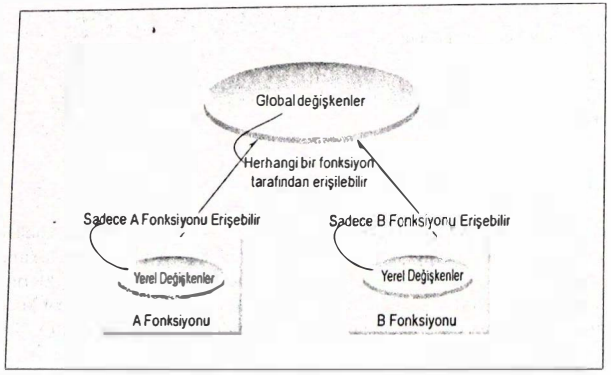
Bu ifade ile, **kgs** değişkenine **lbstokg()**'dan dönen değer atanır.

return İfadesi

lbstokg() fonksiyonuna pound cinsinden bir ağırlığı simgeleyen bir argüman aktarılır ve bu değer **pounds** parametresinde saklanır. Fonksiyon, pound değerini bir sabit ile çarparak kilogram cinsinden karşılık gelen ağırlığı hesaplar; sonuç **kilograms** değişkeninde saklanır. Bu değişkenin değeri daha sonra, fonksiyonu çağıran programa bir **return** ifadesi kullanılarak döndürülür:

```
return kilograms;
```

Dikkat ederseniz, hem **main()**'de hem de **lbstokg()**'da kilogram değişkenini saklamak için bir yer mevcuttur: **main()** içinde **kgs** ve **lbstokg()** içinde **kilograms**. Fonksiyon döndüğü zaman **kilograms** içindeki değer **kgs** içine *kopyalanır*. Fonksiyonu çağıran program, fonksiyonun içindeki **kilograms** değişkenine erişmez; sadece değişkenin değeri döndürülür. Bu işlem Şekil 5.6'da gösterilmiştir.



ŞEKİL 5.6: Bir değer döndürmek.

Bir fonksiyona birçok argüman aktırılabilirken, fonksiyondan sadece tek bir değer döner. Daha fazla bilgi döndürmeniz gerektiğinde bu bir kısıtlamadır. Bununla birlikte, fonksiyonlardan birden fazla değişken döndürmenin başka yöntemleri de vardır. Biri, bu bölümde daha sonra göreceğimiz, argümanları referans olarak aktarmaktır. Diğeri ise birazdan göreceğimiz gibi, birden fazla değeri birer üye olarak içeren bir yapıyı fonksiyondan döndürmektir.

Bir fonksiyonun döndürdüğü tipi fonksiyon deklarasyonuna her zaman dahil etmeniz gerekir. Eğer fonksiyon bir değer döndürmüyorsa, bunu belirtmek için **void** anahtar kelimesini

kullanın. Eğer fonksiyon deklarasyonunda döndürülen değerin tipini kullanmazsanız, derleyici fonksiyonun bir **int** döndürdüğünü varsayar. Örneğin, aşağıdaki deklarasyon **somefunc()** fonksiyonunun döndürdüğü değerin **int** tipinde olduğunu derleyiciye söyler.

```
somefunc();          //deklarasyon --- dondurulen tipi int olarak varsayar
```

Bunun nedeni tarihsel olarak, C'nin ilk sürümlerindeki kullanıma bağlıdır. Uygulamalarınızda bu varsayılan tipten yararlanmaya çalışmamalısınız. Gerçekte **int** olsa bile, her zaman döndürülen tipi açıkça belirtin. Bu, program listenizi tutarlı ve okunaklı kılar.

Gereksiz Değişkenlerden Kurtulmak

CONVERT programı, anlaşılırlık amacıyla kullanılan ama gerçekten gerekli olmayan birkaç tane değişken içerir. Bu programın bir varyasyonu olan **CONVERT2**, değişkenlerin yerine deyimler içeren fonksiyonların nasıl sık sık kullanıldığını gösterir.

```
// convert2.cpp
// gereksiz degiskenleri ortadan kaldirir
#include <iostream>
using namespace std;
float lbstokg(float);      //deklarasyon

int main()
   {
   float lbs;

   cout << "\nEnter your weight in pounds: ";
   cin >> lbs;
   cout << "Your weight in kilograms is " << lbstokg(lbs)
        << endl;
   return 0;
   }
//-------------------------------------------------------------
//lbstokg()
//pound'u kilograma cevirir
float lbstokg(float pounds)
   {
   return 0.453592 * pounds;
   }
```

main()'de **kgs** değişkeni **CONVERTS** programından çıkarılır. Bunun yerine, **cout** ifadesinin içine **lbstokg()** fonksiyonu doğrudan eklenir:

```
cout << "Your weight in kilograms is " << lbstokg(lbs) << endl;
```

Ayrıca, **lbstokg()** fonksiyonunda **kilograms** değişkeni artık kullanılmaz. **0.453592 * pounds** deyimi **return** ifadesine doğrudan eklenir:

```
return 0.453592 * pounds;
```

Hesaplama tamamlanır ve elde edilen sonuç, tıpkı bir değişkenin değerinde olduğu gibi, fonksiyonu çağıran programa döndürülür.

Anlaşılırlık sağlamak için programcılar genellikle **return** ifadesinde kullanılan deyimin etrafına parantez koyarlar:

```
return (0.453592 * pounds);
```

Derleyici tarafından gerekli olmasa bile, bir deyim içindeki ekstra parantezlerin bir zararı olmaz; üstelik, biz zavallı insanların program listelerini okumasını kolaylaştırmaya yardımcı olabilirler.

Deneyimli C++ (ve C) programcıları **CONVERT2**'nin kısa ve öz şeklini, **CONVERT**'in bol laflı şekline muhtemelen tercih edeceklerdir. Bununla beraber, **CONVERT2** özellikle uzman olmayanlar için anlaşılması kolay değildir. Cesarete karşı anlaşılırlık meselesi, kişisel tercihlerinize ve kodunuzu okuyacak olanların beklentilerine dayanan bir tarz meselesidir.

Yapı Değişkenlerini Döndürmek

Yapıların fonksiyonlara argüman olarak kullanılabileceğini görmüştük. Yapıları fonksiyondan dönen değer olarak da kullanabilirsiniz. **RETSRC** programı, **Distance** tipinde değişkenleri toplayan ve aynı tipte bir değer döndüren bir fonksiyon içerir:

```
// retstrc.cpp
// fonksiyondan bir yapi dondurmek
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct Distance                          //Ingiliz sistemine gore mesafe
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////////
Distance addengl(Distance, Distance);    //deklarasyonlar
void engldisp(Distance);

int main()
   {
   Distance d1, d2, d3;                  //uc tane uzunluk tanimla
                                         //kullanicidan d1 uzunlugunu al
   cout << "\nEnter feet: "; cin >> d1.feet;
   cout << "Enter inches: "; cin >> d1.inches;
                                         //kullanicidan d2 uzunlugunu al
   cout << "\nEnter feet: "; cin >> d2.feet;
   cout << "Enter inches: "; cin >> d2.inches;

   d3 = addengl(d1, d2);                 //d3, d1 ve d2 nin toplami
   cout << endl;
   engldisp(d1); cout << " + ";          //butun uzunluklari ekranda goster
   engldisp(d2); cout << " = ";
   engldisp(d3); cout << endl;
   return 0;
   }
//-------------------------------------------------------------
// addengl()
// Distance tipinde iki yapiyi toplar, sum'i dondurur
Distance addengl(Distance dd1, Distance dd2)
   {
   Distance dd3;                         //Sum icin yeni bir yapi tanimla

   dd3.inches = dd1.inches + dd2.inches;  //inches'leri topla
   dd3.feet = 0;                          //(muhtemel elde icin)
   if(dd3.inches >= 12.0)                 //eger inches >= 12.0,
      {                                   //inches'ten
      dd3.inches -= 12.0;                 //12.0 cikar ve
      dd3.feet++;                         //feet'i
      }                                   //1 artir
   dd3.feet += dd1.feet + dd2.feet;       //feet'leri topla
   return dd3;                            //yapiyi dondur
   }
//-------------------------------------------------------------
// engldisp()
// Distance tipindeki yapiyi ayak ve inc olarak ekranda goster
void engldisp(Distance dd)
   {
   cout << dd.feet << "\'-" << dd.inches << "\"";
   }
```


Programın kullanıcının inç ve ayak biçiminde iki uzunluk girmesini ister; bu iki uzunluğu **addengl()** fonksiyonunu çağırarak toplar ve **ENGLDISP** programında gösterilen **engldisp()** fonksiyonunu kullanarak sonucu ekranda gösterir. İşte programın bir çıktısı:

```
Enter feet: 4
Enter inches: 5.5

Enter feet: 5
Enter inches: 6.5
4 '-5.5 " + 5 '-6.5 "= 10 '-0 "
```

Programın **main()** bölümü, her biri **Distance** tipinde bir yapı ile simgelenen iki uzunluğu **addengl()** fonksiyonunu çağırarak toplar:

```
d3= addengl(d1, d2);
```

Bu fonksiyon **d1** ve **d2**'nin toplamını **Distance** tipinde bir yapı olarak döndürür. Sonuç, **main()**'de **d3** yapısına atanır.

Bu program, yapıların, döndürülen değer olarak nasıl kullanıldığını göstermesinin yanı sıra, ayrıca, aynı program içinde iki tane (**main()**'i de sayarsanız üç tane) fonksiyon kullanımı da gösterir. Fonksiyonları istediğiniz sırada düzenleyebilirsiniz. Tek kural şudur: Fonksiyon deklarasyonu program listesinde, bu fonksiyona yapılan çağrılardan önce yer almalıdır.

Referans Argümanları

Referans, bir değişkene *takma isim* (*alias*) – farklı bir isim – vermeye imkan verir. Referansların fonksiyonlara argüman olarak aktarılması en önemli kullanımlarından biridir.

Değer olarak aktarılan fonksiyon argümanlarının örneklerini görmüştük. Argümanlar değer olarak aktarıldığında, çağrılan fonksiyon argümanla aynı tipte yeni bir değişken tanımlar ve argümanın değerini bu değişkene kopyalar. Önceden vurguladığımız gibi, fonksiyon kendisini çağıran programdaki orijinal değişkene erişemez, sadece kendi hazırladığı kopyayı kullanabilir.

Şayet fonksiyonun kendisini çağıran programdaki orijinal değişkeni değiştirmesi gerekmiyorsa, argümanların değer olarak aktarılması avantajlıdır. Aslında bu yaklaşım, fonksiyonun orijinal değişkene zarar vermemesini garanti altına alır.

Argümanların referans olarak aktarılması farklı bir mekanizmaya dayanır. Fonksiyona bir değer aktarılması yerine, fonksiyonu çağıran programdaki orijinal değişkene *işaret eden bir referans* fonksiyona aktarılır. (Bu kadarını bilmenize gerek yok gerçi ama, yine de bahsedelim: Fonksiyona aktarılan aslında değişkenin *bellek adresidir*.)

Argümanların referans olarak aktarılmasının sağladığı önemli bir avantaj, fonksiyonun kendisini çağıran programdaki değişkenlere erişimini sağlamasıdır. Diğer avantajlarının yanı sıra bu özellik, fonksiyonun kendisini çağıran programa birden fazla değer döndürmesine imkan verir.

Basit Veri Tiplerini Referans Olarak Aktarmak

Sıradaki örnek olan **REF**, basit bir değişkenin referans olarak aktarılmasını gösterir.

```
// ref.cpp
// argumanin referans olarak aktarilmasi
#include <iostream>
using namespace std;

int main()
   {
   void intfrac(float, float&, float&);          //deklarasyon
   float number, intpart, fracpart;              //float degiskenleri

   do {
      cout << "\nEnter a real number: ";         //kullanicidan gelen sayi
      cin >> number;
      intfrac(number, intpart, fracpart);        //int ve frac'i bul
      cout << "Integer part is " << intpart       //ekrana yaz
           << ", fraction part is " << fracpart << endl;
      } while( number != 0.0 );                  //donguden 0.0 durumunda cik
   return 0;
   }
//--------------------------------------------------------------
// intfrac()
// reel sayilarin tam ve kesirli kisimlarini buluyor
void intfrac(float n, float& intp ,float& fracp)
   {
   long temp = static_cast<long>(n);             //long'a donustur
   intp = static_cast<float>(temp);              //tekrar float'a donustur
   fracp = n - intp;                             //tamsayi kismi cikar
   }
```

Bu programın **main()** bölümünde kullanıcının **float** tipinde bir sayı girmesi istenir. Program bu sayıyı tamsayı ve ondalık kısım olarak ikiye ayırır. Yani, kullanıcıdan gelen değer eğer 12.456 ise program, bu sayının tamsayı kısmının 12.0, ondalık kısmının 0.456 olduğunu bildirmesi gerekir. Bu iki değeri bulmak için **main()**, **intfrac()** fonksiyonunu çağırır. İşte örnek bir etkileşim:

```
Enter a real number: 99.44
Integer part is 99, fractional part is 0.44
```


Bazı derleyiciler ondalık kısım için gerçek olmayan bazı rakamlar üretebilir, mesela 0.440002 gibi. Bu, derleyicinin dönüşüm rutinindeki hatadan kaynaklanır ve ihmal edilebilir. Aşağıda Şekil 5.7'ye bir bakın.

intfrac() fonksiyonu, (n parametresine aktarılan) sayıyı bir tip ataması yardımıyla **long** tipinde bir değişkene çevirerek sayının tamsayı kısmını bulur. Tip ataması için şu deyim kullanılır:

```
long temp = static_cast<long>(n);
```

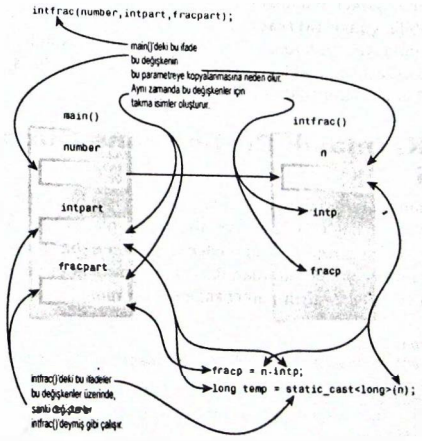
Bu ifade, sayının ondalık kısmını etkili bir biçimde keser, çünkü tamsayı tipindeki sayılar (elbette) sadece tamsayı kısmı içerirler. Sonuç daha sonra bir başka tip ataması (cast) yardımıyla yeniden **float** tipine döndürülür:

```
intp=static_cast<float>(temp);
```

Ondalık kısım, sayının kendisinden tamsayı kısmının çıkarılmış halidir. (**fmod()** ismindeki bir kütüphane fonksiyonunun aynı işlemleri **double** tipi için gerçekleştirdiğini vurgulamak isteriz.)

intfrac() fonksiyonu sayının tam ve ondalık kısımlarını bulabilir; peki, bunları tekrar **main()**'e nasıl döndürebilir? Bir **return** ifadesi kullanarak bir tanesini döndürebilir, fakat ikisini birden değil. Bu problem referans argümanları kullanılarak çözülür. İşte fonksiyonun deklarasyonu:

```
void intfrac(float n, float& intp, float& fracp)
```



ŞEKİL 5.7: **REF** programında argümanların referans olarak aktarılması.

Referans argümanları, veri tipinin peşinden gelen ampersand (**&**) simgesi ile belirtilir:

```
float& intp;
```

`&` simgesi, argüman olarak hangi değişken geçirilmişse, `intp`'nin o değişken için bir takma isim olduğunu belirtir. Diğer bir ifadeyle, `intfrac()` fonksiyonu içinde `intp` ismini kullandığınızda aslında `main()` içindeki `intpart`'ı kast etmiş olursunuz. `&` simgesi, "referanstır" anlamında kullanılabilir. Yani,

```
float& intp;
```

ifadesi, `intp` kendisine aktarılan `float` tipindeki değişkene bir referanstır, anlamındadır. Aynı şekilde, `fracp` de `fracpart` için bir takma isim, veya bir referanstır.

Fonksiyon deklarasyonu, tanımdaki ampersand kullanımını yineler:

```
void intfrac(float, float&, float&);        //ampersandlar
```

Tanımda da olduğu gibi ampersand, referans olarak aktarılan argümanların peşinden gelir. Ampersand fonksiyon çağrılarında kullanılmaz:

```
intfrac(number, intpart, fracpart);    //ampersand yok
```

Sadece fonksiyon çağrısına bakarak bir argümanın referans olarak mı, değer olarak mı aktarıldığını anlamının bir yolu yoktur.

`intpart` ve `fracpart` referans olarak geçirilirken, `number` değer olarak geçirilir. `intp` ve `intpart` bellekteki aynı yer için verilmiş farklı isimlerdir. `fracp` ve `fracpart` için de aynı şey geçerlidir. Diğer yandan, `intfrac()` içindeki `n` parametresi değer olarak aktarıldığı için bu, içine `number` değişkeninin değerinin kopyalandığı ayrı bir değişkendir. `n` parametresi değer olarak aktarılabilir, çünkü `intfrac()` fonksiyonu `number`'ı değiştirmek zorunda değildir.

(C programcıları *"referanstır"* anlamındaki ampersandı, *"adresidir"* anlamına gelen aynı simge ile karşılaştırmamalılar. Bunlar farklı kullanımlardır. `&` simgesinin *"adresidir"* anlamını "İşaretçiler" adlı Bölüm 10'da ele alacağız.)

Daha Karmaşık Bir Referans Olarak Aktarım Örneği

Basit argümanları referans olarak aktarmak için işte biraz daha karmaşık bir örnek. Diyelim ki, programınızda bir çift sayınız var ve siz, küçük olanın her zaman diğerinden önce gelmesini istiyorsunuz. Bunu yapmak için `order()` isminde bir fonksiyonu çağırabilirsiniz. `order()`, kendisine referans olarak aktarılan iki sayıyı kontrol eder ve eğer ilki diğerinden büyükse orijinal sayıların yerlerini değiştirir. `REFORDER` programının listesi şöyledir:

```
// reforder.cpp
// referans olarak aktarilan iki argumani siralar
#include <iostream>
using namespace std;

int main()
   {
   void order(int&, int&)              //prototip

   int n1=99, n2=11;                   //bu cift sirali degil
   int n3=22, n4=88;                   //bu cift sirali
```



```
   order(n1, n2);                      //her cifti sirala
   order(n3, n4);

   cout << "n1=" << n1 << endl;        //tum sayilari ekrana yaz
   cout << "n2=" << n2 << endl;
   cout << "n3=" << n3 << endl;
   cout << "n4=" << n4 << endl;
   return 0;
   }
//--------------------------------------------------------------
void order(int& numb1, int& numb2)     //iki sayiyi sirala
   {
   if(numb1 > numb2)                   //eger 1. 2.den buyukse,
      {
      int temp = numb1;                //yerlerini degistir
      numb1 = numb2;
      numb2 = temp;
      }
   }
```

`main()` içinde iki çift sayı vardır – birinci çift sıralı değil, ikincisi sıralı. `order()` fonksiyonu her çift için bir kez çağrılır, sonra tüm sayılar ekrana yazdırılır. Program çıktısı, birinci çiftin yerlerinin değiştirildiğini, ikinci çiftin değiştirilmediğini açıkça ortaya koyar. Programın çıktısı şöyledir:

```
n1=11
n2=99
n3=22
n4=88
```

`order()` fonksiyonu içinde birinci değişken `num1` ve ikinci değişken `num2` olarak adlandırılır. Eğer `num1`, `num2`'den büyükse fonksiyon `num1`'i `temp`'in içinde saklar, `num2`'yi `num1`'in içine koyar ve son olarak da `temp`'i yeniden `num2`'nin içine yerleştirir. `num1` ve `num2`'nin, fonksiyona hangi argümanlar geçirildiyse, o argümanlar için sadece farklı birer isim olduğunu hatırlayın; bu örnekte `num1` ve `num2`, fonksiyona yapılan ilk çağrıda `n1` ve `n2`, ikinci çağrıda `n3` ve `n4` için farklı birer isimdir. Elde edilen sonuç, fonksiyonu çağıran programdaki orijinal argümanların sırasını kontrol etmek ve gerekiyorsa bu argümanların yerlerini değiştirmektir.

Referans argümanlarını bu şekilde kullanmak bir tür uzaktan kumandalı bir işlemdir. Fonksiyonu çağıran program fonksiyona, programdaki hangi değişkenlerle işlem yapacağını bildirir. Fonksiyon da bu değişkenlerin gerçek isimlerini bile bilmeden bu değişkenleri değiştirir. Bu, şuna benzer: Boyacıları çağırıyorsunuz, fakat boyacılar ofislerinden hiç çıkmamalarına rağmen, siz arkanıza yaslanıyorsunuz ve yemek odanızın renginin esrarengiz biçimde değiştiğini izliyorsunuz.

Referans Olarak Yapıları Aktarmak

Basit veri tiplerini aktarır gibi yapıları da referans olarak aktarabilirsiniz. `REFERST` programı `Distance` tipindeki değerler üzerinde ölçek dönüşümleri gerçekleştirir. Bir ölçek dönüşümü, bir grup uzaklığı belli bir faktör ile çarpmayı gerektirir. Eğer uzaklık 6'-8" ise ve ölçek faktörü 0.5 ise, yeni uzaklık 3'-4" oluyor. Bu tür bir dönüşüm bir binanın tüm boyutlarına uygulanabilir; böylece bina küçülür fakat oranları aynı kalır.

```
// referst.cpp
// yapilari referans olarak aktarmak
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
struct Distance                         //Ingiliz uzaklik olculeri
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////
void scale( Distance&, float );         //fonksiyon
void engldisp( Distance );              //deklarasyonları

int main()
   {
   Distance d1={ 12, 6.5 };             //d1 ve d2'ye ilk degerleri ata
   Distance d2={ 10, 5.5 };

   cout << "d1 = "; engldisp(d1);       //eski d1 ve d2'yi ekranda goster
   cout << "\nd2 = "; engldisp(d2);

   scale(d1, 0.5);                      //d1 ve d2'yi olcekle
   scale(d2, 0.25);

   cout << "\nd1 = "; engldisp(d1);     //yeni d1 ve d2'yi ekranda goster
   cout << "\nd2 = "; engldisp(d2);
   cout << endl;
   return 0;
   }
//-----------------------------------------------------------
// scale()
// Distance tipindeki degerleri factor e gore olcekliyor
void scale( Distance& dd, float factor)
   {
   float inches = (dd.feet*12 + dd.inches) * factor;
   dd.feet = static_cast<int>(inches /12);
   dd.inches = inches - dd.feet * 12;
   }
//-----------------------------------------------------------
// engldisp()
// Distance tipindeki yapiyi ayak ve inc olarak ekranda gosterir
void engldisp( Distance dd )            //Distance tipinde dd parametresi
   {
   cout << dd.feet << "\'-" << dd.inches << "\"";
   }
```

REFERST iki **Distance** değişkenine – **d1** ve **d2** – başlangıçta belirli değerler atar ve bunları ekranda görüntüler. Sonra **scale()** fonksiyonunu çağırarak **d1**'i 0.5 ile ve **d2**'yi 0.25 ile çarpar. Son olarak, uzaklıkların sonuç değerlerini ekranda görüntüler. Programın çıktısı şöyledir:

```
d1=12 '-6.5 "
d2=10 '-5.5 "
d1=6 '-3.25 "
d2=2 '-7.375 "
```

scale() fonksiyonun yapılan iki çağrı şöyledir:


```
scale(d1,0.5);
scale(d2,0.25);
```

İlk çağrı **d1**'in 0.5 ile çarpılmasını, ikinci çağrı **d2**'nin 0.25 ile çarpılmasını sağlar. Dikkat ederseniz, tüm bu değişiklikler doğrudan **d1** ve **d2** üzerinde gerçekleşir. Fonksiyon hiçbir değer döndürmez; işlemler doğrudan, **scale()**'e referans olarak aktarılan **Distance** argümanı üzerinde gerçekleştirilir. (Fonksiyonu çağıran programda sadece bir değer değiştirildiği için fonksiyonu, argümanı değer olarak geçirecek ve ölçeklenen değeri döndürecek şekilde yeniden yazabilirsiniz. Böyle bir fonksiyonun çağrısı şu şekilde görünecektir:

```
d1= scale(d1, 0.5);
```

Yine de bu, gereksiz yere laf kalabalığıdır.)

Referans Olarak Aktarımla İlgili Notlar

C'de referanslar yoktur; işaretçiler, gerçi genellikle daha az kullanışlı olsalar da, hemen hemen aynı amaca hizmet ederler. Referans argümanları C++'a, basit değişkenlerin yanı sıra nesneler de dahil olmak üzere çok çeşitli durumlarda esneklik kazandırması için ilave edilmiştir.

Değer ve referans olarak aktarmanın yanı sıra, fonksiyonlara argüman aktarmanın bir üçüncü yolu işaretçi kullanmaktır. Bunu Bölüm 10'da inceleyeceğiz.

Aşırı Yüklenen Fonksiyonlar

Aşırı yüklenen (overloaded) bir fonksiyon kendisine gönderilen verinin cinsine bağlı olarak değişik etkinliklerde bulunabilir. Aşırı yüklemek meşhur bilim adamı fıkrasına benzer. Bu meşhur bilim adamı termosun tüm zamanların en büyük icadı olduğunda ısrar ediyormuş. Niye? "Bu mucizevi bir alet," demiş bilim adamı. "Sıcak şeyleri sıcak tutuyor, fakat soğuk şeyleri de soğuk tutuyor. Peki bunu nasıl biliyor?"

Aşırı yüklenen fonksiyonların ne yapacaklarını bilmeleri de eşit düzeyde esrarengiz görünebilir. Bu tür fonksiyonlar, bir çeşit veri üzerinde bir işlem, farklı bir veri üzerinde başka bir işlem yaparlar. Şimdi konuyu örneklerle daha anlaşılır hale getirelim.

Farklı Sayıda Argümanlar

TABLE örneğindeki **starline()** fonksiyonu ile **TABLEARG** örneğindeki **repchar()** fonksiyonunu hatırlayın. Her ikisini de bu bölümün önceki sayfalarında görmüştük. **starline()** fonksiyonu 45 asteriks kullanarak ekrana bir satır basıyordu; **repchar()** ise bir karakter ve satır uzunluğunu kullanıyordu, her ikisinin de değeri fonksiyon çağrıldığında belirleniyordu. Şimdi, her zaman 45 karakter basan, fakat basılacak karakteri kendisini çağıran programın belirlemesine izin veren üçüncü bir fonksiyon olan **charline()** fonksiyonunu düşünelim. Bu üç fonksiyonun her biri – **starline()**, **repchar()** ve **charline()** – aynı etkinliklerde bulunur, fakat değişik isimlere sahiptirler. Programcılar açısından bu fonksiyonları kullanmak, üç tane isim hatırlamak ve şayet uygulamanın *Fonksiyon Referans* dokümanında alfabetik olarak listelenmişlerse bunları üç yerde aramak anlamına gelir.

Her bir fonksiyonun farklı argümanları olsa bile, bu üç fonksiyonun tümü için aynı ismi kullanmak çok daha uygun olurdu. İşte aşağıdaki program olan **OVERLOAD**, bunu mümkün kılar:

```
// overload.cpp
// fonksiyonlarin asiri yuklenmesi
#include <iostream>
using namespace std;

void repchar();                          //deklarasyonlar
void repchar(char);
void repchar(char, int);

int main()
   {
   repchar();
   repchar('=');
   repchar('+', 30);
   return 0;
   }
//-------------------------------------------------------------
//repchar()
//45 asteriks goruntuler
void repchar()
   {
   for(int j=0; j<45; j++)               //dongu her zaman 45 kez doner
      cout << '*';                       //her zaman asteriks basar
   cout << endl;
   }
//-------------------------------------------------------------
// repchar()
// belirlenen karakteri 45 kez kopyalar
void repchar(char ch)
   {
   for(int j=0; j<45; j++)               //dongu her zaman 45 kez doner
      cout << ch;                        //belirlenen karakteri basar
   cout << endl;
   }
//-------------------------------------------------------------
// repchar()
// belirlenen karakterin belirlenen sayida kopyasini goruntuler
void repchar(char ch,int n)
   {
   for(int j=0; j<n; j++)                //dongu n kez doner
      cout << ch;                        //belirlenen karakteri basar
   cout << endl;
   }
```

Bu program karakterlerden oluşan üç satır basar. Çıktı şöyledir:

```
*********************************************
=============================================
++++++++++++++++++++++++++++++
```

İlk iki satır 45 karakter, üçüncü ise 30 karakter uzunluğundadır.

Program aynı isme sahip üç tane fonksiyon içerir. Üç tane fonksiyon deklarasyonu, üç tane fonksiyon çağrısı ve üç tane fonksiyon tanımı vardır. Derleyiciyi ümitsizce karmaşaya düşmekten alıkoyan nedir? Derleyici, fonksiyonların birini diğerinden ayırmak için fonksiyon imzasını – argümanların sayısını ve veri tiplerini – kullanır. Başka bir deyişle, hiç argüman almayan şu deklarasyon


```
void repchar();
```

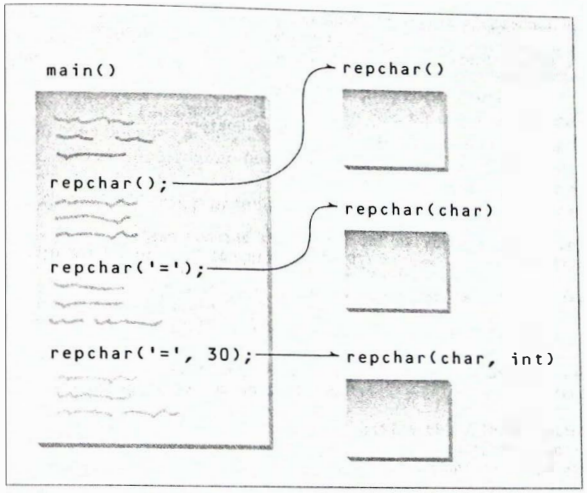
aşağıdaki deklarasyonun ifade ettiği fonksiyondan tamamen farklı bir fonksiyonu kast eder:

```
void repchar(char);
```

Bu fonksiyon **char** tipinde bir argüman alır. Ya da hem **char** hem de **int** tipinde iki argüman alan şu deklarasyondan da tamamen farklı bir fonksiyonu kast eder:

```
void repchar(char, int);
```

İsimleri aynı, fakat argüman sayıları farklı birkaç tane fonksiyonu gören derleyici programcının hata yaptığını düşünebilirdi (C'de olacağı budur). Fakat bunun yerine derleyici, çok toleranslı biçimde her tanım için ayrı bir fonksiyon hazırlar. Bu fonksiyonlardan hangisinin çağrılacağı, çağrıda belirtilen argüman sayısına dayanır. Şekil 5.8 bu işlemi gösteriyor.



ŞEKİL 5.8: Aşırı yüklenmiş fonksiyonlar.

Değişik Argüman Türleri

OVERLOAD örneğinde isimleri aynı fakat argüman sayıları farklı birkaç tane fonksiyon geliştirdik. Derleyici, tipleri farklı olmak şartıyla, aynı argüman sayısına sahip aşırı yüklenmiş fonksiyonlar arasında da ayrım yapabilir. Aşağıdaki program, **OVERENGL**, bir niceliği ayak ve inç biçiminde görüntülemek için aşırı yüklenmiş bir fonksiyon kullanır. Fonksiyonun tek argümanı **Distance** tipinde bir yapı (**ENGLDISP** programında kullanılan tipte) veya **float** tipinde basit bir değişken olabilir. Programda argümanın tipine bağlı olarak değişik fonksiyonlar kullanılır.

```
// overengl.cpp
// asiri yuklenen fonksiyonlar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
struct Distance                         //Ingiliz uzunluk sistemi
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////
void engldisp(Distance);                //deklarasyonlar
void engldisp(float);

int main()
   {
   Distance d1;                         //Distance tipinde uzaklik
   float d2;                            //float tipinde uzaklık
                                        //kullanicidan d1 uzunlugunu al
   cout << "\nEnter feet: "; cin >> d1.feet;
   cout << "Enter inches: "; cin >> d1.inches;
                                        //kullanicidan d2 uzunlugunu al
   cout << "Enter entire distance in inches: "; cin >> d2;
   cout << "\nd1 = ";
   engldisp(d1);                        //1. Uzunlugu goster
   cout << "\nd2 = ";
   engldisp(d2);                        //2. Uzunlugu goster
   cout << endl;
   return 0;
   }
//-------------------------------------------------------------
// engldisp()
// Distance tipindeki yapiyi ayak ve inc formatinda ekranda goster
void engldisp(Distance dd)              //Distance tipinde dd isimli parametre
   {
   cout << dd.feet << "\'-" << dd.inches << "\"";
   }
//-------------------------------------------------------------
// engldisp()
// float tipindeki degiskeni ayak ve inc formatinda goruntule
void engldisp(float dd)                 //float tipinde dd isimli parametre
   {
   int feet = static_cast<int>(dd / 12);
   float inches = dd - feet*12;
   cout << feet << "\'-" << inches << "\"";
   }
```

Kullanıcıdan iki uzaklık değeri girmesi istenir. Birinci değer, ayak ve inç olarak ayrı ayrı girilmeli, ikinci değer inç cinsinden tek bir sayı olarak (örneğin, 9'-1.5" yerine 109.5 inç olarak) girilmelidir. Program, birinci uzaklık için **Distance** tipinde bir değer görüntülemek için ve ikinci uzaklık için **float** tipinde bir değer görüntülemek için aşırı yüklenmiş bir fonksiyon olan **engldisp()**'i çağırır. Programla etkileşim şöyle olabilir:

```
Enter feet: 5
Enter inches: 10.5
Enter entire distance in inches: 76.5
d1 = 5 '-10.5 "
d2 = 6 '-4.5 "
```


Dikkat ederseniz, **engdisp()**'in değişik versiyonları aynı işleri yapsa da, ikisinin kodu oldukça farklıdır. Girdiyi sadece inç olarak alan versiyonun, sonucu görüntülemeden önce ayak ve inç formatına çevirmesi gerekir.

Aşırı yüklenen fonksiyonlar, hatırlanması gereken fonksiyon isimlerini azaltarak programcının yaşamını kolaylaştırabilirler. Aşırı yüklenen fonksiyonların kullanılmadığı durumda ortaya çıkan karmaşıklığa bir örnek olarak, bir sayının mutlak değerini bulmak için kullanılan C++ kütüphane rutinlerini ele alabiliriz. Bu rutinlerin C++'ın yanı sıra C'de de çalışması gerektiğinden (C'de fonksiyonların aşırı yüklenmesi yoktur) her veri tipi için ayrı bir mutlak değer rutininin olması gerekir. Bu rutinlerden dört tane vardır: **int** tipi için **abs()**, karmaşık sayılar için **cabs()**, **double** tipi için **fabs()** ve **long** tipi için **labs()**. C++'ta tek bir isim, **abs()**, tüm bu veri tipleri için yeterli olurdu.

Sonradan göreceğimiz gibi, aşırı yüklenen fonksiyonlar değişik tipte nesneleri ele almak için de ayrıca kullanışlıdır.

Fonksiyonların Yinelenmesi

Fonksiyonların varlığı, yinelenme (*recursion*) olarak adlandırılan bir programlama tekniğini mümkün kılıyor. Yinelenme, bir fonksiyonun kendisini çağırmasıdır. Bu biraz olmayacak bir şey gibi gelebilir; üstelik, yinelenen bir fonksiyon genellikle bir programlama hatasıdır. Yine de, doğru kullanıldığında bu teknik şaşırtıcı ölçüde güçlü olabilir.

Yinelenme, uzun uzun açıklamalardan çok, bir örnek ile çok daha kolay anlaşılır. Öyleyse bunu daha önceden gördüğümüz bir programa uygulayalım: "Döngüler ve Kararlar" adlı Bölüm 3'te **FACTOR** programının farklı bir sürümü aşağıda görülüyor. Bu program bir sayının faktöriyelini hesaplamak için bir **for** döngüsü kullanmıştı. (Faktöriyellerle ilgili açıklama için o örneğe bakınız.) Yeni programımız olan **FACTOR2**, döngü yerine yinelenme kullanır.

```
// factor2.cpp
// yinelenme kullanarak faktoriyel hesabi
#include <iostream>
using namespace std;
unsigned long factfunc(unsigned long);      //deklarasyon

int main()
   {
   int n;                                   //kullanici tarafindan girilen sayi
   unsigned long fact;                      //faktoriyel

   cout << "Enter an integer: ";
   cin >> n;
   fact = factfunc(n);
   cout << "Factorial of " << n << " is " << fact << endl;
   return 0;
   }
//-------------------------------------------------------------
// factfunc()
// faktoriyelleri hesaplamak icin kendi kendisini cagirir
unsigned long factfunc(unsigned long n)
   {
   if(n > 1)
      return n * factfunc(n-1);             //kendini cagir
   else
      return 1;
   }
```

Bu programın çıktısı Bölüm 3'teki FACTOR programının çıktısı ile aynıdır.

FACTOR2'nin main() bölümü mantıklı görünür: main(), argümanı kullanıcının girdiği bir sayı olan factfunc() isminde bir fonksiyonu çağırır. Fonksiyon daha sonra bu sayının faktöriyelini main()'e döndürür.

factfun() fonksiyonu ayrı bir hikayedir. Bu fonksiyon ne yapar? Eğer n, 1'den büyükse, fonksiyon kendini çağırır. Dikkat ederseniz, bunu yaparken çağrıldığı argümandan sayı değeri bir eksik olan bir argüman kullanır. Varsayalım, bu fonksiyon main()'den 5 argümanı ile çağrılmış olsun. Bu durumda, kendisinin ikinci bir versiyonunu 4 argümanı ile çağıracaktır. Sonra üçüncü versiyonunu 3 argümanı ile çağıracaktır, vs.

Fonksiyonun her versiyonunun, bir yandan kendi kendisini yeniden çağırmakla meşgulken diğer yandan kendi n değerini de sakladığına dikkat edin.

factfun() kendi kendisini dört kez çağırdıktan sonra, fonksiyonun beşinci versiyonu 1 argümanı ile çağrılır. Fonksiyon bunu if ifadesinden anlar ve bu kez kendini çağırmak yerine, önceki versiyonlardan farklı olarak, dördüncü versiyona 1 değerini döndürür. Dördüncü versiyon değer olarak 2'yi saklamıştı; sakladığı 2 değerini kendisine dönen 1 değeri ile çarpar ve sonucu (2) üçüncü versiyona döndürür. Üçüncü versiyon değer olarak 3'ü saklamıştı; sakladığı 3 değerini kendisine dönen 2 değeri ile çarpar ve sonucu (6) ikinci versiyona döndürür. İkinci versiyon değer olarak 4'ü saklamıştı; sakladığı 4 değerini kendisine dönen 6 ile çarpar ve sonucu (24) birinci versiyona döndürür. Birinci versiyon değer olarak 5'i saklamıştı; sakladığı 5 değerini kendisine dönen 24 değeri ile çarpar ve sonucu (120) main()'e döndürür.

Dolayısıyla, bu örnekte beş tane fonksiyon çağrısı vardır. Her fonksiyon çağrısının peşinden bir adet fonksiyon dönüşü gelir. İşte bu işlemin bir özeti:

Versiyon	İş	Argüman veya Döndürülen Değer
1	Çağrı	5
2	Çağrı	4
3	Çağrı	3
4	Çağrı	2
5	Çağrı	1
5	Dönüş	1
4	Dönüş	2
3	Dönüş	6
2	Dönüş	24
1	Dönüş	120

Yinelenen her fonksiyona, yinelemeye son vermek için bir şart sağlamak zorundayız. Aksi halde, fonksiyon sonsuza kadar kendisini çağırır ve programın çökmesine neden olur. factfunc() fonksiyondaki if ifadesi bu görevi görür; n, 1 olduğunda yinelemeyi durdurur.

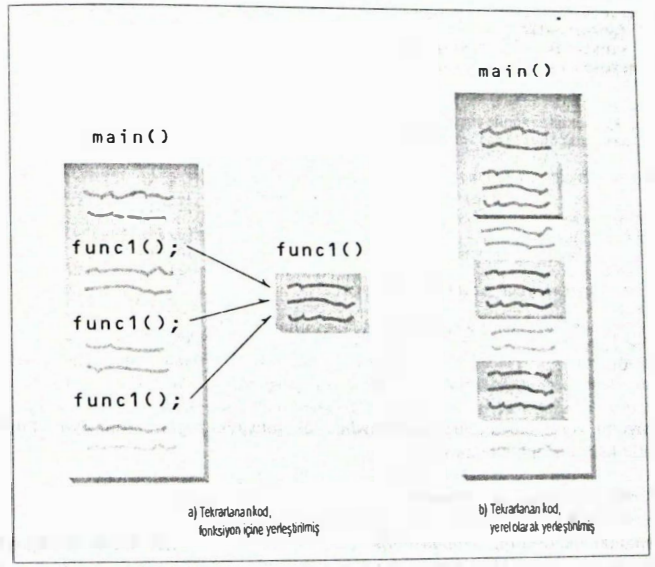
Yinelenen bir fonksiyon kendisini çağırırken, bunun birçok versiyonunun bellekte saklandığı doğru mu? Aslında, değil. Her versiyonun değişkenleri bellekte saklanır, fakat fonksiyonun kodu tek kopyadır. Böyleyken bile, defalarca kendi kendini çağıran (yinelenen) bir fonksiyon bellekte saklanan çok sayıda değişken üretebilir; sistemin hepsini saklayacak yeterli yeri yoksa bu durum sistem açısından problem oluşturabilir.

Yerel (Inline) Fonksiyonlar

Fonksiyonların bellek alanından kazandırdığını bahsetmiştik, çünkü fonksiyona yapılan tüm çağrılar aynı kodun çalıştırılmasına neden olur; fonksiyonun gövdesinin bellekte kopyalanmasına gerek kalmaz. Derleyici, bir fonksiyon çağrısı gördüğünde normal olarak fonksiyonun içine zıplamak için bir komut üretir. Fonksiyonun sonunda ise, bu bölümde önceki sayfalarda Şekil 5.1'de gösterildiği gibi, çağrının peşinden gelen komuta geri zıplar.

Bu olaylar zinciri belki bellek alanından kazandırıyor olabilir, fakat fazladan zaman alır. Önce fonksiyona atlamak için bir komut (Assembly dilinde CALL komutu gibi veya benzeri bir komut) gereklidir; ayrıca saklaçları (register) kaydetmek için, fonksiyonu çağıran programdaki yığına (stack) argümanları yüklemek için, fonksiyon içinde bunları yığından argümanları çıkarmak için (eğer fonksiyonun argümanı varsa), saklaç bilgilerini saklaçlara geri kaydetmek için çeşitli komutlar gereklidir. Ayrıca fonksiyonu çağıran programa geri dönmek için de bir komut gereklidir. Fonksiyonun döndürdüğü değer de (eğer varsa) ilaveten ele alınmalıdır. Tüm bu komutlar programın çalışmasını yavaşlatır.

Kısa fonksiyonlar kullanıldığında programın çalışma süresinden kazanmak için fonksiyon gövdesini doğrudan fonksiyonu çağıran programın kodunun içine yerel (inline) olarak yerleştirmeyi tercih edebilirsiniz. Yani, kaynak dosyadan yapılan her fonksiyon çağrısında, fonksiyona atlamak yerine fonksiyonun gerçek kodu programa yerleştirilir. Bir fonksiyon ve yerel kod arasındaki fark Şekil 5.9'da gösteriliyor.



ŞEKİL 5.9: Fonksiyona karşı yerel kod.

Tekrarlanan kod uzun bir program parçası ise bu tür kodların normal birer fonksiyon gibi ele alınmaları genellikle daha iyidir: Bellek alanından elde edilen kazanım, çalışma hızından yapılacak nispeten küçük fedakarlığa değer. Fakat, kısa bir kod parçasını sıradan bir fonksiyon gibi ele almak, bellek alanından çok az bir tasarruf sağlarken, zaman açısından büyük bir fonksiyon kadar bir yük bindirir. Aslında, eğer fonksiyon çok kısa ise, bunu çağırmak için gerekli komutlar, fonksiyonun gövdesindeki komutlar kadar bir alan kaplayabilir. Böylece, sırf zaman açısından değil, aynı zamanda bellek açısından da yük binmiş olur.

Bu gibi durumlarda, gerekli kodu, gereken yerlere aynı ifadeler grubunu ekleyerek tekrarlayabilirsiniz. Aynı kod parçasını tekrarlayarak eklemenin tehlikesi, program düzeninin sağladığı avantajları ve fonksiyonlarla elde edilen anlaşılırlığı yitirmektir. Program daha hızlı çalışıp, daha az yer kaplayabilir belki ama, program listesi daha uzun ve daha karmaşık hale gelir.

Bu kararsızlığın çözümü *yerel fonksiyonlardır*. Bu tür bir fonksiyon, kaynak dosya içinde normal bir fonksiyon gibi yazılır, fakat derleyici tarafından bir fonksiyon olarak değil, yerel kod olarak derlenir. Kaynak dosya, iyi organize edilmiş ve kolay okunur şeklini korur, çünkü fonksiyon, kod içinde ayrı bir bütün olarak gösterilir. Bununla birlikte, program derlenirken, bir fonksiyon çağrısı yapıldığında, fonksiyonun gövdesi gerçekte programa eklenir.

Çok kısa olan fonksiyonlar, diyelim ki bir-iki satır uzunluğunda olanlar, yerel olarak yazılmaya adaydır. İşte, **CONVERT2** programının bir varyasyonu olan **INLINE**. Bu programda **lbstokg()** fonksiyonu yerel olarak kullanılıyor.

```
// inliner.cpp
// yerel fonksiyonlar
#include <iostream>
using namespace std;

//lbstokg()
//pound'u kilograma donusturur
inline float lbstokg(float pounds)
   {
   return 0.453592 * pounds;
   }
//-------------------------------------------------------------
int main()
   {
   float lbs;
   cout << "\nEnter your weight in pounds: ";
   cin >> lbs;
   cout << "Your weight in kilograms is " << lbstokg(lbs)
        << endl;
   return 0;
   }
```

Bir fonksiyonu yerel hale getirmek kolaydır: Tek ihtiyacınız olan, fonksiyon tanımında **inline** anahtar kelimesini kullanmaktır:

```
inline float lbstokg(float pounds)
```

inline anahtar kelimesinin aslında derleyiciye yapılan bir *istekten* ibaret olduğunun farkında olmalısınız. Derleyici kimi zaman bu isteği dikkate almaz ve fonksiyonu normal bir

fonksiyon gibi derleyebilir. Mesela, fonksiyonun yerel olabilmek için fazlasıyla uzun olduğuna karar verebilir.

(C programcıları, C++'taki yerel -inline- fonksiyonların çoğunlukla C'deki **#define** makrolarının yerini aldığına dikkat etmelidirler. C++'taki yerel fonksiyonlar, C'dekilerle aynı amaca hizmet ederler, fakat daha iyi tip kontrolü sağlarlar ve C'nin makrolarındaki gibi parantez kullanımıyla ilgili özel ilgi gerektirmezler.)

Varsayılan Argümanlar

Şaşırtıcı gelebilir ama, bir fonksiyon, argümanlarının tümünü açıkça belirtmeden de çağrılabilir. Bu özellik, her tür fonksiyon üzerinde çalışmaz: Fonksiyon deklarasyonu, açıkça belirtilmeyen argümanlar için varsayılan (default) değerler sağlamalı.

OVERLOAD programının bir varyasyonu olan aşağıdaki program bu özelliği gösterir. **OVERLOAD** içinde değişik sayıda argümanla başa çıkması için aynı isme sahip üç değişik fonksiyon kullanmıştık. Şimdiki örnek, **MISSARG**, aynı amaca farklı bir yoldan ulaşır.

```
// missarg.cpp
// eksik ve varsayilan argumanlar
#include <iostream>
using namespace std;

void repchar(char='*',int=45);          //varsayilan argumanlarla
                                        //fonksiyon deklarasyonu
int main()
   {
   repchar();                           //45 asteriks bas
   repchar('=');                        //45 esittir isareti bas
   repchar('+', 30);                    //30 arti isareti bas
   return 0;
   }
//-------------------------------------------------------------
// repchar()
// ekranda karakterlerden olusan bir satir goruntule
void repchar(char ch, int n)            //gerekiyorsa
   {                                    //varsayilan degerler fonksiyona saglanir
   for(int j=0; j<n; j++)               //dongu n kez donuyor
      cout << ch;                       //ch'i ekrana basiyor
   cout << endl;
   }
```

Bu programda **repchar()** fonksiyonu iki argüman alır. Fonksiyon, **main()**'den üç kez çağrılır. İlk seferinde fonksiyon argümansız çağrılır, ikincide bir argüman ile, üçüncü de iki argüman ile çağrılır. İlk iki çağrı neden çalışıyor? Çünkü çağrılan fonksiyon, fonksiyonu çağıran program tedarik etmediği durumda kullanılmak üzere tanımlanmış varsayılan argümanlar içerir. Varsayılan argümanlar, **repchar()**'ın deklarasyonunda açıkça belirtilir:

```
void repchar(char='*', int=45);     //deklarasyon
```

Varsayılan argüman, tip isminden hemen sonra yerleştirilen eşittir işaretinin peşinden gelir. Deklarasyonda ayrıca, aşağıda gösterildiği gibi, değişken isimleri de kullanabilirsiniz:

```
void repchar(char reptChar='*', int numberReps=45);
```

Bir fonksiyon çağrılırken eğer bir argüman eksikse, bu argümanın son argüman olduğu varsayılır. `repchar()` fonksiyonu, tek argümanın değerini `ch` parametresine atar ve `n` parametresi için varsayılan 45 değerini kullanılır.

Eğer iki argüman da eksikse, fonksiyon `ch`'e, `'*'` varsayılan değerini ve `n`'ye de 45 varsayılan değerini atar. Yani, fonksiyona yapılan üç çağrının her biri farklı sayıda argüman içeriyor olsalar bile, düzgün çalışır.

Eksik olan argümanların, argüman listesinin en sonunda yer alan argümanlar olması gerektiğini hatırınızdan çıkarmayın. Sondaki üç argümanı çıkarabilirsiniz, fakat sondan bir öncekini çıkarıp, sonuncuyu dahil edemezsiniz. Bu makul bir yaklaşımdır: Ortadan bir tane çıkardığınızda derleyici hangi argümanı çıkardığınızı nereden bilebilir? (Eksik argümanlar virgül ile ayrılabilirdi, fakat virgül yazım hatalarına maruz kalabildiği için C++ tasarımcıları bu ihtimali göz ardı etmişlerdir.) Fonksiyonun varsayılan değer sağlamadığı argümanları dışarıda bırakırsanız, derleyicinin hata mesajı verecek olması şaşırtıcı değildir.

Varsayılan argümanlar, mesela argümanların hemen hemen her zaman aynı değere sahip olduğu durumlarda, argümanları tekrar tekrar yazma zahmetine girmek istemediğiniz zaman kullanışlı olabilir. Bu tür argümanlar ayrıca, program yazıldıktan sonra programcının, başka bir argüman ekleyerek fonksiyonun becerisini artırmaya karar verdiği durumlarda da kullanışlı olabilir. Varsayılan argüman kullanmak demek, yeni fonksiyon çağrıları daha fazla sayıda argüman kullanırken mevcut fonksiyon çağrıları eski sayıda argüman kullanarak işlerine devam edebilir demektir.

Kapsam (Scope) ve Depolama Sınıfı

Fonksiyonlar hakkında bilgi sahibi olduğumuza göre, değişkenler ve fonksiyonların birbiriyle etkileşimiyle ilgili C++'ın iki özelliğini inceleyebiliriz: *Kapsam* (*scope*) ve *depolama sınıfı*. Bir değişkenin kapsamı programın hangi bölümlerinin bu değişkene erişebileceğini belirler. Değişkenin depolama sınıfı ise değişkenin ne kadar süre var olacağını gösterir. Bunu kısaca özetleyeceğiz, sonra konuyu daha ayrıntılı ele alacağız.

İki değişik türde kapsam burada önemlidir: Yerel kapsam ve dosya kapsamı. (Daha sonra bir başkasını, sınıf kapsamını da göreceğiz.

- *Yerel* kapsama sahip değişkenler sadece bir blok içinde erişilebilirler.
- *Dosya* kapsamına sahip değişkenler bir dosya boyunca erişilebilirler.

Bir blok temel olarak bir çift küme parantezi içinde yer alan koddur. Dolayısıyla, bir fonksiyon gövdesi de bir bloktur.

İki çeşit depolama sınıfı vardır: Otomatik ve statik.

- *Otomatik* depolama sınıfına sahip değişkenler, içinde tanımlı oldukları fonksiyonun yaşam süresi boyunca varlıklarını sürdürürler.
- *Statik* depolama sınıfına sahip değişkenler programın yaşam süresi boyunca varlıklarını sürdürürler.

Şimdi bunların ne anlama geldiğini görelim.

Yerel Değişkenler

Şimdiye kadar örnek programlarda kullandığımız değişkenlerin neredeyse tümü kullanıldıkları fonksiyon gövdesinin içinde tanımlanmışlardı. Yani, tanımlar fonksiyon gövdesini sınırlayan küme parantezlerinin içinde kalıyordu:


```
void somefunc()
   {
   int somevar;              //degiskenler
   float othervar;           //fonksiyon govdesi icinde tanimlaniyor
   // diger ifadeler
   }
```

Değişkenler `main()`'in içinde veya diğer fonksiyonların içinde tanımlanabilir; her ikisinin de etkisi aynıdır, çünkü `main()` de bir fonksiyondur. Bir fonksiyon içinde tanımlanan değişkenlere *yerel* (*local*) *değişkenler* denir, çünkü bu değişkenler yerel kapsama sahiptir. Bunlara kimi zaman otomatik değişkenler de denir, çünkü bu değişkenlerin otomatik depolama sınıfı vardır.

Şimdi, fonksiyonlar içinde tanımlanan değişkenlerin bu iki önemli özelliğini inceleyelim.

Depolama sınıfı

Bir yerel değişken, içinde tanımlandığı fonksiyon çağrılana kadar ortaya çıkmaz. (Daha doğru bir ifade ile şöyle söyleyebiliriz, *herhangi* bir kod bloğu içinde tanımlanan değişkenler bu kod bloğu çalıştırılana kadar oluşturulmazlar. Yani, bir döngü gövdesi içinde tanımlanan değişkenler sadece döngü çalıştırıldığı sürece mevcuttur.) Az önce verilen program parçasında `somevar` ve `othervar` isimli değişkenler `somefunc()` fonksiyonu çağrılana kadar mevcut değildir. Yani, bunların değerlerini saklamak için bellekte yer ayrılmamıştır; bu değişkenler tanımsızdır. Kontrol `somefuc()`'a geçtiğinde, değişkenler oluşturulur ve bunlar için bellekte yer ayrılır. Daha sonra, `somefunc()` geri döndüğünde ve kontrol tekrar fonksiyonu çağıran programa geçtiğinde, değişkenler ortadan kaldırılır ve değerleri yitirilir. Bu değişkenlere *otomatik* ismi verilir, çünkü değişkenler fonksiyon çağrıldığında otomatik olarak oluşturulur ve fonksiyon geri döndüğünde otomatik olarak ortadan kaldırılır.

Bir değişkenin oluşturulması ve ortadan kaldırılması arsında geçen süreye değişkenin *yaşam süresi* (kimi zaman da *sürekliliği*) denir. Yerel bir değişkenin yaşam süresi, içinde tanımlı olduğu fonksiyonun çalışma süresi ile örtüşür.

Değişkenlerin yaşam sürelerini kısıtlamanın arkasında yatan fikir, bellek alanından kazanmaktır. Eğer bir fonksiyon çalışmıyorsa, bu fonksiyon çalışırken kullanılan değişkenlere muhtemelen ihtiyaç yoktur. Bunları ortadan kaldırmak bellek alanını boşaltır, böylece bu bellek alanı diğer fonksiyonlar tarafından kullanılabilir.

Kapsam

Bir değişkenin *kapsamı* (scope), *erişilebilirliği* olarak da adlandırılır, bir program içinde değişkenin erişilebildiği yerleri tanımlar. Değişken, programın bazı bölümlerinde yer alan ifadelerin içinde kullanılabilir, fakat diğer ifadelerde böyle bir girişim *bilinmeyen değişken* hata mesajına yol açabilir. Değişkenin kapsamı, değişkenin görülebildiği (erişilebildiği) program parçasıdır.

Bir fonksiyon içinde tanımlı değişkenler, sadece tanımlı oldukları fonksiyon içinde görülebilirler, yani, erişilebilirler. Diyelim ki, bir program içinde iki fonksiyonunuz var:

```
void somefunc()
   {
   int somevar;              //yerel degiskenler
   float othervar;

   somevar = 10;             //tamam
   othervar = 11;            //tamam
```

```
    nextvar = 12;                   //kural disi: somefunc() icinde erisilebilir degil
    }

void otherfunc()
   {
   int nextvar;                     //yerel degisken

   somevar = 20;                    //kural disi: otherfunc() icinde erisilebilir degil
   othervar = 21;                   //kural disi: otherfunc() icinde erisilebilir degil
   nextvar = 22;                    //tamam
   }
```

`nextvar` değişkeni `somefunc()` fonksiyonu içinde erişilebilir değildir; `somevar` ve `othervar` değişkenleri de `otherfunc()` içinde erişilebilir değildir.

Değişkenlerin erişilebilirliğini kısıtlamak programın düzenlenmesine ve modüllere ayrılmasına yardımcı olur. Bir fonksiyon içindeki değişkenlerin, diğer fonksiyonların kazara yapabileceği değişikliklere karşı emniyette olduğu konusunda içiniz rahat olabilir, çünkü diğer fonksiyonlar bu değişkenleri göremez. Bu özellik, eski moda prosedürel programları organize etme yöntemi olan *yapısal programlamanın* önemli bir parçasıdır. Değişkenlerin erişilebilirliğini kısıtlamak aynı zamanda nesne yönelimli programlamanın da önemli bir parçasıdır.

Bir fonksiyon içinde tanımlı değişkenler açısından depolama sınıfı ile kapsam örtüşür: Bu değişkenler, sadece içinde tanımlı oldukları fonksiyon çalıştırıldığı sürece mevcutturlar ve sadece bu fonksiyon içinde erişilebilirdirler. Yine de, bazı tür değişkenler için yaşam süresi ve erişilebilirlik aynı anlama gelmez.

İlk Değer Ataması

Yerel bir değişken oluşturulduğunda derleyici buna ilk değer atamak için uğraşmaz. Yani, değişken rasgele bir değer ile başlar; bu değer 0 olabilir, fakat büyük ihtimalle başka bir değer olacaktır. Eğer değişkeninizin ilk değeri olmasını istiyorsanız, bunu, aşağıdaki gibi, açıkça sizin yapmanız gerekir:

```
int n = 33;
```

Böylece, değişkeniniz bu değer ile başlayacaktır.

Global Değişkenler

Bir başka değişken türü *global değişkenlerdir*. Yerel değişkenler bir fonksiyon içinde tanımlanırken, global değişkenler herhangi bir fonksiyonun dışında tanımlanır. (Ayrıca, herhangi bir sınıfın dışında da tanımlanır. Bunu daha sonra göreceğiz.) Bir global değişken bir dosya içindeki tüm fonksiyonlar tarafından erişilebilir. (Hatta diğer dosyalardan da erişilme potansiyeli vardır.) Daha net açıklayacak olursak, bir global değişken, program listesinde değişkenin tanımından sonra gelen tüm fonksiyonlara tarafından erişilebilirdir. Genellikle global değişkenlerinizin tüm fonksiyonlar tarafından erişilebilir olmasını istersiniz. Bu nedenle bu değişkenlerin deklarasyonlarını listenin en başına koyarsınız. Global değişkenlere kimi zaman *harici* (*external*) *değişkenler* de denir, çünkü bu değişkenler herhangi bir fonksiyonun dışında tanımlanır.

Aşağıda yer alan `EXTERN` isimli programda üç tane fonksiyonun hepsi bir global değişkene erişir.


```
// extern.cpp
// global degiskenler
#include <iostream>
using namespace std;
#include <conio.h>                  //getch()icin

char ch = 'a';                      //global degisken ch

void getachar();
void putachar();                    //fonksiyon deklarasyonları

int main()
   {
   while( ch != '\r' )              //main(), ch'e erisir
      {
      getachar();
      putachar();
      }
   cout << endl;
   return 0;
   }
//--------------------------------------------------------------
void getachar()                     //getachar(), ch'e erisir
   {
   ch = getch();
   }
//--------------------------------------------------------------
void putachar()                     //putachar(), ch'e erisir
   {
   cout << ch;
   }
```

`EXTERN` içindeki bir fonksiyon olan `getachar()`, klavyeden bir karakter okur. Bunun için bir kütüphane fonksiyonu olan `getch()`'i kullanıyor. `getch()` tıpkı `getche()` gibidir; yalnızca `getch()`,(isminin sonundaki `e` harfinin eksikliğinden anlaşılacağı gibi) okunan karakteri ekrana yazmaz (echo). İkinci bir `EXTERN` fonksiyonu olan `putachar()`, her karakteri ekranda gösterir. Elde edilen sonuç, ne yazdıysanız aynısı ekranda normal biçimde gösterilir:

```
I'm typing in this line of text
```

Bu programla ilgili önemli bir husus, `ch` değişkeninin fonksiyonların hiçbirinde tanımlanmış olmamasıdır. Bunun yerine, `ch` programın en başında, ilk fonksiyondan önce tanımlanır. Bu bir global (harici) değişkendir. Program listesinde `ch`'in tanımının adından gelen fonksiyonların tümü `ch`'e erişebilir – Bu örnekte `EXTERN`'in tüm fonksiyonları erişebilir: `main()`, `getachar()` ve `putachar()`. Yani, `ch`'in erişilebilirliği kaynak dosyanın tümüdür.

Global Değişkenlerin Rolü

Global bir değişken, bir program içinde birden fazla fonksiyon tarafından erişilmesi gerektiğinde kullanılır. Prosedürel programlarda global değişkenler genelde en önemli değişkenlerdir. Bununla birlikte, Bölüm 1'de vurguladığımız gibi, global değişkenler organizasyon problemleri de doğurabilir, çünkü bu değişkenler herhangi bir fonksiyon tarafından erişilebilir. Yanlış fonksiyonlar erişilebilir ya da fonksiyonlar hatalı olarak erişebilir. Nesne yönelimli bir programda global değişkenlere duyulan gereksinim de azdır.

İlk Değeri Ataması

Eğer bir global değişkene aşağıdaki gibi ilk değeri atanmışsa

```
int exvar = 199;
```

bu değer atama işlemi program ilk kez yüklendiğinde yapılır. Eğer bir global değişkene program tarafından açıkça ilk değer atanmamışsa – örneğin, şöyle tanımlanmışsa

```
int exvar;
```

bu global değişkene, ilk kez oluşturulurken otomatik olarak 0 değeri atanır. (Bu durum, yerel değişkenlerdekine benzemez. Yerel değişkenlere eğer ilk değer atanmamışsa, bu değişkenler oluşturulurken muhtemelen rasgele veya *anlamsız* bir değer içerirler.)

Yaşam Süresi ve Erişilebilirlik

Global değişkenler, statik depolama sınıfına sahiptir. Bu, şu anlama gelir: Global değişkenler programın yaşam süresi boyunca varlıklarını sürdürürler. Program başladığında global değişkenler için bir bellek alanı ayrılır ve bu bellek alanı program bitene kadar varlığını sürdürür. Global değişkenleri tanımlarken **static** anahtar kelimesini kullanmanıza gerek yoktur, çünkü bu değişkenlere zaten otomatik olarak statik depolama sınıfı verilir.

Vurguladığımız gibi, global değişkenler tanımlandıkları noktadan itibaren tüm dosya boyunca erişilebilirlerdir. Eğer **ch**, **main()**'in peşinden ama **getachar()**'dan önce tanımlanmış olsaydı, **getachar()** ve **putachar()** tarafından erişilebilir olacaktı, fakat **main()**'den erişilemeyecekti.

Statik Yerel Değişkenler

Şimdi başka bir tür değişkene bakalım: *Statik yerel değişken*. Ayrıca statik global değişkenler de vardır, fakat bunları kullanmak sadece, birden fazla dosya kullanan programlarda anlamlıdır. Birden fazla dosya kullanan programları ise Bölüm 13'e kadar incelemeyeceğiz.

Statik bir yerel değişken, otomatik yerel değişken ile aynı erişilebilirliğe sahiptir (yani, kendisinin tanımlı olduğu fonksiyon içinde.) Ancak, böyle bir değişkenin yaşam süresi, bir global değişkeninkiyle aynıdır; tek farkla, statik yerel değişken kendisini içeren fonksiyona yapılan ilk çağrıya kadar hayat bulmaz. Ondan sonra programın yaşam süresi boyunca varlığını sürdürür.

Bir fonksiyon çalışmıyorken de, yani fonksiyona yapılan çağrılar arasında, bir değer hatırlaması gerekiyorsa, bu gibi durumlarda statik yerel değişkenler işe yarar. Sıradaki örnekte, **getavg()** isminde bir fonksiyon sürekli olarak ortalama hesaplar. Fonksiyon, önceden ortalamasını bulduğu sayıların toplamını ve kaç tane olduklarını hatırlar. Fonksiyonu çağıran programdan argüman olarak ne zaman yeni bir sayı gelse, fonksiyon bu sayıyı da toplama ekler, sayacı bir artırır, toplamı sayaca bölerek yeni ortalamayı hesaplar ve bu yeni ortalamayı döndürür. **STATIC** isimli bu programın listesi işte şöyledir:

```
// static.cpp
// statik degiskenler
#include <iostream>
using namespace std;
float getavg(float);                    //deklarasyon
```



```
int main()
   {
   float data=1, avg;

   while( data != 0 )
      {
      cout << "Enter a number: ";
      cin >> data;
      avg = getavg(data);
      cout << "New average is " << avg << endl;
      }
   return 0;
   }
//-------------------------------------------------------------
// getavg()
// eski arti yeni verilerin ortalamasini hesaplar
float getavg(float newdata)
   {
   static float total = 0;
   static int count = 0;          // program icinde sadece bir kez
                                  // static degiskenlere ilk deger ataması

   count++;                       //sayaci (count) bir artir
   total += newdata;              //toplama yeni veriyi ekle
   return total / count;          //yeni ortalamayi dondur
   }
```

İşte bir etkileşim örneği:

```
Enter a number: 10
New average is 10   <=== total = 10, count = 1
Enter a number: 20
New average is 15   <=== total = 30, count = 2
Enter a number: 30
New average is 20   <=== total = 60, count = 3
```

getavg() içindeki statik değişkenler **total** ve **count**, **getavg()** fonksiyonu döndükten sonra da değerlerini korurlar. Böylece, fonksiyon bir kez daha çağrıldığında bu değişkenler hazır olurlar.

İlk Değer Ataması

Statik değişkenlere ilk değer ataması yaparken, **getavg()** fonksiyonundaki **total** ve **count** gibi, ilk değer atama işlemi sadece bir kez – değişkenlerin içinde bulunduğu fonksiyon ilk kez çağrıldığında – yapılır. Olağan yerel değişkenlerden farklı olarak, statik yerel değişkenlere fonksiyonun daha sonraki çağrılarında yeniden değer atanmaz.

Depolama

Eğer işletim sistemi mimarisine aşina iseniz, yerel değişkenlerin ve fonksiyon argümanlarının yığında (stack); global ve statik değişkenlerin katmanda (heap) üzerinde saklandığını bilmek belki ilginizi çekebilir.

Tablo 5.2 yerel, statik yerel ve global değişkenlerin yaşam süresi, erişilebilirlikleri ve diğer bazı özelliklerini özetliyor.

TABLO 5.1: Saklama Çeşitleri

	Yerel	Statik Yerel	Global
Erişilebilirlik	fonksiyon	fonksiyon	dosya
Yaşam süresi	fonksiyon	program	program
İlk değer	yok	0	0
Depolama Şekli	yığın	katman	katman
Amaç	Tek bir fonksiyon tarafından kullanılan değişkenler	Yerel ile aynı; fakat, fonksiyon sona erdiğinde değerini korur	Birkaç fonksiyon tarafından kullanılan değişkenler

Referans ile Dönmek

Global değişkenleri öğrendiğimize göre, artık biraz daha tuhaf görünen bir C++ özelliğini inceleyebiliriz. Değerleri fonksiyonlara referans olarak aktarmanın yanı sıra bir fonksiyondan bir değeri de referans olarak döndürebilirsiniz. Niye böyle bir şey yapmak isteyebileceğiniz anlaşılmaz gelebilir. Bunun bir sebebi, büyük nesneleri kopyalamayı önlemektir. Bu konuyu, "Sanal Fonksiyonlar" adlı Bölüm 11'de göreceğiz. Diğer bir neden ise, bu sayede eşittir işaretinin sol tarafında fonksiyon çağrısı yapabiliyor olmanızdır. Bu biraz tuhaf bir kavramdır; bu nedenle, şimdi bunu bir örnek üzerinde görelim. **RETREF** programı bu mekanizmayı gösterir.

```
// retref.cpp
// referans degerlerini dondurmek
#include <iostream>
using namespace std;
int x;                          //global degisken
int& setx();                    //fonksiyon deklarasyonu

int main()
   {                            //sol taraftaki fonksiyon cagrisini kullanarak
   setx() = 92;                 //x'e bir deger ata
   cout << "x=" << x << endl;   //x'in icindeki yeni degeri ekranda goster
   return 0;
   }
//-------------------------------------------------------------
int& setx()
   {
   return x;                    //degistirilecek degeri dondurur
   }
```

Bu programda **setx()** fonksiyonu, döndürülen tip bir referans tipi, **int&**, olacak şekilde tanımlanır:

```
int& setx();
```

Bu fonksiyon şu ifadeyi içerir:

```
return x;
```


Burada kullanılan x global değişken olarak tanımlamıştı. Şimdi – işte burası çok acayip görünür – eşittir işaretinin sol tarafında bu fonksiyona bir çağrı yerleştirilir:

```
setx()= 92;
```

Bu ifadenin sonucu şudur: Fonksiyondan dönen değişkene eşittir işaretinin sağındaki değer atanır. Yani, x'e 92 değeri verilir. Programın çıktısı, böyle bir atamanın gerçekleştirildiğini doğrular:

```
x=92
```

Eşittir İşaretinin Sol Tarafında Fonksiyon Çağrıları

Bu hâlâ tuhaf mı geliyor? Bir değer döndüren sıradan bir fonksiyon, sanki kendisi bir değermiş gibi kullanılabilir:

```
y=squareroot(x);
```

Burada, **squareroot(x)** hangi değere sahipse (mesela, 27.2), bu değer y'ye atanır. Fonksiyon, sankı kendisi salt bir değermiş gibi ele alınır. Diğer yandan, referans döndüren bir fonksiyon ise bir değişken gibi ele alınır. Böyle bir fonksiyon bir değişkeni *takma bir isim* (*alias*) ile döndürür; bu değişken fonksiyonun dönüş ifadesinde yer alan değişkendir. **RETREF.CPP**'de **setx()** fonksiyonu x değişkenine bir referans döndürür. Bu fonksiyon çağrıldığında, fonksiyon sanki x değişkeniymiş gibi ele alınıyor. Bu nedenle, eşittir işaretinin sol tarafında kullanılabiliyor.

Bundan çıkarılabilecek iki sonuç vardır. Birincisi; referans döndüren bir fonksiyondan sabit bir değer döndüremezsiniz. **setx()** içinde şunu yazamazsınız:

```
int& setx()
   {
   return 3;
   }
```

Bunu denemeye kalkarsanız, derleyici *lvalue* gerektiğine dair sizi uyarır. *lvalue* (soldaki değer), eşittir işaretinin sol tarafına yazılabilen bir şeydir. Yani, bir değişkendir, sabit bir değer olamaz.

Daha ustaca söylemek gerekirse, yerel bir değişkene referans döndüremezsiniz:

```
int& setx()
   {
   int x = 3;
   return x;   //hata
   }
```

Bunda yanlış olan nedir? Problem şu: Bir fonksiyonun yerel değişkenleri, fonksiyon döndükten sonra muhtemelen yok edilir. Bu nedenle, sonradan var olmayacak bir şeye referans döndürmenin bir manası yoktur.

Henüz Telaşlanmayın

Elbette, niye bir kimse eşittir işaretinin sol tarafında bir fonksiyon çağrısı kullanmak ister sorusu henüz cevaplanmadı. Prosedürel programlamada bu teknik muhtemelen çok fazla kullanılmıyordur. Yukarıdaki örnekte olduğu gibi, aynı sonuca ulaşmak için daha kolay yöntemler var. Bununla birlikte, "Operatörlerin Aşırı Yüklenmesi"adlı Bölüm 8'de, referans döndürmenin vazgeçilmez bir teknik olduğunu göreceğiz. O zamana kadar, bunu aklınızın bir köşesinde saklayın.

const Fonksiyon Argümanları

Bir argümanı referans olarak aktarma işleminin, fonksiyonu çağıran programdaki bir değişkeni, fonksiyonun değiştirmesine imkan vermek amacıyla kullanıldığını görmüştük. Bununla birlikte, referans olarak aktarmanın diğer nedenleri de var. Birinci neden; verimliliktir. Fonksiyon argümanı olarak kullanılan kimi değişkenler çok büyük olabilir; büyük bir yapı buna bir örnek olabilir. Eğer argüman büyükse, referans olarak aktarmak çok daha verimli olacaktır, çünkü sahnenin gerisinde değişkenin tümünün aktarılması yerine, sadece bir adres gerçekte aktarılır.

Diyelim ki, verimlilik açısından bir argümanı referans olarak geçirmek istiyorsunuz. Fakat, hem fonksiyonun bu değişkeni değiştirmesini istemiyorsunuz, hem de fonksiyonun onu *değiştiremeyeceğini* garanti etmek istiyorsunuz.

Böyle bir garantiyi sağlamak için fonksiyon deklarasyonunda değişkenin önüne **const** niteleyicisini kullanabilirsiniz. **CONSTARG** programı bunun nasıl göründüğünü gösteriyor.

```
// constarg.cpp
// sabit fonksiyon argumanlari

void aFunc(int&a,const int&b);      //deklarasyon

int main()
   {
   int alpha = 7;
   int beta = 11;
   aFunc(alpha, beta);
   return 0;
   }
//--------------------------------------------------------------
void aFunc(int& a, const int& b)    //tanim
   {
   a = 107;          //tamam
   b = 111;          //hata: sabit bir arguman degistirilemez
   }
```

Burada **aFunc()** fonksiyonunun **beta** değişkenini değiştiremeyeceğinden emin olmak istiyoruz. (**alpha**'yı değiştirip değiştirmeyeceği bizi ilgilendirmiyor.) Bu nedenle, fonksiyon deklarasyonunda (ve tanımında) **const** niteleyicisini **beta** değişkeni ile kullanıyoruz:

```
void aFunc(int& alpha,const int& beta);
```

Artık **aFunc()** içinde **beta**'yı değiştirme girişimi derleyici tarafından hata olarak değerlendirileceklir. C++'ın tasarım felsefelerinden biri, hataların program çalışırken ortaya çıkmasını beklemektense derleyicinin hataları bulmasının daha iyi bir yaklaşım olduğudur. **const** fonksiyon argümanlarının kullanımı bu yaklaşımın uygulanışına bir örnektir.

Bir **const** değişkeni bir fonksiyona referans olarak geçirmek istiyorsanız, seçeneğiniz yoktur: Bu değişken fonksiyon deklarasyonunda **const** olarak tanımlanmak *zorundadır*. (Bir **const** argümanı değer olarak geçirme işleminde bir problem yok, çünkü fonksiyon orijinal değişkeni zaten hiçbir şekilde değiştiremez.)

Pek çok kütüphane fonksiyonu sabit argümanları aynı şekilde kullanır. İlerledikçe örneklerini göreceğiz.

Özet

Fonksiyonlar programları düzenlemeye yardımcı olan yöntemlerden biridir. Fonksiyonlar, ayrıca, bir kod bloğuna bir isim vererek ve bu bloğun programın diğer bölümlerinde de çalıştırılmasını sağlayarak programın boyutunu küçültmeye yardımcı olurlar. Fonksiyon *deklarasyonları* (prototipler) fonksiyonun nasıl göründüğünü açıkça belirtir. Fonksiyon *çağrıları*, kontrolü fonksiyona aktarır. Fonksiyon *tanımları* ise fonksiyonu oluşturan ifadeleri içerir. Fonksiyon *deklaratörü*, tanımın birinci satırıdır.

Argümanlar fonksiyona *değer* veya *referans* olarak gönderilebilir. Argümanlar değer olarak gönderilirse, fonksiyon argümanın bir kopyası üzerinde çalışır. Referans olarak gönderildiğinde ise fonksiyon, kendisini çağıran programdaki orijinal argüman üzerinde çalışır.

Fonksiyonlar sadece tek bir değer döndürebilir. Fonksiyonlar, alışılmış biçimde, değer döndürür; ancak, referans da döndürebilir. Fonksiyonun referans döndürmesi, fonksiyon çağrısının bir değer atama ifadesinde sol tarafta kullanılmasına imkan verir. Argümanlar ve dönen değerler basit veri tipleri veya yapılar olabilir.

Aşırı yüklenen (overloaded) bir fonksiyon aslında, isimleri aynı olan fonksiyonlar topluluğudur. Fonksiyon çağrıldığında hangisinin çalıştırılacağı, fonksiyon çağrısında belirtilen argümanların tipine ve sayısına bağlıdır.

Bir *yerel (inline)* fonksiyon, kaynak dosyasındaki normal bir fonksiyona benzer, fakat fonksiyonun kodunu doğrudan fonksiyonu çağıran programına içine ekler. Yerel fonksiyonlar normal fonksiyonlardan daha hızlı çalışırlar. Ancak, eğer çok küçük değillerse, normal fonksiyonlardan çok daha fazla bellek gerektirebilirler.

Eğer bir fonksiyon varsayılan (default) argümanlar kullanıyorsa, bu fonksiyona yapılan çağrılar deklarasyonda yer alan argümanların tümünü içermek zorunda değildir. Eksik olan argümanlar için fonksiyonun sağladığı varsayılan değerler kullanılır.

Değişkenler *depolama sınıfı* denilen bir özelliğe sahiptir. En çok kullanılan depolama sınıfı *otomatik* olandır. Yerel değişkenler otomatik depolama sınıfına sahiptir: Sadece içinde tanımlı oldukları fonksiyon çalıştığı sürece bu değişkenler var olabilir ve sadece bu fonksiyonun içinde erişilebilirler. *Global* değişkenler statik depolama sınıfına sahiptir: Programın yaşam süresi boyunca varlıklarını sürdürürler. Ayrıca bütün bir dosya boyunca erişilebilirler. *Statik* yerel değişkenler de programın yaşam süresi boyunca varlıklarını sürdürürler, fakat sadece kendi fonksiyonları içinde erişilebilirler.

Bir fonksiyon, **const** niteleyicisi ile belirtilen argümanlarının hiçbirini değiştiremez. Fonksiyonu çağıran programda önceden **const** olarak tanımlanmış bir değişken, bir **const** argüman olarak fonksiyona geçirilmelidir.

Bölüm 4'te nesnelerin iki ana parçasından biri olan yapıları gözden geçirdik. Bu bölümde ise ikinci parça olan fonksiyonları inceledik. Artık bu iki bileşeni bir araya getirip, Bölüm 6'nın konusu olan nesneleri oluşturmaya hazırız.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Bir fonksiyonun tek en önemli görevi
   a. bir kod bloğuna bir isim vermektir.
   b. programın boyutunu küçültmektir.
   c. argümanları kabul etmek ve bir dönüş değeri sağlamaktır.
   d. bir programı kavramsal birimler bazında düzenlemeye yardımcı olmaktır.
2. Bir fonksiyonun kendisi fonksiyonun t_____dır.
3. `foo` kelimesini ekranda görüntüleyen `foo()` isminde bir fonksiyon yazın.
4. Bir satırlık fonksiyon tanımı, fonksiyonun d_____ veya p_____dir.
5. Fonksiyonun yapması gereken işi gerçekleştiren ifadeler fonksiyonun _____ oluşturur.
6. Bir fonksiyonu hayata geçiren program ifadesine fonksiyon _____ denir.
7. Fonksiyon tanımının ilk satırı _____ olarak adlandırılır.
8. Bir fonksiyonun argümanı,
   a. fonksiyonu çağıran programdan bir değer alan, fonksiyon içindeki bir değişkendir.
   b. fonksiyonun, fonksiyonu çağıran programın değerlerini kabul etmemek için direnme yollarından biridir.
   c. fonksiyonu çağıran program tarafından fonksiyona gönderilen bir değerdir.
   d. fonksiyonu çağıran programa fonksiyonun döndürdüğü değerdir.
9. Doğru/Yanlış: Argümanlar değer olarak aktarıldığı zaman fonksiyon, kendisini çağıran programdaki orijinal argümanlarla çalışır.
10. Bir fonksiyon deklarasyonunda argüman isimleri kullanmanın amacı nedir?
11. Aşağıdakilerden hangisi bir fonksiyona kurallara uygun olarak aktarılabilir?
    a. Bir sabit.
    b. Bir değişken.
    c. Bir yapı.
    d. Bir başlık dosyası.
12. Bir fonksiyon deklarasyonunda yer alan içi boş parantezlerin önemi nedir?
13. Bir fonksiyondan kaç tane değer döndürülebilir?
14. Doğru/Yanlış: Bir fonksiyon bir değer döndürdüğünde, fonksiyon çağrısının tümü eşittir işaretinin sağ tarafında yer alabilir ve başka bir değişkene değer olarak atanabilir.
15. Bir fonksiyonun dönüş tipi nerede belirtilir?
16. Hiçbir şey döndürmeyen bir fonksiyonun dönüş tipi _____dur.
17. İşte bir fonksiyon:

```
int times2(int a)
  {
  return (a*2);
  }
```

Bu fonksiyonu çağırmak için gerekli olan her şeyi içeren bir `main()` programı yazın.

18. Bir argüman referans olarak aktarıldığında
    a. argümanın değerini saklamak için fonksiyonun içinde bir değişken oluşturulur.
    b. fonksiyon argümanın değerine erişemez.

   c. argümanın değerini saklamak için fonksiyonu çağıran program içinde geçici bir değişken oluşturulur.
   d. fonksiyon, argümanın kendisini çağıran programdaki orijinal değerine erişir.
19. Argümanları referans olarak aktarmanın başlıca sebebi nedir?
20. Aşırı yüklenen (overloaded) fonksiyonlar
    a. isimleri aynı olan bir grup fonksiyondur.
    b. aynı argüman sayısına ve tipine sahiptir.
    c. programcılar için hayatı kolaylaştırır.
    d. stres yüzünden beklenmedik zamanlarda çökebilir.
21. `bar()` isminde iki tane aşırı yüklenmiş fonksiyon için deklarasyon yazın. Her ikisi de `int` tipinde değer döndürsün. Birincisi `char` tipinde bir tane argüman; ikincisi, `char` tipinde iki tane argüman alsın. Eğer bu imkansız ise, nedenini açıklayın.
22. Genel olarak, bir yerel (inline) fonksiyon normal bir fonksiyondan daha _____ çalışır, fakat daha _____ bellek gerektirir.
23. `float` tipinde bir tane argüman alan ve `float` tipinde bir değer döndüren `foobar()` isminde bir yerel fonksiyon için bir deklaratör yazın.
24. Bir varsayılan argümanın değeri hakkında aşağıdakilerden hangisi söylenebilir?
    a. Fonksiyonu çağıran program tarafından sağlanabilir.
    b. Fonksiyon tarafından sağlanabilir.
    c. Sabit bir değere sahip olmalıdır.
    d. Değişken bir değere sahip olmalıdır.
25. İki argüman alan ve `char` tipinde bir değer döndüren `blyth()` isminde bir fonksiyon için bir deklarasyon yazın. İlk argüman `int` tipinde; ikincisi `float` tipinde ve 3.14159 varsayılan değerine sahip olmalıdır.
26. Kapsam (scope) ve depolama sınıfı, bir değişkenin _____ ve _____ ile ilgilidir.
27. Bir global değişkene, bu değişkenle aynı dosya içinde olan fonksiyonlardan hangileri erişebilir?
28. Bir yerel değişkene hangi fonksiyonlar erişebilir?
29. Bir statik yerel değişken hangi amaç için kullanılır?
    a. Bir değişkeni birkaç fonksiyon tarafından erişilebilir kılmak için.
    b. Bir değişkeni sadece bir fonksiyon tarafından erişilebilir kılmak için.
    c. Fonksiyonun çalışmadığı zamanlarda belleği korumak için.
    d. Fonksiyonun çalışmadığı zamanlarda bir değeri korumak için.
30. Eğer bir fonksiyon bir değeri referans olarak döndürüyorsa, fonksiyon çağrısını hangi alışılmadık yere yerleştirebilirsiniz?

Alıştırmalar

Yıldızlı soruların cevaplarını Ek G'de bulabilirsiniz.

1. "C++ Programlama Esasları" adlı Bölüm 2'de ele alınan `CIRCAREA` programına bakın. Bir dairenin alanını aynı şekilde bulan `circarea()` isminde bir fonksiyon yazın. Fonksiyon, `float` tipinde bir argüman almalı ve aynı tipte bir argüman döndürmelidir. Sonra, kullanıcıdan yarıçap değeri alan, `circarea()` fonksiyonunu çağıran ve sonucu ekranda gösteren bir `main()` fonksiyonu yazın.*

2. Bir `n` sayısının `p` kuvveti, `n`'yi `p` kez kendisiyle çarpmaya eşittir. `power()` isminde bir fonksiyon yazın. Bu fonksiyon `n` için `double` ve `p` için bir `int` değeri almalı ve sonucu `double` değerinde döndürmelidir. `p` için varsayılan değer olarak 2 kullanın. Böylece, eğer argüman dahil edilmemişse, `n` sayısının karesi alınsın. Bu fonksiyonu test etmek için gerekli değerleri kullanıcıdan alan bir `main()` fonksiyonu yazın.*
3. Referans olarak iki argümanı alan ve bu iki sayının küçüğüne 0 değeri atayan `zeroSmaller()` isminde bir fonksiyon yazın. Bu fonksiyonu kullanan bir `main()` programı yazın.*
4. Argüman olarak iki `Distance` değeri alan ve bunlardan büyük olanı döndüren bir fonksiyon yazın. Kullanıcıdan iki `Distance` değeri alan, bunları karşılaştıran ve büyük olanı görüntüleyen bir `main()` programını da dahil edin.*
5. Saat, dakika ve saniye için üç tane `int` değerini argüman olarak alan ve buna denk gelen zamanı saniye cinsinden (`long` tipinde) döndüren `hms_to_secs()` isminde bir fonksiyon yazın. Bu fonksiyonu tekrar tekrar çağıran bir program yazın. Programınız kullanıcıdan saat, dakika ve saniye bilgilerini (12:59:59 formatında) alsın, fonksiyonu çağırsın ve sonuç olarak toplam saniyeyi ekranda göstersin.
6. "Yapılar" başlıklı Bölüm 4'te Alıştırma 11'deki, `struct time` tipinde iki değeri toplayan programla başlayın. Programın işlevi aynı kalsın, fakat programı iki fonksiyon kullanacak şekilde değiştirin. Birinci fonksiyon `time_to_secs()`, `time` tipinde sadece tek bir argüman içersin ve karşılık gelen zamanı saniye cinsinden (`long` tipine) döndürsün. İkinci fonksiyon `secs_to_time()`, sadece tek bir argüman olarak saniye cinsinden (`long` tipinde) saati alsın ve `time` tipinde bir yapı döndürsün.
7. Alıştırma 2'deki sadece `double` tipiyle işlem yapan `power()` fonksiyonu ile başlayın. Aynı isme sahip bir dizi 'aşırı yüklenmiş' (overloaded) fonksiyon oluşturun. Bu fonksiyonlar `double`'a ilaveten `char`, `int`, `long` ve `float` tipleriyle de çalışsın. Bu fonksiyonları bütün argüman tipleri ile kullanan bir `main()` programı yazın.
8. İki `int` değerinin yerlerini değiştiren `swap()` isimli bir fonksiyon yazın. Tamsayılar fonksiyona, kendisini çağıran program tarafından gönderilir. (Dikkat ederseniz, bu fonksiyon kendisini çağıran programdaki değişkenlerin değerlerini değiştirir, fonksiyonun içindekileri değil.) Argümanları ne şekilde aktarmanız gerektiğine karar vermeniz gerekir. Bu fonksiyonu kullanan bir `main()` programı oluşturun.
9. Alıştırma 8'i tekrarlayın. Fakat bu kez iki `int` değişken yerine `swap()` fonksiyonunun iki `struct time` değerini değiştirmesini sağlayın (Alıştırma 6'ya bakın).
10. Kaç kere çağrıldığını ekrana mesaj olarak yazan bir fonksiyon yazın. Örneğin, "Ben 3 kez çağrıldım" gibi. Bu fonksiyonu en az 10 kez çağıran bir `main()` programı yazın. Bu fonksiyonu iki değişik yoldan uygulamaya çalışın. Önce, çağrılma sayısını tutması için bir global değişken kullanın. İkinci olarak, yerel statik değişken kullanın. Hangisi daha uygun? Niye yerel bir değişken kullanamıyorsunuz?
11. Bölüm 4'te Alıştırma 10'daki `sterling` yapısına dayanan ve kullanıcıdan eski-stil İngiliz biçiminde (£9:19:11) iki para miktarı alan bir program yazın. Program bu paraları toplasın ve sonucu yine eski stil İngiliz biçiminde ekranda göstersin. Üç tane fonksiyon kullanın. Birinci fonksiyon kullanıcıdan pound-shilling-pence değerini almalı ve bu değeri `sterling` tipinde bir yapı olarak geri döndürmelidir. İkinci fonksiyon `sterling` tipinde iki argüman almalı ve argümanların toplamını simgeleyen aynı tipte bir yapı döndürmelidir. Üçüncü fonksiyon `sterling` tipinde bir argüman almalı ve bunun değerini ekranda göstermelidir.

12. Bölüm 4'te Alıştırma 12'deki dört fonksiyonlu ondalık hesap makinesini tekrar gözden geçirin. Böylece, program dört aritmetik işlemin her biri için bir fonksiyon kullansın. Bu fonksiyonlar **fadd()**, **fsub()**, **fmul()** ve **fdiv()** olarak adlandırılabilir. Bu fonksiyonların her biri **struct fraction** tipinde iki argüman almalı ve aynı tipte bir argüman döndürmelidir.

ALTINCI BÖLÜM

NESNELER VE SINIFLAR

Nihayet beklediğiniz konulara geldik: Nesneler ve sınıflar. Artık temel bilgileri aldık. Veri unsurlarını gruplamaya yarayan yapıları öğrendik. Programın içindeki işlemleri kendi isimleriyle çağrılabilen parçalara bölmemize yarayan fonksiyonları inceledik. Kitabın bu bölümünde de tüm bu kavramları kullanarak sınıfları tanımlayacağız. Basitlerinden başlayarak çeşitli sınıfları tanıyacağız. Daha sonra nispeten karmaşık sınıfları göreceğiz. Öncelikle sınıf ve nesnelerin ayrıntılarına eğileceğiz. Bölümün sonunda ise daha geniş bir perspektiften bakarak nesne yönelimli programlama yaklaşımının bize ne fayda sağlayacağını ele alacağız.

Bu bölümü okurken "Genel Görünüm" adını verdiğimiz Bölüm 1'deki kavramları yeniden gözden geçirmeniz yararlı olabilir.

Basit Bir Sınıf

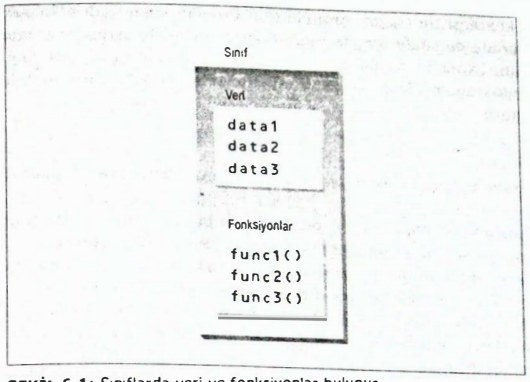
İlk programımızda bir sınıf ve o sınıfa ait iki nesne var. Basit olmakla birlikte bu program C++ sınıflarının sözdizimini ve genel özelliklerini gösterir. **SMALLOBJ** adını verdiğimiz programın kaynak kodu şöyle:

```
// smallobj.cpp
// kucuk ve basit bir nesne ornegi
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class smallobj                  //bir sinif tanimla
   {
   private:
      int somedata;             //sinif verisi
   public:
      void setdata(int d)       //veriyi atayan uye fonksiyon
         { somedata = d; }
      void showdata()           //veriyi gosteren uye fonksiyon
         { cout << "Data is " << somedata << endl; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   smallobj s1,s2;              //smallobj sinifindan iki nesne tanimla

   s1.setdata(1066);            //veriyi atamak icin uye fonksiyonu cagir
   s2.setdata(1776);

   s1.showdata();               //veriyi gostermek icin uye fonksiyonu cagir
   s2.showdata();
   return 0;
   }
```

Bu programda tanımlanan **smallobj** sınıfında bir veri öğesi, iki de üye fonksiyon vardır.

Üye fonksiyonlar, sınıfın dışında veri öğelerine ulaşmanın tek yolunu oluştururlar. İlk üye fonksiyon, veri öğesine bir değer atar. İkinci üye fonksiyon da bu değeri göstermeye yarar. (Bu ifadeler biraz karışık gelmiş olabilir; ilerledikçe bu terimlerin anlamını daha iyi kavrayacağız.)

Veri ve fonksiyonları tek bir bütün haline getirmek nesne yönelimli programlamanın temel yaklaşımıdır. Şekil 6.1 bunu açıklar.



ŞEKİL 6.1: Sınıflarda veri ve fonksiyonlar bulunur.

Sınıflar ve Nesneler

Bölüm 1'den de hatırlayacağız üzere, bir nesne ile sınıf arasındaki ilişki tıpkı bir değişken ile veri tipi arasındaki ilişki gibidir. Bir nesnenin, kendi sınıfının bir *örneği* (*instance* - *kopya*) olduğu söylenir. Aynı benim 1954 model Chevrolet'min, o markanın bir örneği olması gibi... **SMALLOBJ** örneğinde adı **smallobj** olan sınıf, programın ilk kısmında tanımlanmıştır. Daha sonra programın ana kısmında, yani **main()**'de bu sınıftan **s1** ve **s2** adlı iki örnek daha tanımlanır.

Bu nesnelerin her birine veri atanır. Sonra nesneler kendi değerlerini gösterirler. Programın çıktısı şöyledir:

```
Data is 1066 <==== bunu s1 nesnesi gosterir
Data is 1776 <==== bunu s2 nesnesi gosterir
```

Öncelikle, programın ilk kısmını – **smallobj** sınıfının tanımlandığı bölümü – ayrıntılı olarak inceleyeceğiz. Daha sonra da **main()**'in bu sınıfın nesneleri üzerinde neler yaptığına bakacağız.

Sınıfı Tanımlamak

SMALLOBJ'in kaynak kodundan aldığımız, **smallobj** sınıfının tanımı (bazen tanım yerine *belirteç* de denir) şöyledir:

```
class smallobj                  //bir sinif tanimla
   {
   private:
      int somedata;             //sinif verisi
   public:
      void setdata(int d)       //veriyi atayan uye fonksiyon
         { somedata = d; }
      void showdata()           //veriyi gosteren uye fonksiyon
         { cout << "Data is " << somedata << endl; }
   };
```

Tanım, `class` anahtar kelimesi ile başlar. Ardından bu örnekte sınıfın adı olan `smallobj` gelir. Yapıda olduğu gibi burada da sınıfın gövdesi küme parantezleriyle ayrılır ve sonunda da noktalı virgül ile tamamlanır. (Noktalı virgülü unutmayın. Yapı ve sınıflar gibi veri yapılarının noktalı virgülle bittiğini; fonksiyon ve döngü gibi kontrol yapılarının sonunda ise noktalı virgül kullanılmadığını hatırda tutun.)

private ve public

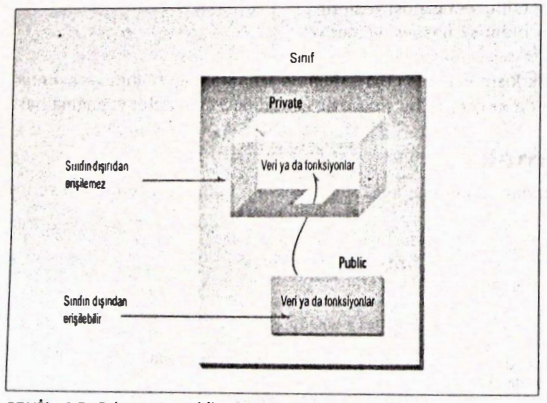
Sınıf bloğunda henüz öğrenmediğimiz iki anahtar kelime yer alıyor: `private` ve `public`. Peki bunlar ne işe yarıyor?

Nesne yönelimli programlamanın en önemli özelliklerinden biri *veri saklama*dır (*data hiding*). Saklama derken, paranoya sahibi programcıların davranışlarından söz etmiyoruz. Burada kastedilen nokta, veriye kendi sınıfının dışında başka fonksiyonlar tarafından kazara erişilmesini önlemek amacıyla veri saklanmasıdır. Veriyi saklamanın en başta gelen yöntemi, veriyi bir sınıfın içine koyup sonra da private (özel) olarak tanımlamaktır. Private olan veri ya da fonksiyonlara, sadece kendi sınıfları içinden erişilebilir. Public (genel, açık) olan veri ya da fonksiyonlara ise kendi sınıfları dışından da erişmek mümkündür. Şekil 6.2 bunu açıklar.

Kimden Saklıyoruz?

Veri saklamayı bilgisayar veri tabanlarının korunması için kullanılan güvenlik teknikleri ile karıştırmayın. Güvenlik sağlarken örneğin, bir kullanıcının belli bir veri tabanına girmeden önce bir şifre yazmasını isteyebilirsiniz. Burada şifrenin amacı yetkisiz ya da kötü niyetli kullanıcıların verileri bozmasını (hatta bazen verileri okumasını bile) önlemektir.

Veri saklama ise veriyi, programın içinde ona ulaşması gerekmeyen bölümlerden gizlemekten ibarettir. Daha da netleştirirsek, amacımız bir sınıfın verisini diğer sınıflardan saklamaktır. Veri saklama, iyi niyetli programcıları masum hatalardan koruma amacına yöneliktir. Private verilere erişmeyi çok isteyen programcılar bunu yapmanın bir yolunu bulabilirler. Ama burada önemli olan private veriye kazara erişmenin zorlaştırılmış olmasıdır.



ŞEKİL 6.2: Private ve public.

Sınıf Verisi

Örnekteki `smallobj` sınıfında `int` tipinden bir veri öğesi, yani `somedata` var. Bir sınıfın içindeki veri öğelerine *veri üyeleri* (bazen de *üye veriler*) denir. Herhangi bir yapıda istenen sayıda veri öğesi olabildiği gibi bir sınıfta da istendiği kadar veri üyesi olabilir. Burada `somedata` adlı veri üyesinin önünde `private` anahtar kelimesi olduğu için söz konusu veri üyesine sadece kendi sınıfı içinden erişilebilir.

Üye Fonksiyonlar

Üye fonksiyonlar, bir sınıfın içinde yer alan fonksiyonlardır. (Smalltalk gibi bazı nesne yönelimli dillerde üye fonksiyonlara *yöntem* (method) denir. Bazı yazarlar bu deyimi C++'ta da kullanırlar.) Örneğimizdeki `smallobj`'de iki üye fonksiyon vardır: `setdata()` ve `showdata()`. Bu fonksiyonların gövdeleri küme parantezleriyle aynı satırlara yazılmıştır. Fonksiyon tanımlarını şu şekilde, daha geleneksel bir tarzda da yapabilirsiniz:

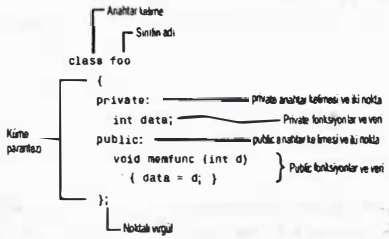
```
void setdata(int d)
   {
   somedata = d;
   }

ve

void showdata()
   {
   cout << "\nData is " << somedata;
   }
```

Ancak, üye fonksiyonlar küçük olduğunda yerden tasarruf etmek için bunların tanımlarını kısa şekilde yapmak sık tercih edilen bir yöntemdir.

`setdata()` ve `showdata()`'dan önce `public` anahtar kelimesi yer aldığı için bu fonksiyonlara sınıf dışından da erişilebilir. Bunun nasıl yapıldığını birazdan göreceğiz. Sınıf tanımının söz dizimi Şekil 6.3'te görülüyor.

Fonksiyonlar public, Veriler private

Genelde bir sınıftaki veriler private, fonksiyonlar public'tir. Bu, sınıfların kullanım tarzından kaynaklanmaktadır. Veriler, üzerlerinde kazara oynanmasın diye private olarak tanımlanırlar. Veri üzerinde işlem yapan fonksiyonlar ise sınıf dışından erişilebilsinler diye public olarak tanımlanırlar. Ancak, verilerin private, fonksiyonların public olarak tanımlanmaları bir kural değildir. Bazı durumlarda private fonksiyonlar ve public veriler kullanmanız gerekebilir.



ŞEKİL 6.3: Sınıf tanımının söz dizimi.

Sınıf Tanımı İçindeki Üye Fonksiyonlar

`smallobj` sınıfının içindeki üye fonksiyonlar gayet sık rastlanan türden işlemler yaparlar: Bir sınıfın içindeki verileri atamaya ve göstermeye yararlar. `setdata()` fonksiyonu `somedata`'da tutulan değeri gösterir. Bu fonksiyona parametre olarak geçilen değer, değişkene atanmaz.

Üye fonksiyonlar olan `setdata()` ve `showdata()`üye fonksiyonlarının aslında birer tanımdan ibaret olduğuna, fonksiyonların kodlarının sınıf tanımı içinde yer aldığına dikkat edin. (Fonksiyonların kendileri şu açıdan tanım sayılmaz: O sınıftan bir nesne oluşturulana kadar fonksiyon kodu için bellek tahsis edilmez.) Bu şekilde bir sınıfın içinde tanımlanan üye fonksiyonlar otomatik olarak yerel (inline) fonksiyon şeklinde oluşturulur. (Yerel fonksiyonlar "Fonksiyonlar" ana başlıklı Bölüm 5'te işlemiştik.) İleride bir fonksiyonu bir sınıf içinde *deklare edip* başka bir yerde *tanımlamanın* mümkün olduğunu göreceğiz. Sınıfı dışında tanımlanan fonksiyonlar genelde yerel değildir.

Sınıfın Kullanımı

Sınıfı tanımladıktan sonra `main()`'in bunu nasıl kullandığını göreceğiz. Nesnelerin nasıl tanımlandığını ve tanımlandıktan sonra nesnelerin üye fonksiyonlarına nasıl erişildiğini inceleyeceğiz.

Nesnelerin Tanımlanması

`main()`'deki ilk ifade şöyledir:

```
smallobj s1,s2;
```

Burada `smallobj` sınıfından `s1` ve `s2` adlı iki nesne tanımlanır. Hatırlarsanız, `smallobj` sınıfının tanımı ile herhangi bir nesne oluşturulmaz. Bu ifade sadece söz konusu sınıfın, oluşturulduğu zaman neye benzeyeceğini tanımlar. Tıpkı bir yapı tanımının herhangi bir yapı değişkeni oluşturmadan sadece o yapının şeklini belirlemesi gibi... Program işlemleri nesnelerin katılımıyla gerçekleştirilir. Nesne tanımlamak, herhangi bir veri tipinden değişken tanımlamaya benzer: Bellekte bunun için yer ayrılır.

Nesneleri bu şekilde tanımlamak demek o nesneleri oluşturmak demektir. Bu işleme *örnek oluşturmak* (*instantiating*) da denir. Çünkü bu şekilde sınıfın bir örneği (*instance*) oluşturulur. Bir nesne, bir sınıfın özel bir örneğidir. Nesnelere bazen *örnek değişkenler* (*instance variables*) adı da verilir.

Üye Fonksiyonların Çağrılması

Program üzerinden devam edersek, `main()`'de sonraki iki ifade, `setdata()` üye fonksiyonunu çağırır:

```
s1.setdata(1066);
s2.setdata(1776);
```

Bu ifadeler normal fonksiyon çağrılarına benzemiyor. Neden `s1` ve `s2` gibi nesne isimleri, fonksiyon isimlerine nokta ile birleştiriliyor?

Bu tuhaf söz dizimi *belli bir nesne ile ilişkili olan* bir üye fonksiyonu çağırmak için kullanılır. `setdata()`, `smallobj` sınıfının bir üye fonksiyonu olduğu için her zaman bu sınıfın bir nesnesi ile bağlantılı olarak çağrılmalıdır. Tek başına şöyle yazmak anlamsız olur:


```
setdata(1066);
```

Çünkü bir üye fonksiyon her zaman belli bir nesnenin üzerinde işlem yapmak için çağrılır; sınıfın geneli üzerinde çalışmak için değil. Sınıfa bu şekilde erişmeye çalışmak tıpkı bir otomobilin proje çizimlerini hareket ettirmeye çalışmak gibi anlamsız olur. Tek problem, bu ifadenin anlamdan yoksun olması değildir. Denemeye kalkışırsanız böyle bir ifadeye derleyicinin hata mesajı verdiğini görürsünüz. Bir sınıfın üye fonksiyonlarına sadece o sınıfın bir nesnesi tarafından erişilebilir. Bir üye fonksiyonu kullanmak için, nokta operatörü, nesne ismi ile üye fonksiyonu birbiriyle ilişkilendirir. Burada kullanılan söz dizimi, yapı üyeleri için kullandığımız söz dizimine benzer. Ancak, parantezler, bir veri parçasına değil de bir üye fonksiyona erişmekte olduğumuzu gösterir. (Nokta operatörüne *sınıf üyesi erişim operatörü* de denir.)

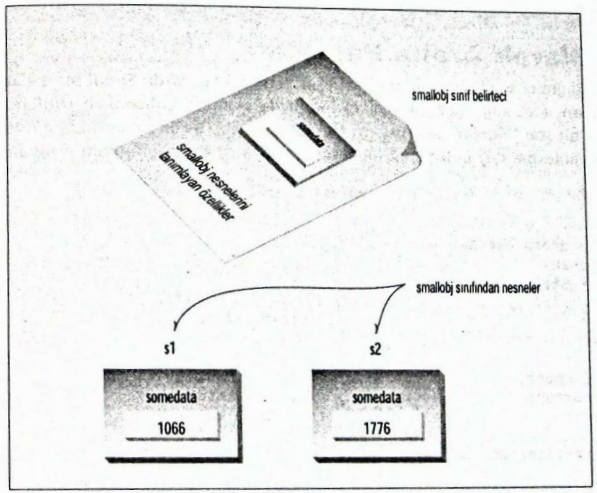
`setdata()`'yı ilk çağırdığımız noktada

```
s1.setdata(1066);
```

`s1` nesnesinin `setdata()` üye fonksiyonu çalışır. Bu fonksiyon, `s1` nesnesindeki `somedata` değişkeninin değerini 1066'ya eşitler. İkinci çağrıda ise

```
s2.setdata(1776);
```

`s2` nesnesinin içindeki `somedata` değişkenine 1776 değeri atanır. Böylece, Şekil 6.4'de gösterildiği gibi artık, `somedata` değişkenleri farklı değerler içeren iki nesnemiz vardır.



ŞEKİL 6.4: `smallobj` sınıfından iki nesnesi.

Benzer şekilde, `showdata()` fonksiyonuna yapılan aşağıdaki iki çağrı ise, iki nesnenin değerlerinin ekranda gösterilmesini sağlar:

```
s1.showdata();
s2.showdata();
```

Mesajlar

Bazı nesne yönelimli dillerde üye fonksiyonları çağırmaya *mesaj* denir. Bu nedenle

```
s1.showdata();
```

çağrısı `s1`'e, verisini göstermesini bildiren bir *mesaj gönderimi* olarak düşünülebilir. Burada kullanılan mesaj terimi C++'ın resmi bir tabiri değildir; sadece üye fonksiyonları işlerken kullanabileceğimiz yararlı bir kavramdır. Mesaj kavramı, nesnelerin, üye fonksiyonlar aracılığı ile haberleştiğimiz ayrı bütünler olduğunu vurgular. Bölüm 1'deki şirket benzetmesini hatırlarsak, bunu, satış birimindeki sekretere mesaj göndererek güneybatı dağıtım bölgesinde satılan ürünlerin listesini istemeye benzetebiliriz.

Fiziksel Nesneler Olarak C++ Nesneleri

Programlama yaparken pek çok durumda programların içindeki nesneler fiziksel nesneleri temsil eder: Hissedilebilen ya da görülebilen nesneleri program nesnesi olarak tanımlarız. Bu tür durumlar, program ile gerçek dünya arasındaki örtüşmenin canlı örneklerini sergiler. Şimdi böyle iki örneğe bakacağız: araba parçaları ve grafik daireler.

Nesneler Olarak Araba Parçaları

Son örnekte kullandığımız `smallobj` sınıfında tek bir veri parçası vardı. Şimdi biraz daha iddialı bir sınıfa bakalım. ("İddialı" derken siyaset bilimi derslerinde bahsedilen sınıflardan söz etmiyoruz.) Örneği en son "Yapılar" adlı Bölüm 4'te görülen **PARTS** adlı programda gördüğümüz araba parçaları envanterine dayalı bir sınıf oluşturacağız. **OBJPART** programının program listesi şöyledir:

```
// objpart.cpp
// nesne olarak araba parcasi
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class part                              //sinifi tanimla
   {
   private:
      int modelnumber;                  //model numarasi
      int partnumber;                   //parca numarasi
      float cost;                       //parcanin maliyeti
   public:
      void setpart(int mn, int pn, float c)       //veriyi belirle
         {
         modelnumber = mn;
         partnumber = pn;
         cost = c;
         }
      void showpart()                   //veriyi goster
```



```
         {
         cout << "Model " << modelnumber;
         cout << ", part " << partnumber;
         cout << ", costs $" << cost << endl;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   part part1;                                  //part sinifina ait
                                                //nesneyi tanimla
   part1.setpart(6244, 373, 217.55F);           //uye fonksiyonu cagir
   part1.showpart();                            //uye fonksiyonu cagir
   return 0;
   }
```

Bu programda `part` sınıfına yer verilmiştir. **SMALLOBJ**'deki gibi bir veri parçası yerine, bu sınıfta üç veri parçası vardır: `modelnumber`, `partnumber` ve `cost`. Tek üye fonksiyon olan `setpart()`, üç veri parçasının tümüne aynı anda veri sağlar. Diğer bir fonksiyon olan `showpart()` ise üç veri parçasının tümünde kayıtlı olan verileri gösterir.

Bu örnekte `part` cinsinden sadece bir adet nesne oluşturulur: `part1`. `setpart()` adlı üye fonksiyon bu parçanın verilerini 6244, 373, ve 217.55 değerlerine eşitler. Daha sonra, `showpart()` adlı üye fonksiyon da bu üç değeri gösterir. Programın çıktısı şöyledir:

```
Model 6244, part 373, costs $217.55
```

Bu örneğimiz, **SMALLOBJ**'den biraz daha gerçekçidir. Bir parça envanteri için program tasarlıyor olsaydınız muhtemelen `part` gibi bir sınıf oluşturmak isterdiniz. İşte bu araba parçaları örneği, gerçek dünyadaki fiziksel bir nesneyi simgeleyen bir C++ nesnesi örneğidir.

Nesne Olarak Daire

Sıradaki örneğimizde, bir daireyi simgelemek için kullanılan bir nesneyi inceleyeceğiz: Bu, bilgisayar ekranınızda görüntülenen türde bir dairedir. Bir görüntü, elinize alabileceğiniz bir yedek parça kadar somut bir nesne değildir. Yine de, programınız çalıştığında böyle bir daireyi kesinlikle ekranınızda görebilirsiniz.

Örneğimiz Bölüm 5'teki **CIRCSTRC** programının nesne yönelimli bir versiyonudur. (O programda olduğu gibi, projenize uygun Console Graphic Lite dosyalarını eklemeniz gerekir. Bu dosyalar, Giriş bölümünde anlatıldığı gibi, yayıncının Web sitesinden indirilebilir. Ek E, "Console Graphic Lite" bölümü bu dosyalarla ilgilidir. Ayrıca kullandığınız derleyici için de ilgili Ek bölümüne bakın.) Program değişik özelliklerde üç tane daire oluşturur ve bunları ekranda görüntüler. İşte **CIRCLES**'ın listesi:

```
// circles.cpp
// grafik nesneleri olarak daireler
#include "msoftcon.h "                  //grafik fonksiyonlari icin
////////////////////////////////////////////////////////////////
class circle                            //grafik daire
   {
   protected:
      int xCo, yCo;                     //merkezin koordinatlari
      int radius;
```

```
      color fillcolor;                    //renk
      fstyle fillstyle;                   //dolgu deseni
   public:                                //dairenin niteliklerini ayarla
      void set(int x, int y, int r, color fc, fstyle fs)
         {
         xCo = x;
         yCo = y;
         radius = r;
         fillcolor = fc;
         fillstyle = fs;
         }
      void draw()                         //daireyi ciz
         {
         set_color(fillcolor);            //rengi ayarla
         set_fill_style(fillstyle);       //dolguyu ayarla
         draw_circle(xCo, yCo, radius);   //ici dolu bir daire ciz
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   init_graphics();              //grafik sistemini ilk kullanima hazirla

   circle c1;                    //daireleri olustur
   circle c2;
   circle c3;
                                 //daire niteliklerini ayarla
   c1.set(15, 7, 5, cBLUE, X_FILL);
   c2.set(41, 12, 7, cRED, O_FILL);
   c3.set(65, 18, 4, cGREEN, MEDIUM_FILL);

   c1.draw();                    //daireleri ciz
   c2.draw();
   c3.draw();
   set_cursor_pos(1, 25);        //imleci sol alt koseye tasi
   return 0;
   }
```

Bu programın çıktısı, Bölüm 5'te Şekil 5.5'te gösterilen **CIRCSTRC** programının çıktısı ile aynıdır. Bu iki programı karşılaştırmayı enteresan bulabilirsiniz. **CIRCLES**'da her daire **CIRCSTRC**'dekinden farklı olarak, bir yapı değişkeni ve alakasız bir **circ_draw**() fonksiyonunun kombinasyonu yerine, bir C++ nesnesi olarak simgelenir. **CIRCLES**'da bir daire ile bağlantılı her şeyin – niteliklerin ve fonksiyonların – bir sınıf tanımı içinde nasıl bir araya getirildiğine dikkat edin.

CIRCLES'daki daire sınıfı **draw()** fonksiyonunun yanı sıra niteliklerini ayarlamak için bir de beş argümanlı **set()** fonksiyonunu gerektirir. Daha sonra bu fonksiyonu bırakıp, bunun yerine bir kurucu fonksiyon kullanmanın daha avantajlı olduğunu göreceğiz.

Veri Tipi Olarak C++ Nesneleri

C++ nesnelerinin simgeleyebildiği, işte başka tür bir bilgi daha: Kullanıcı tarafından tanımlanan bir veri tipinin değişkenleri. Bu örnekte, Bölüm 4'te bahsedilen İngiliz sistemine göre ölçülmüş bir uzaklığı simgeleyen nesneler kullanacağız. **ENGLOBJ**'nin listesi şöyledir:


```
// englobj.cpp
// Ingiliz olcumlerini kullanan nesneler
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance                                //Ingiliz Distance (uzaklik) sinifi
   {
   private:
      int feet;
      float inches;
   public:
      void setdist(int ft, float in)          //Distance'i argumanlara esitle
         { feet = ft; inches = in; }

      void getdist()                          //kullanicidan uzakligi al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }

      void showdist()                         //uzakligi ekranda goster
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist2;                     //iki uzaklik tanimla

   dist1.setdist(11, 6.25);                   //dist1'i ayarla
   dist2.getdist();                           //dist2'yi kullanicidan al
                                              //uzakliklari ekranda goster
   cout << "\ndist1 = "; dist1.showdist();
   cout << "\ndist2 = "; dist2.showdist();
   cout << endl;
   return 0;
   }
```

Bu programda, **Distance** sınıfı iki tane veri içerir: **feet** ve **inches**. Bu, Bölüm 4'teki örneklerde görülen **Distance** yapısına benzer. Fakat burada **Distance** sınıfının ayrıca üç tane de üye fonksiyonu vardır: **setdist()**, argümanları kullanarak **feet** ve **inches**'e değer atar; **getdist()**, kullanıcının klavyeden girdiği **feet** ve **inches** değerlerini alır ve **showdist()**, uzaklığı ayak ve inç biçiminde ekranda gösterir.

Dolayısıyla, **Distance** sınıfına ait bir nesnenin değeri iki şekilde ayarlanabilir. **main()** içinde **Distance** sınıfının iki nesnesi tanımlanır: **dist1** ve **dist2**. İlkine, **setdist()** üye fonksiyonu, 11 ve 6.25 argümanları ile birlikte kullanılarak, değer atanır. İkincinin değeri ise kullanıcı tarafından sağlanır. Programla etkileşim şu şekilde olabilir:

```
Enter feet: 10
Enter inches: 4.75
dist1 = 11'-6.25" <==== argümanlar üzerinden alınır
dist2 = 10'-4.75" <==== kullanıcı tarafından sağlanır
```

Kurucu Fonksiyonlar

ENGLOBJ örneği, bir nesne içindeki verilere değer atamak için üye fonksiyonlarının iki şekilde kullanılabileceğini gösterir. Ancak kimi zaman, bir nesne ilk kez oluşturulduğunda, bir üye fonksiyonu ayrıca çağırmaya gerek kalmadan, kendi kendisini ilk kullanıma hazırlayabilse (initialize) çok daha uygun olabilir. C++'ta otomatik ilk kullanıma hazırlama işlemi, *kurucu fonksiyon (constructor)* olarak adlandırılan özel bir üye fonksiyon kullanılarak gerçekleştirilir. Bir kurucu fonksiyon, ne zaman bir nesne oluşturulursa otomatik olarak çalıştırılan üye fonksiyondur.

Bir Sayaç Örneği

Örnek olsun diye, genel amaçlı bir programlama öğesi olarak kullanılabilecek bir nesneler sınıfı oluşturacağız. Bir *sayaç (counter)*, bir şeyleri sayan bir değişkendir. Dosya erişimlerini, kullanıcının Enter tuşuna kaç kez bastığını ya da bir bankaya giren müşterileri sayabilir. Böyle bir olayın her yinelenişinde sayaç bir artırılır (sayaca 1 eklenir). Ayrıca, o andaki toplamı bulmak için sayaca erişilebilir.

Bu sayacın programda önemli olduğunu ve birçok farklı fonksiyon tarafından erişilmesi gerektiğini varsayalım. C gibi prosedürel bir dilde bir sayaç muhtemelen bir global değişken olarak gerçeklenirdi. Ancak, Bölüm 1'de vurguladığımız gibi, global değişkenler programın tasarımını karmaşıklaştırır. Üstelik, global değişkenler kazara değişikliğe maruz kalabilir. **COUNTER** isimli bu örnek, sadece üye fonksiyonları tarafından değiştirilebilen bir sayaç değişkenine imkan verir.

```
// counter.cpp
// nesne bir sayac degiskenini simgeler
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;                  //sayacin gosterdigi sayi
   public:
      Counter() : count(0)                 //kurucu fonksiyon
         { /* bos govde */ }
      void inc_count()                     //count'u bir artir
         { count++; }
      int get_count()                      //count'u dondur
         { return count; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                         //tanimla ve ilk kullanima hazirla

   cout << "\nc1=" << c1.get_count();      //ekranda goster
   cout << "\nc2=" << c2.get_count();

   c1.inc_count();                         //c1'i bir artir
   c2.inc_count();                         //c2'yi bir artir
   c2.inc_count();                         //c2'yi bir artir
```



```
   cout << "\nc1=" << c1.get_count();      //tekrar ekranda goster
   cout << "\nc2=" << c2.get_count();
   cout << endl;
   return 0;
   }
```

Counter sınıfının veri olarak tek bir üyesi vardır: **unsigned int** tipinde **count** (sayacın gösterdiği sayı her zaman pozitif olduğu için). **Counter** sınıfının üç tane de üye fonksiyonu vardır: kurucu fonksiyon olan **Counter()** (buna birazdan göz atacağız), **count**'a bir ekleyen **inc_count()** fonksiyonu ve **count**'un şu anki değerini döndüren **get_count()** fonksiyonu.

Otomatik İlk Kullanıma Hazırlama

Counter tipinde bir nesne ilk kez oluşturulduğunda, bunun **count** değişkeninin başlangıçta 0 değerini almasını isteriz. Herşeyden önce, birçok sayaç 0'dan başlar. Bunu gerçekleştirmek için bir **set_count()** fonksiyonu tanımlar ve bunu 0 argüman değeri ile çağırabiliriz ya da **count**'a her zaman 0 değerini atayan bir **zero_count()** fonksiyonu tanımlayabiliriz. Ancak, bir **Counter** nesnesi oluşturduğumuz her zaman bu tür fonksiyonları çalıştırmak zorundayız.

```
Counter c1;        //bunu yaptigimiz her zaman
c1.zero_count();   //bunu da yapmak zorundayiz
```

Bu hataya açık bir yaklaşımdır, çünkü programcı nesneyi oluşturduktan sonra bunu ilk kullanıma hazırlamayı unutabilir. Her nesnenin oluşturulmasının ardından kendisini ilk kullanıma hazırlaması, özellikle belirli bir sınıfın çok sayıda nesneye sahip olduğu durumlarda çok daha güvenilir ve uygundur. **Counter** sınıfında, **Counter** kurucu fonksiyonu bunu gerçekleştirir. Ne zaman **Counter** tipinde yeni bir nesne oluşturulsa, bu fonksiyon otomatik olarak çağrılır. Dolayısıyla, **main()** içinde bulunan aşağıdaki ifade **Counter** tipinde iki nesne oluşturur:

```
Counter c1,c2;
```

Her biri oluşturulurken, kendi kurucu fonksiyonu, yani **Counter()** çalıştırılır. Bu fonksiyon **count** değişkenine 0 değerini atar. Yani, bu bir tek ifade, iki tane nesne oluşturmanın yanı sıra bu nesnelerin **count** değişkenlerine 0 değerini atma etkisine sahiptir.

Sınıf ile Aynı İsim

Kurucu fonksiyonların bazı ilginç özellikleri vardır. Öncelikle, kurucu fonksiyonların üyesi oldukları sınıfın ismi ile tamamen aynı isme sahip olmaları bir tesadüf değildir. Bu, derleyicinin bu fonksiyonları kurucu fonksiyon olarak anlamasını sağlayan yollardan biridir.

İkinci olarak, kurucu fonksiyonlar için dönüş tipi kullanılmaz. Neden? Kurucu fonksiyonlar sistem tarafından otomatik olarak çağrıldıkları için bunların bir değer döndürmeleri gereken bir program yoktur; bu nedenle, dönüş değerinin manası yoktur. Bu da, derleyicilerin bu fonksiyonların kurucu fonksiyon olduklarını anlamalarını sağlayan ikinci bir yoldur.

İlk Değerler Listesi

Bir kurucu fonksiyonun sürdürmesi gereken en genel görevlerinden biri, veri üyelerine ilk değerleri atamaktır. **Counter** sınıfında, kurucu fonksiyon **count** üyesine 0 değerini atamak zorundadır. Bu işlemin kurucu fonksiyonun gövdesinde yapılacağını düşünüyor olabilirsiniz. Şu şekilde:

```
count()
   {count=0;}
```

Ancak, bu (çalışıyor olmasına rağmen) tercih edilen bir yaklaşım değildir. Bir veri üyesine ilk değerini şöyle atamalısınız:

```
count():count(0)
   { }
```

İlk değer atama işlemi, üye fonksiyon deklaratörünün peşinden, fakat fonksiyon gövdesinden önce yer alır. Atanacak değerden önce iki nokta gelir. Değer, üye verinin peşinden parantez içinde yazılır. Eğer birden fazla üyeye değer atanacaksa, bu değerler virgül ile ayrılır. Elde edilen, bir *ilk değerler listesidir* (kimi zaman başka isimlerle de anılır, mesela *üyelerin ilk değerler listesi* gibi.).

```
someClass(): m1(7), m2(33), m2(4) <==== ilk degerler listesi
   { }
```

Kurucu fonksiyon gövdesi içindeki üyelere neden ilk değerleri vermiyoruz? Bunun sebepleri karmaşıktır; fakat bir gerçekle bağlantılıdır: İlk değerler listesi ile değer atanan üyelere daha kurucu fonksiyon çalışmaya bile başlamadan ilk değerleri verilir. Bu, bazı durumlarda önemlidir. Örneğin ilk değerleri listesi, **const** üye verilerine ve referanslarına ilk değer atamanın tek yoludur. Kurucu fonksiyon gövdesinde, sıradan fonksiyonlarda olduğu gibi, basit bir değer atmadan daha karmaşık faaliyetlerin gerçekleştirilmesi gerekir.

Counter Programının Çıktısı

Bu programın **main()** bölümü, **c1** ve **c2** adlı iki tane sayaç oluşturarak **Counter** sınıfını kullanır. **main()**, sayaçların kurucu fonksiyon tarafından ayarlanan ilk değerlerinin ekranda görüntülenmesini sağlar. Sonra **c1** ve **c2** iki kez artırılır; sayaçların tekrar kendi içeriğini görüntüleyebilmeleri sağlanır (bu bağlamda yanlış bir durum yok). İşte programın çıktısı:

```
c1=0
c2=0
c1=1
c2=2
```

Eğer bu, kurucu fonksiyonların ilan edildiği gibi çalıştıklarına dair yeterli bir kanıt değilse, kurucu fonksiyonu, çalıştığı zaman ekrana bir mesaj basacak şekilde yeniden yazabiliriz.

```
Counter() : count(0)
   { cout << "I'm the constructor\n"; }
```

Şimdi programın çıktısı şöyle görünür:

```
I'm the constructor
I'm the constructor
c1=0
c2=0
c1=1
c2=2
```


Gördüğünüz gibi, aşağıdaki ifade

```
Counter c1, c2;
```

main() içinde çalıştırıldığı zaman kurucu fonksiyon iki kez – bir kez **c1** için, bir kez de **c2** için – çalıştırılmış olur.

Kendi Verinizi Kendiniz Yapın

Kurucu fonksiyonlar, düşünecek olursanız oldukça şaşırtıcıdır. Dil derleyicileri (C veya VB, hatta C++ için) yazan her kimse, kullanıcı bir değişken tanımladığında, bir kurucu fonksiyonun eşdeğerini çalıştırmak zorundadır. Mesela, bir yerde bir **int** tanımladığınızda, bunun için bellek dört byte ayıracak bir kurucu fonksiyon vardır. Eğer kendi kurucu fonksiyonlarımızı yazabiliyorsak, bir derleyici yazarının yükünün bir kısmını alarak işe başlayabiliriz. Daha sonra inceleyeceğimiz gibi bu, kendi veri tiplerimizi oluşturma yolunda bir adımdır.

Bir Grafik Örneği

Daha önceki **CIRCLES** örneğini, **set()** fonksiyonu yerine bir kurucu fonksiyon kullanarak yeniden yazalım. Bu kurucu fonksiyon, **circles**'ın beş niteliğine ilk değer ataması yapmak için beş argüman ve ilk değerler listesinde beş madde içerecektir. İşte **CIRCTOR**'ın program listesi:

```
// circtor.cpp
// daireler ilk degerleri icin kurucu fonksiyonlar kullanir
#include "msoftcon.h"                    //grafik fonksiyonlari icin
////////////////////////////////////////////////////////////////
class circle                             //grafik daire
   {
   protected:
      int xCo, yCo;                      //merkezin koordinatlari
      int radius;
      color fillcolor;                   //renk
      fstyle fillstyle;                  //dolgu deseni
   public:
                                         //kurucu fonksiyon
   circle(int x, int y, int r, color fc, fstyle fs) :
      xCo(x), yCo(y), radius(r), fillcolor(fc), fillstyle(fs)
      { }

   void draw()                           //daireyi ciz
      {
      set_color(fillcolor);                  //rengi ayarla
      set_fill_style(fillstyle);             //dolguyu ayarla
      draw_circle(xCo, yCo, radius);         //ici dolu bir daire ciz
      }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   init_graphics();          //grafik sisteminin ilk kullanima hazirlanmasi
                             //daireleri olustur
   circle c1(15, 7, 5, cBLUE, X_FILL);
   circle c2(41, 12, 7, cRED, O_FILL);
   circle c3(65, 18, 4, cGREEN, MEDIUM_FILL);
```

```
    c1.draw();                  //daireleri ciz
    c2.draw();
    c3.draw();
    set_cursor_pos(1, 25);      //imleci sol alt koseye tasi
    return 0;
    }
```

Bu program `CIRCLES` ile aynıdır; sadece `set()` fonksiyonu bir kurucu fonksiyon ile değiştirilmiştir. Bunun `main()`'i ne kadar sadeleştirdiğine dikkat edin. Her nesne için, biri nesneyi oluşturmak, diğeri niteliklerini ayarlamak amacıyla ayrı iki ifade kullanmak yerine, artık tek bir ifade ile aynı anda hem nesne oluşturulur hem de nesnenin nitelikleri ayarlanır.

Yok Edici Fonksiyonlar

Bir nesne ilk kez oluşturulduğunda özel bir üye fonksiyonun – kurucu fonksiyonun – otomatik olarak çağrıldığını gördük. Bir nesne ortadan kaldırıldığında da başka bir fonksiyonun otomatik olarak çağrılacağını tahmin edebilirsiniz. Gerçekte de olan budur. Bu tür bir fonksiyona *yok edici fonksiyon* (*destructor*) denir. Bir yok edici fonksiyon, kurucu fonksiyon ile aynı isme sahiptir (bu, sınıf isminin aynısıdır), fakat bu isimden önce bir tilde (~) işareti kullanılır:

```
class Foo
   {
   private:
      int data;
   public:
      Foo() : data(0)   //kurucu fonksiyon (sinif ile ismi ayni)
         { }
      ~Foo()            //yok edici fonksiyon (ayni isim ama tilde ile)
         { }
};
```

Kurucu fonksiyonlar gibi yok edici fonksiyonların da dönüş değeri yoktur. Ayrıca argüman da almazlar (burada yapılan varsayım, bir nesneyi yok etmenin sadece bir tek yolunun var olduğudur.)

Yok edici fonksiyonlar en yaygın olarak, bir nesne için kurucu fonksiyonların ayırdığı bellek alanını iade etmek için kullanırlar. Bu etkinlikleri "İşaretçiler" adlı Bölüm 10'da inceleyeceğiz. O zamana kadar yok edici fonksiyonları pek fazla kullanmayacağız.

Fonksiyon Argümanları Olarak Nesneler

Bir sonraki örneğimiz `ENGLOBJ` programına bazı süslemeler ekler. Ayrıca sınıfların bazı yeni özelliklerini de gösterir. Bunlar, kurucu fonksiyonların aşırı yüklenmesi, üye fonksiyonları sınıf dışında tanımlamak ve – belki de en önemli olarak – fonksiyon argümanları olarak nesnelerin kullanılmasıdır. `ENGLCON`'un program listesi işte şöyledir:

```
// englcon.cpp
// kurucu fonksiyon ornegi, uye fonksiyon kullanarak nesneleri toplar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                          //Ingiliz uzunluk olcusu sinifi
   {
```



```
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0)          //kurucu fonksiyon (argumani yok)
         { }

      Distance(int ft, float in) : feet(ft), inches(in)   //kurucu fonksiyon (iki argumani var)
         { }

      void getdist()
         {                                       //uzunlugu kullanicidan al
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }

      void showdist()
         { cout << feet << "\'-" << inches << '\"'; }   //uzakligi ekranda goster

      void add_dist(Distance, Distance);     //deklarasyon
   };
//--------------------------------------------------------------
                                              // d2 ve d3 uzakliklarini topla
void Distance::add_dist(Distance d2, Distance d3)
   {
   inches = d2.inches + d3.inches;            //inches'i topla
   feet=0;                                    //(muhtemel elde icin)
   if(inches >= 12.0)                         //eger total 12.0'yi gecerse,
      {                                       //inches'i 12.0 azalt
      inches -= 12.0;                         //ve
      feet++;                                 //feet'i
      }                                       //1 artir
   feet += d2.feet + d3.feet;                 //feet'i topla
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist3;                     //iki uzaklik tanimla
   Distance dist2(11, 6.25);                  //dist2'yi tanimla ve ilk degerini ver

   dist1.getdist();                           //kullanicidan dist1'i al
   dist3.add_dist(dist1, dist2);              //dist3 = dist1 + dist2

                                              //butun uzakliklari ekranda goster
   cout << "\ndist1= "; dist1.showdist();
   cout << "\ndist2= "; dist2.showdist();
   cout << "\ndist3= "; dist3.showdist();
   cout << endl;
   return 0;
   }
```

Bu program bir ilk değeri olan `dist2` isimli bir uzaklık ile başlar. Program uzaklıkların toplamını bulmak için bu uzaklığa, değeri kullanıcı tarafından sağlanmış `dist1` uzaklığını ekler. Program daha sonra üç uzaklığın hepsini ekranda gösterir.

```
Enter feet: 17
Enter inches: 5.75
```

```
dist1 = 17' -5.75"
dist2 = 11' -6.25"
dist3 = 29' -0"
```

Şimdi, yeni özelliklerin bu programda nasıl uygulandığına bir bakalım.

Aşırı Yüklenen Kurucu Fonksiyonlar

Distance tipindeki değişkenlere ilk kez oluşturulduklarında bir değer verebilmek uygun olur. Yani, şu tür ifadeler kullanmak isteyebiliriz:

```
Distance width(5, 6.25);
```

Bu ifade, **width** isminde bir nesne tanımlar ve aynı zamanda **feet**'e 5, **inches**'e 6.25 değerlerini vererek bu nesneyi ilk kullanıma hazırlar.

Bunu gerçekleştirmek için şöyle bir kurucu fonksiyon yazılır:

```
Distance(int ft, float in) : feet(ft), inches(in)
   { }
```

Bu ifade, kurucu fonksiyona argüman olarak hangi değerler geçiriliyorsa, bu değerleri **feet** ve **inches** üye verilerine atar. Buraya kadar her şey yolunda.

Ancak, biz ayrıca, aynı **ENGLOBJ** programında yaptığımız gibi **Distance** tipindeki değişkenleri başlangıç değeri atamadan da tanımlamak istiyoruz.

```
Distance dist1, dist2;
```

ENGLOBJ'de kurucu fonksiyon yoktu ama, tanımlarımız yeterli olmuştu. Kurucu fonksiyon olmadan bunlar nasıl işe yaramıştı? Çünkü, derleyici tarafından otomatik olarak gizli ve argümansız bir kurucu fonksiyon program içine yerleştirilir ve biz her ne kadar sınıf içinde bir kurucu fonksiyon tanımlamış olmasak da, bu kurucu fonksiyon nesneleri oluşturur. Bu argümansız kurucu fonksiyon, *varsayılan* (*default*) *kurucu fonksiyon* olarak adlandırılır. Eğer nesneler, kurucu fonksiyon tarafından otomatik olarak oluşturulmuş olmasalardı, tanımlanmamış bir kurucu fonksiyon içeren bir sınıfın nesnelerini tanımlayacaktınız.

Genellikle varsayılan (argümansız) kurucu fonksiyonun veri üyelerini de ilk kullanıma hazırlamak isteyebiliriz. Eğer kurucu fonksiyonun bunu yapmasına izin verirsek, veri üyelerine hangi değerlerin verildiğinden haberimiz olmaz. Hangi değerlerin verildiğini önemsiyorsak kurucu fonksiyonu açıkça tanımlamamız gerekir. Bu, **ENGLECON** içinde gösterilmiştir:

```
Distance() : feet(0), inches(0.0)      //varsayilan kurucu fonksiyon
   { }           //fonksiyon govdesi yok; fonksiyon bir is yapmiyor
```

Veri üyelerine başlangıçta sabit değerler atanır. Bu kez, **feet** için bir 0 tamsayı değeri ve **inches** için de bir 0.0 float değeri atanır. Artık argümansız kurucu fonksiyon ile başlangıç değerleri atanan nesneler kullanılabilir. Üstelik, başlangıçta bu nesnelerin bazı rasgele değerler yerine 0 uzaklığı (0 feet artı 0.0 inç) simgeledikleri sağlanmış olur.

Şimdi ortada aynı isme sahip, **Distance()** isminde, iki tane fonksiyon olduğundan ötürü kurucu fonksiyon için *aşırı yüklenmiş* (*overloaded*) diyebiliriz. Bir nesne oluşturulduğunda bu iki kurucu fonksiyondan hangisinin çalıştırılacağı, tanımda kullanılan argüman sayısına bağlıdır:


```
Distance length;              //birinci kurucu fonksiyonu cagirir
Distance width(11, 6.0);      //ikinci kurucu fonksiyonu cagirir
```

Sınıfın Dışında Tanımlanan Üye Fonksiyonlar

Şimdiye kadar bir sınıf tanımı içinde yer alan üye fonksiyonları gördük. Her zaman böyle olması gerekmez. **ENGLCON**, **Distance** sınıf tanımında yer almayan, **add_dist()** adında bir üye fonksiyon içerir. Sınıf içinde bu fonksiyonun sadece *deklarasyonu* yer alır. Fonksiyon deklarasyonu şu ifade ile verilir:

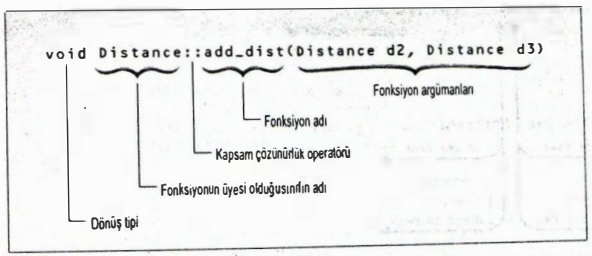
```
void add_dist(Distance, Distance);
```

Bu ifade derleyiciye, bu fonksiyonun sınıfın bir üyesi olduğunu, fakat sınıf deklarasyonunun dışında, program listesinin başka bir yerinde tanımlanacağını söyler.

ENGLCON'da **add_dist()** fonksiyonu sınıf tanımın peşinden tanımlanır. **ENGLCON**, Bölüm 4'teki **ENGLSTRC** fonksiyonundan uyarlanmıştır:

```
                                          //d2 ve d3 uzakliklarini topla
void Distance::add_dist(Distance d2, Distance d3)
   {
   inches = d2.inches + d3.inches;        //inches'i topla
   feet=0;                                //(muhtemel bir elde icin)
   if(inches >= 12.0)                     //eger toplam 12.0'yi.gecerse,
      {                                   //inches'i
      inches -= 12.0;                     //12.0 azalt ve
      feet++;                             //feet'i
      }                                   //1 artir
   feet += d2.feet + d3.feet;             //feet'leri topla
   }
```

Bu tanımdaki deklaratör tanıdık olmayan bazı söz dizimleri içerir. Fonksiyon isminden, yani **add_dist()**'ten önce, sınıfın ismi olan **Distance** ve yeni bir simge, iki tane iki nokta (::) işareti kullanılıyor. Bu simge, *kapsam çözünürlük operatörü* (scope resolution operator) olarak adlandırılır. Bu, herhangi bir sınıfın herhangi bir öğesini açıkça belirtme yollarından biridir. Bu durumda, **Distance::add_dist()** şu anlama gelir: "**Distance** sınıfının **add_dist()** üye fonksiyonu". Şekil 6.5 bu kullanımı gösteriyor.



ŞEKİL 6.5: Kapsam çözünürlük operatörü.

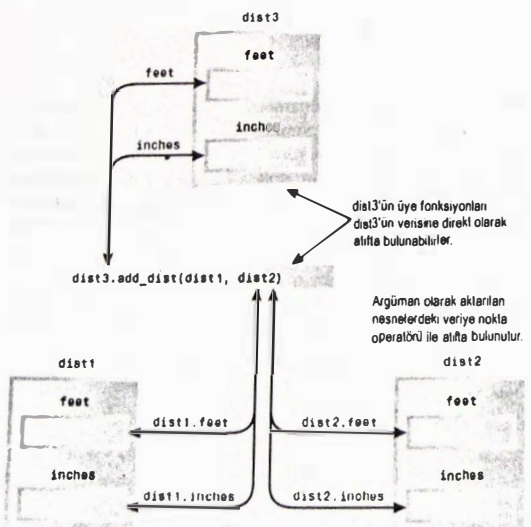
Argüman Olarak Nesneler

Artık **ENGLCON**'un nasıl çalıştığına bakabiliriz. **dist1** ve **dist3** uzaklıkları varsayılan kurucu fonksiyon (argümansız olan fonksiyon) kullanılarak oluşturulur. **dist2** uzaklığı iki argümanlı kurucu fonksiyon ile oluşturulur ve bu argümanlar üzerinden aktarılan değerlere eşitlenir. Değerleri kullanıcıdan alan **getdist()** üye fonksiyonu çağrılarak **dist1** için bir değer elde edilir. Şimdi **dist1** ve **dist2**'yi toplayarak **dist3**'ü elde edebiliriz. **main()**'deki fonksiyon çağrısı bunu gerçekleştirir:

```
dist3.add_dist(dist1, dist2);
```

Toplanacak iki uzaklık olan **dist1** ve **dist2**, **add_dist()** fonksiyonuna argüman olarak gönderilir. Nesneleri içeren argümanların söz dizimi, **int** gibi basit veri tiplerini içeren argümanların söz dizimi ile aynıdır. Yani, nesnenin ismi argüman olarak aktarılır. **add_dist()** fonksiyonu **Distance** sınıfının bir üye fonksiyonu olduğu için, kendisine **Distance** sınıfı argüman olarak gönderildiği takdirde **add_dist()**, bu sınıfın herhangi bir nesnesindeki korumalı veriye erişebilir. Bu erişimi şu tür isimler kullanarak yapabilir: **dist1.inches** ve **dist2.feet**.

add_dist()'i yakından incelediğimizde **add_dist()**'in üye fonksiyonlarla ilgili bazı önemli gerçeklerin altını çizdiğini görürüz. Bir üye fonksiyonun, kendisini çağıran nesneye erişmesine her zaman izin verilir: Bu nesne, üye fonksiyona nokta operatörü ile bağlanmış olan nesnedir. Fakat bu fonksiyon, diğer nesnelere de erişebilir. Aşağıdaki **ENGLCON** içinde yer alan ifadede **add_dist()** hangi nesnelere erişebilir?



ŞEKİL 6.6: Sonucun nesnelerle gösterilmesi.


```
dist3.add_dist(dist1, dist2);
```

add_dist(), kendisini çağıran nesne olan **dist3**'ün yanı sıra, **dist1** ve **dist2**'ye de ayrıca erişebilir, çünkü **dist1** ve **dist2** argüman olarak aktarılır. **dist3**'ü bir tür hayalet argüman olarak düşünebilirsiniz; **dist3** argüman olarak aktarılmasa bile üye fonksiyonun bu nesneye her zaman erişim hakkı vardır. Şu ifadenin belirtmek istediği de budur: "**dist3**'ün **add_dist()** üye fonksiyonunu çalıştır." **feet** ve **inches** değişkenlerine bu fonksiyon içinden erişildiğinde aslında **dist3.feet** ve **dist3.inches**'e erişilmiş olur.

Dikkat ederseniz, sonuç fonksiyondan döndürülmez. **add_dist()** fonksiyonunun dönüş tipi **void**. Sonuç otomatik olarak **dist3** nesnesinde saklanır. Şekil 6.6, **dist1** ve **dist2** uzaklıklarının toplanıp, sonucun **dist3**'te saklanmasını gösteriyor.

Özetle, bir üye fonksiyona yapılan her çağrı (eğer söz konusu fonksiyon statik fonksiyon değilse; buna daha sonra değineceğiz) belirli bir nesne ile bağlantılıdır. Fonksiyon, üye isimlerini tek başına kullanarak (**feet** ve **inches**) nesnenin – private veya public – tüm üyelerine doğrudan erişim hakkına sahip olur. Fonksiyon ayrıca nesnenin ve üyenin ismini, nokta operatörü ile birleştirerek (**dist1.inches** veya **dist2.feet**), aynı sınıfın argüman olarak aktarılmış diğer nesnelerine de dolaylı yoldan erişim sağlayabilir.

Varsayılan Kopyalama Kurucu Fonksiyonu

Nesnelere iki yoldan başlangıç değeri atanabileceğini öğrendik. Argümansız bir kurucu fonksiyon üye verilere sabit değerler atayabilir; birden fazla argüman içeren bir kurucu fonksiyon, üye verilerine argüman olarak aktarılan değerleri atayabilir. Şimdi bir nesneye değer atamanın bir başka yönteminden bahsedeceğiz: Bir nesneye *aynı tipte başka bir nesne* yardımıyla değer atayabilirsiniz. Şaşırtıcı gelebilir ama, bunun için özel bir kurucu fonksiyon oluşturmanıza gerek yoktur; böyle bir fonksiyon tüm sınıflar içinde zaten standart olarak vardır. Bu fonksiyona, *varsayılan kopyalama kurucu fonksiyonu* (*default copy constructor*) denir. Bu, argümanı kurucu fonksiyon ile aynı sınıfın bir nesnesi olan tek argümanlı bir kurucu fonksiyondur. **ECOPYCON** programı bu kurucu fonksiyonun nasıl kullanıldığını gösteriyor.

```
// ecopycon.cpp
// varsayilan kopyalama kurucu fonksiyonuyla nesneleri ilk kullanima hazirlama
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                  //Ingiliz sistemine gore Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
                                //kurucu fonksiyon (argumansiz)
      Distance() : feet(0), inches(0.0)
         { }
      //Dikkat: tek argumanli kurucu fonksiyon yok
                                //kurucu fonksiyon (iki argumanli)
      Distance(int ft,float in):feet(ft),inches(in)
         { }
      void getdist()            //kullanicidan uzakligi al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
```

```
        }
    void showdist()             //uzakligi ekranda goster
        { cout << feet << "\'-" << inches << '\"'; }
    };
//////////////////////////////////////////////////////////////
int main()
    {
    Distance dist1(11, 6.25);   //iki argumanli kurucu fonksiyon
    Distance dist2(dist1);      //tek argumanli kurucu fonksiyon
    Distance dist3 = dist1;     //yine tek argumanli kurucu fonksiyon

                                //tum uzakliklari ekranda goster
    cout << "\ndist1 = "; dist1.showdist();
    cout << "\ndist2 = "; dist2.showdist();
    cout << "\ndist3 = "; dist3.showdist();
    cout << endl;
    return 0;
    }
```

İki argümanlı kurucu fonksiyonu kullanarak **dist1**'e 11'-6.25" değeri atanır. Sonra **Distance** tipinde iki nesne daha tanımlanır: **dist2** ve **dist3**. Bunların her ikisine de **dist1** değeri atanır. Bu işlemin tek argümanlı bir kurucu fonksiyon tanımlamayı gerektirdiğini düşünebilirsiniz, fakat bir nesneye aynı tipte başka bir nesne yoluyla değer atamak özel bir olaydır. Bu tanımların her ikisi de varsayılan kopyalama kurucu fonksiyonu kullanır. **dist2** nesnesine aşağıdaki ifade ile değer atanır:

```
Distance dist2(dist1);
```

Bu ifade, **Distance** sınıfı için tanımlanan varsayılan kopyalama kurucu fonksiyonunun, **dist1**'in üyelerini **dist2**'nin üzerine birer birer kopyalamasını sağlıyor. Şaşırtıcı gelebilir, fakat tamamen aynı etkiye sahip farklı bir format daha vardır: Bu format da **dist1**'in üyelerini birer birer **dist3**'e kopyalamayı sağlar.

```
Distance dist3 = dist1;
```

Bu bir değer atama ifadesi gibi görünüyorsa da, aslında değildir. Her iki format da varsayılan kopyalama kurucu fonksiyonunu çağırır; bu nedenle, her ikisi de birbirinin yerine değiştirilerek kullanılabilir. İşte programın çıktısı:

```
dist1 = 11'-6.25"
dist2 = 11'-6.25"
dist3 = 11'-6.25"
```

Bu çıktıdan görülüyor ki, **dist2** ve **dist3**'e, **dist1** ile aynı değerler atanmıştır. "Sanal Fonksiyonlar" adlı Bölüm 11'de, varsayılan kopyalama kurucu fonksiyonunu aşırı yükleyerek kendi kopyalama fonksiyonumuzu nasıl oluşturacağımızı ele alacağız.

Fonksiyonlardan Nesneler Döndürmek

ENGLCON örneğinde nesnelerin fonksiyonlara argüman olarak aktarıldığını gördük. Şimdi bir nesne döndüren bir fonksiyon örneğini inceleyeceğiz. **ENGLRET**'i üretmek için **ENGLCON** programı üzerinde değişiklikler yapacağız:


```
// englret.cpp
// fonksiyon Distance tipinde bir deger dondurur
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////////
class Distance                          //Ingiliz sistemine gore Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0) //kurucu fonksiyon (argumansiz)
         { }
      Distance(int ft, float in) : feet(ft), inches(in)
                                        //kurucu fonksiyon (iki argumanli)
         { }

      void getdist()
         {                              //kullanicidan uzakligi al
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }
      void showdist()                   //uzakligi ekranda goster
         { cout << feet << "\'-" << inches << '\"'; }
      Distance add_dist(Distance);      //topla
   };
//-------------------------------------------------------------
//bu uzakligi d2'ye ekle, toplami dondur
Distance Distance::add_dist(Distance d2)
   {
   Distance temp;                       //gecici degisken
   temp.inches = inches + d2.inches;    //inches'leri topla
   if(temp.inches >= 12.0)              //eger toplam 12.0'yi gecerse,
      {                                 //inches'i
      temp.inches -= 12.0;              //12.0 azalt ve
      temp.feet=1;                      //feet'i
      }                                 //1 artir
   temp.feet += feet + d2.feet;         //feet'leri topla
   return temp;
   }
//////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist3;               //iki uzaklik tanimla
   Distance dist2(11, 6.25);            //dist2'yi tanimla, deger ata

   dist1.getdist();                     //kullanicidan dist1'i al
   dist3 = dist1.add_dist(dist2);       //dist3 = dist1 + dist2

                                        //tum uzakliklari ekranda goster
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
return 0;
}
```

ENGLRET programı **ENGLCON**'a çok benzer; fakat, her ikisi arasındaki farklar, fonksiyonların nesnelerle nasıl çalıştığı ile ilgili önemli özellikleri açığa çıkarır.

Argümanlar ve Nesneler

ENGLCON'da iki uzaklık `add_dist()` fonksiyonuna argüman olarak aktarılmıştı, sonuç ise `add_dist()`'in üyesi olduğu bir nesnenin içinde, yani `dist3`'te saklanmıştı. ENGLRET'te tek uzaklık olan `dist2`, argüman olarak `add_dist()`'e aktarılır. Bu, `add_dist()`'in bir üyesi olduğu `dist1` nesnesi ile toplanır. Sonuç, `main()`'de `dist3`'e aşağıdaki ifade ile atanır.

```
dist3 = dist1.add_dist(dist2);
```

Bunun etkisi ENGLCON'da karşılık gelen ifadeninki ile aynıdır; fakat bu, biraz daha doğal görünür, çünkü değer atama operatörü (=), daha doğal bir şekilde kullanılır. Bölüm 8'de "Operatörlerin Aşırı Yüklenmesi" bahsinde, çok daha doğal bir deyim elde etmek için aritmetik + operatörünü nasıl kullanacağımızı öğreneceğiz.

```
dist3 = dist1 + dist2;
```

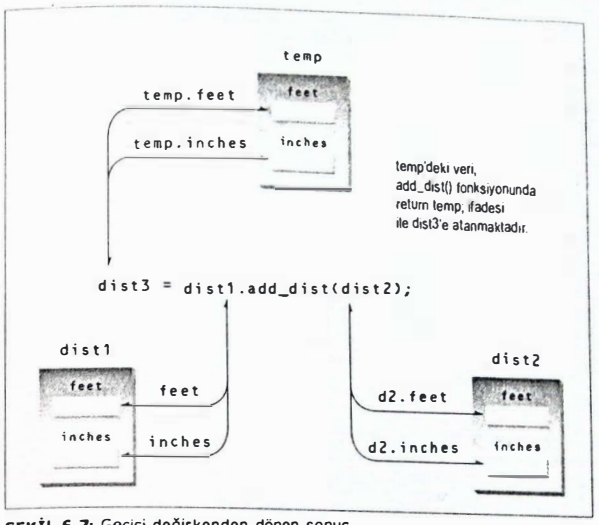
İşte ENGLRET'teki `add_dist()` fonksiyonu:

```
//bu uzakligi d2 uzakligina ekle, toplami dondur
Distance Distance::add_dist(Distance d2)
   {
   Distance temp;                      //gecici degisken
   temp.inches = inches + d2.inches;   //inches'leri topla
   if(temp.inches >= 12.0)             //eger toplam 12.0'yi gecerse,
      {                                //inches'leri
      temp.inches -= 12.0;             //12.0 azalt ve
      temp.feet = 1;                   //feet'i
      }                                //1 artir
   temp.feet += feet + d2.feet;        //feet'leri topla
   return temp;
   }
```

Bunu ENGLCON'daki aynı fonksiyon ile karşılaştırın. Gördüğünüz gibi, ikisi arasında güçlükle fark edilen farklılıklar vardır. ENGLRET versiyonunda `Distance` sınıfının geçici bir nesnesi oluşturulur. Bu nesne, kendisini çağıran programa döndürülünceye kadar toplamı içinde saklar. Toplam iki uzaklığı ekleyerek elde edilir. İlki, `add_dist()`'in bir üyesi olduğu bir nesne, yani `dist1`'dir. Bu nesnenin üye verileri fonksiyon içinden `feet` ve `inches` ile erişilir. İkinci uzaklık ise, argüman olarak aktarılan bir nesne, yani `dist2`'dir. Bunun üye verilerine ise `d2.feet` ve `d2.inches` ile erişilir. Sonuç `temp`'te saklanır; sonuca `temp.feet` ve `temp.inches` ile erişilir. Bu `temp` nesnesi daha sonra fonksiyon tarafından aşağıdaki ifade kullanılarak döndürülür:

```
return temp;
```

Dönen bu değer, `main()`'de `dist3`'e atanır. Dikkat ederseniz, `dist1` değiştirilmez, sadece `add_dist()` fonksiyonuna değer temin eder. Şekil 6.7 bunun nasıl göründüğünü ortaya koyuyor.



ŞEKİL 6.7: Geçici değişkenden dönen sonuç.

İskambil Kağıdı Oyununa Bir Örnek

Gerçek dünyayı modelleyen daha büyük nesnelere örnek olarak şimdi Bölüm 4'teki iskambil kağıdı oyununun bir varyasyonuna göz atalım. `CARDOBJ` adındaki bu program nesneleri kullanmak amacıyla yazıldı. Program, hiçbir yeni kavram sunmaz, fakat şu ana değin ele aldığımız programlama fikirlerinin neredeyse tamamını kullanır.

`CARDS` örneğinde olduğu gibi, `CARDOBJ` belirli değerleri olan üç tane kart seçer ve kullanıcıyı şaşırtmak amacıyla kartların yerlerini değiştirir. Ancak, `CARDOBJ`'de kartların her biri `card` sınıfının bir nesnesidir. İşte program listesi:

```
// cardobj.cpp
// nesne olarak iskambil kagidi
#include <iostream>
using namespace std;
enum Suit {clubs, diamonds, hearts, spades };
const int jack = 11;                    //2 den 10 a kadar
const int queen = 12;                   //isimsiz tamsayilar
const int king = 13;
const int ace = 14;
////////////////////////////////////////////////////////////////
class card
   {
   private:
      int number;                       //2'den 10'a kadar, vale, kiz, papaz, as
      Suit suit;                        //maca, karo, kupa, sinek
   public:
      card ()                           //kurucu fonksiyon (argumansiz)
```

```
        { }                             //kurucu fonksiyon (iki argumanli)
      card (int n, Suit s) : number(n), suit(s)
        { }
      void display();                   //karti ekranda goster
      bool isEqual(card);               //baska bir karta benziyor mu?
   };
//-------------------------------------------------------------
void card::display()                    //karti ekranda goster
   {
   if(number >= 2 && number <= 10)
      cout << number << " of ";
   else
      switch(number)
         {
         case jack: cout << "jack of "; break;
         case queen: cout << "queen of "; break;
         case king: cout << "king of "; break;
         case ace: cout << "ace of "; break;
         }
   switch(suit)
      {
      case clubs: cout << "clubs "; break;
      case diamonds: cout << "diamonds "; break;
      case hearts: cout << "hearts "; break;
      case spades: cout << "spades "; break;
      }
   }
//-------------------------------------------------------------
bool card::isEqual(card c2)             //kartlar esitse true dondur
   {
   return (number==c2.number && suit==c2.suit) ? true : false;
   }
//////////////////////////////////////////////////////////////////
int main()
   {
   card temp, chosen, prize;        //cesitli kartlari tanimla
   int position;

   card card1( 7, clubs );          //card1'i tanimla ve ilk kullanima hazirla
   cout << "\nCard 1 is the ";
   card1.display();                 //card1'i ekranda goster

   card card2( jack, hearts );      //card2'yi tanimla ve ilk kullanima hazirla
   cout << "\nCard 2 is the ";
   card2.display();                 //card2'yi ekranda goster

   card card3( ace, spades );       //card3'u tanimla ve ilk kullanima hazirla
   cout << "\nCard 3 is the ";
   card3.display();                 //card3'u ekranda goster

   prize = card3;                   //tahmin edilecek kart prize'da tutulur

   cout << "\nI'm swapping card 1 and card 3";
   temp = card3; card3 = card1; card1 = temp;

   cout << "\nI'm swapping card 2 and card 3";
   temp = card3; card3 = card2; card2 = temp;
```



```
   cout << "\nI'm swapping card 1 and card 2";
   temp = card2; card2 = card1; card1 = temp;

   cout << "\nNow,where (1, 2, or 3) is the";
   prize.display();                 //prize kartini ekranda goster
   cout << "?";
   cin >> position;                 //kullanicinin konum tahminini al

   switch (position)
      {                             //chosen'a kullanicinin tahmini koy
      case 1: chosen = card1; break;
      case 2: chosen = card2; break;
      case 3: chosen = card3; break;
      }
   if(chosen.isEqual(prize))        //secilen kart prize mi?
      cout << "That's right! You win!";
   else
      cout << "Sorry. You lose.";
   cout << " You chose the ";
   chosen.display();                //secilen karti ekranda goster
   cout << endl;
   return 0;
   }
```

card sınıfı içinde iki kurucu fonksiyon vardır. Birincisi, argüman almaz ve **main()**'de **temp**, **chosen** ve **prize** kartlarını oluşturmak amacıyla kullanılır. Bu kartlara ilk değerleri atanmaz. İkinci kurucu fonksiyon, iki argüman alır; **card1**, **card2** ve **card3**'ü oluşturmak ve bu kartlara belirli ilk değerleri atamak amacıyla kullanılır. Kurucu fonksiyonların yanı sıra **card**'ın her ikisi de sınıfın dışında tanımlanan diğer iki üye fonksiyonu daha vardır.

display() fonksiyonu hiç argüman almaz, sadece üyesi bulunduğu kart nesnesini, kartın numara ve takım verilerini kullanarak ekranda görüntüler. **main()**'deki

```
chosen.display();
```

ifadesi kullanıcı tarafından seçilen kartı ekranda gösterir.

isEqual() fonksiyonu, kartın argüman olarak aktarılan kartla aynı olup olmadığını kontrol eder. Fonksiyon, üyesi olduğu kart ile argüman olarak aktarılan kartı karşılaştırmak için koşul operatörü kullanır. Bu fonksiyon **if...else** ifadesi kullanılarak da yazılabilirdi:

```
if(number==c2.number && suit==c2.suit)
   return true;
else
   return false;
```

Fakat koşul operatörü daha derli toplu görünür.

isEqual() içinde argümana **c2** denilmesinin sebebi, karşılaştırılacak iki kart olduğunu hatırlatması içindir: Kartlardan ilki, **isEqual()**'ın üyesi olduğu nesnedir. **main()**'deki şu deyim

```
if(chosen.isEqual(prize))
```

seçilen (**chosen**) kart ile **prize** nesnesinde saklanan kartı karşılaştırır.

Kullanıcı yanlış tahminde bulunduğunda programın çıktısı işte şöyledir:

```
Card 1 is the 7 of clubs
Card 2 is the jack of hearts
Card 3 is the ace of spades
I'm swapping card 1 and card 3
I'm swapping card 2 and card 3
I'm swapping card 1 and card 2
Now, where (1, 2, or 3) is the ace of spades? 1
Sorry, you lose. You chose the 7 of clubs
```

Yapılar ve Sınıflar

Bu kitapta şu ana kadar yer alan örnekler, yapıları, verileri gruplama yollarından biri olarak; sınıfları da hem veri hem de fonksiyonları gruplama yollarından biri olarak ortaya koydu. Aslında, yapıları sınıfları kullandığınız biçimle neredeyse tamamen aynı biçimde kullanabilirsiniz. Sınıf ve yapı arasındaki biçimsel tek fark, bir sınıfın içindeki üyelerin varsayılan durumda private olmalarıdır. Bir yapının içindekiler ise varsayılan durumda public olurlar. Sınıflar için kullana geldiğimiz format şöyle:

```
class foo
   {
   private:
      int data1;
   public:
      void func();
   };
```

Sınıflarda private varsayılan durum olduğu için, bu anahtar kelimeyi kullanmak gereksizdir. Rahatlıkla şu şekilde de yazabilirsiniz:

```
class foo
   {
      int data1;
   public:
      void func();
   };
```

data1 yine private olacaktır. Bir çok programcı bu stili tercih eder. Biz **private** anahtar kelimesini dahil etmek istiyoruz, çünkü bu, anlaşılırlığı artırmayı sağlar.

Bu sınıfın yaptığı işin aynısını gerçekleştiren bir yapı kullanmak istiyorsanız, public üyeleri private üyelerden önce yerleştirmek şartıyla **public** anahtar kelimesinden kurtulabilirsiniz:

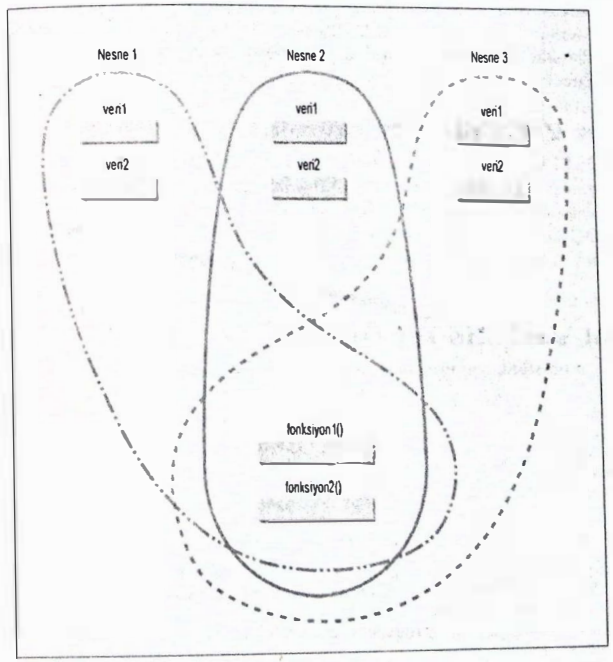
```
struct foo
   {
      void func();
   private:
      int data1;
   };
```

Çünkü **public** zaten varsayılan durumdur. Ancak, programcılar pek çok durumda bir **struct** iladesini bu şekilde kullanmazlar. Yapıları sadece verileri gruplamak için, sınıfları ise hem veri hem de fonksiyonları gruplamak için kullanırlar.

Sınıflar, Nesneler ve Bellek

Size şöyle bir izlenim vermiş olabiliriz: Bir sınıftan oluşturulan her nesne, bu sınıfın verilerinin ve üye fonksiyonlarının ayrı ayrı kopyalarını içerir. Bu ilk varsayım olarak iyi bir varsayımdır, çünkü bu varsayım, nesnelerin komple, kendi kendini içeren bir bütün olduklarını ve sınıf tanımını kullanmak için tasarlandıklarını vurgular. Bunu zihnimizde canlandırmak için bir üretim bandında kayan ve her biri bir plana göre (sınıfı tanımı) hazırlanmış arabalar (nesneler) olarak düşünebiliriz.

Aslında işler bu kadar da basit değildir. Her nesnenin kendisine ait ayrı veri parçaları içerdiği doğrudur. Fakat diğer yandan, şu ana kadar inanmanız istenilenlerin aksine, belirli bir sınıfın nesnelerinin tümü aynı üye fonksiyonları kullanılırlar. Üye fonksiyonlar sadece bir kez oluşturulur ve belleğe yerleştirilirler. Bu, üye fonksiyonlar sınıf tanımı içinde tanımlandıkları zaman olur. Bu makul bir yaklaşımdır; gerçekten de bir sınıfın üye fonksiyonlarının kopyasını, bu sınıfın bir nesnesinin oluşturulduğu her seferde oluşturmanızın bir anlamı yoktur; çünkü nesnelerin her birinin fonksiyonları zaten birbirinin aynıdır. Ancak, veriler farklı değerler saklayacaktır; bu nedenle, her nesne için her veri parçasının ayrı bir örneği olmalıdır. Nesneler tanımlandığında veriler de belleğe yerleştirilir; yani, her nesne için ayrı bir veri kümesi mevcuttur. Şekil 6.8 bunun nasıl göründüğünü izah ediyor.



ŞEKİL 6.8: Nesneler, veri, fonksiyonlar ve bellek.

Bu bölümün başındaki SMALLOBJ örneğinde smallobj tipinde iki nesne yer alır; bu nedenle somedata'nın bellekte iki örneği mevcuttur. Bununla birlikte, setdata() ve showdata() fonksiyonlarının bellekte yalnızca bir örneği vardır. Bu fonksiyonlar sınıfın tüm fonksiyonları tarafından paylaşılır. Bu bir karmaşa doğurmaz (en azından tek kanallı (single-threaded) sistemlerde), çünkü her seferinde sadece bir fonksiyon çalıştırılır.

Pek çok kez, bütün bir sınıf için sadece tek bir üye fonksiyon olduğunu bilmenize gerek yoktur. Her nesneyi, hem kendi verilerini hem de kendi üye fonksiyonlarını içeriyor şeklinde göz önünde canlandırmanız çok daha basittir. Fakat bazı durumlarda, mesela çalışan bir programın boyutunu tahmin ederken, perdenin arkasında neler olduğunu bilmek yararlı olabilir.

Statik Sınıf Verisi

Her nesnenin kendisine ait ayrı veri içerdiğini söyledikten sonra, şimdi bunu biraz düzeltmemiz gerekiyor. Eğer bir veri parçası **static** olarak bildirilmişse, bu sınıfa ait kaç tane nesne olursa olsun, sınıfın tümü için sadece bir tane böyle bir veri parçası oluşturulur. Bir statik veri parçası, aynı sınıfın tüm nesnelerinin ortak bir bilgiyi paylaşmaları gerektiğinde kullanışlıdır. **static** olarak tanımlanmış bir üye değişkeni, normal bir statik değişkene benzer özelliklere sahiptir: Sadece sınıf içinde erişilebilirdir; fakat, kapsamı programın tamamıdır. Bu sınıfa ait hiç nesne olmasa da, statik veri parçası varlığını sürdürür (Statik değişkenlerle ilgili bilgi için Bölüm 5'e bakınız.) Bununla birlikte, normal bir statik değişken, fonksiyon çağrıları arasında bilgiyi yitirmemek için kullanılırken, bir sınıfın statik üye verileri bu sınıfın nesneleri arasında bilgiyi paylaşmak için kullanılır.

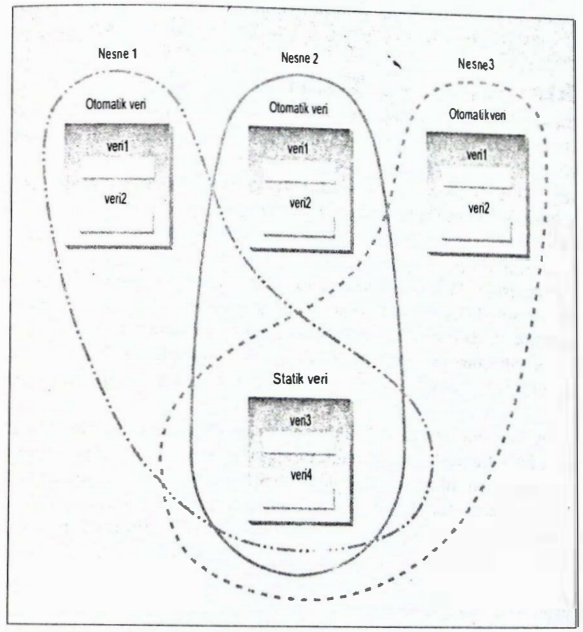
Statik Sınıf Verisinin Kullanımları

Neden statik üye verisi kullanmak isteyebilirsiniz? Örnek olarak bir nesne, programda kendi sınıfına ait kaç tane nesne olduğunu bilmek istiyor diyelim. Bir araba yarışında, örneğin, bir yarış arabası diğer arabalardan kaç tanesinin hâlâ yarışta olduğunu bilmek isteyebilir. Bu durumda, bir statik değişken olan **count**, sınıfın bir üyesi olarak dahil edilebilir. Tüm nesnelerin bu değişkene erişimi olacaktır. Her biri için bu aynı değişken olacaktır; her biri aynı toplamı görecektir.

Statik Sınıf Verisine Bir Örnek

STATDATA örneği basit bir statik veri üyesini tanıtıyor:

```
// statdata.cpp
// statik sinif verisi
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class foo
   {
   private:
      static int count;          //tum nesneler icin tek veri degiskeni
                                 //dikkat:sadece "deklarasyon"!!
   public:
      foo()                      //nesne olusturuldugunda sayaci artirir
         { count++; }
      int getcount()             //sayaci dondurur
         {return count;}
   };
//--------------------------------------------------------------
int foo::count = 0;              //sayacin *tanimi*
////////////////////////////////////////////////////////////////
int main()
   {
   foo f1, f2, f3;               //uc nesne olustur

   cout << "count is " << f1.getcount() << endl;  //her nesne
   cout << "count is " << f2.getcount() << endl;  //ayni degeri
   cout << "count is " << f3.getcount() << endl;  //gorur
   return 0;
   }
```

ŞEKİL 6.9: Statik ve otomatik üye değişkenler.

Bu örnekteki **foo** sınıfı tek bir veri parçası içerir. **count** isimli bu veri parçası **static int** tipindedir. Bu sınıf için tanımlı kurucu fonksiyon, **count**'un bir artırılmasını sağlar. **main()** içinde **foo** sınıfının üç nesnesi tanımlanır. Kurucu fonksiyon üç kere çağrıldığı için **count** da üç kere artırılmış olur. Bir diğer üye fonksiyon olan **getcount()**, **count** içinde tutulan değeri döndürür. Bu fonksiyon her üç nesneden de çağrılır; beklediğimiz gibi, her biri ekrana aynı değeri basar. Programın çıktısı işte şöyledir:

```
count is 3 <==== statik veri
count is 3
count is 3
```

Eğer `count` için statik bir değişken yerine sıradan bir otomatik değişken kullanmış olsaydık, her kurucu fonksiyon `count`'un kendisine ait özel kopyasını artırmış olacaktı, çıktı şu şekilde değişirdi:

```
count is 1 <==== otomatik veri
count is 1
count is 1
```

Statik sınıf değişkenleri, statik olmayan sıradan değişkenler kadar sık kullanılmaz; fakat bir çok durumda önemlidirler. Şekil 6.9'da statik değişkenlerle otomatik değişkenler karşılaştırılıyor.

Ayrı Ayrı Deklarasyon ve Tanım

Statik üye verisi alışılmamış bir format gerektirir. Sıradan değişkenler genellikle aynı ifade içinde deklare edilir (derleyiciye isimleri ve tipleri bildirilir) ve tanımlanır (derleyici değişkeni saklayabilmek için bellekte yer ayırır). Statik üye verisi ise, diğer yandan, ayrı iki ifade gerektirir. Değişkenin deklarasyonu sınıf tanımında yer alırken, değişken aslında, global değişkene çok benzer biçimde, sınıfın dışında tanımlanır.

Niye böyle iki parçalı bir yaklaşım kullanılır? Eğer statik üye verisi sınıfın içinde tanımlanmış olsaydı (ki, C++'ın ilk sürümlerinde gerçekte böyleydi), bu, sınıf tanımının sadece bir taslak olduğu ve bellekte yer ayırmadığı düşüncesi ile çelişmiş olacaktı. Statik üye verisinin tanımını sınıfın dışına koymak, böyle bir değişken için sadece bir kez, program çalışmaya başlamadan önce bellekte yer ayrıldığını ve bir statik üye değişkeninin tüm sınıf tarafından erişildiğini vurgulamaya yardımcı olur. Sıradan üye verisinde olduğu gibi, burada her nesne, değişkenin kendisine ait bir versiyonuna sahip değil. Bu şekliyle bir statik üye değişkeni daha çok bir global değişkeni andırıyor.

Statik verileri, hatalı biçimde ele almak çok kolaydır; üstelik, derleyici bu tür hatalar konusunda yardımcı da olmaz. Eğer bir statik değişkenin deklarasyonunu dahil edip, tanımını unutmuşsanız derleyiciden uyarı mesajı almazsınız. Bağlayıcıya gelene kadar her şey düzgün görünür; fakat bağlayıcı, tanımlanmamış bir global değişkene erişmeye çalıştığınızı size söyleyecektir. Tanımı dahil edip, sınıf ismini (`STATDATA` örneğinde `foo::` ifadesi) unuttuğunuz zaman da bu durumla karşılaşırsınız.

const ve Sınıflar

Normal değişkenleri değişikliğe uğramaktan korumak için `const` kullanıldığını birkaç örnek üzerinde görmüştük. Bölüm 5'te `const` niteleyicisinin fonksiyon argümanları ile kullanılabileceğini gördük. Böylece, bir fonksiyonun kendisine referans olarak aktarılan bir değişken üzerinde değişiklik yapması önlenebilir. Artık sınıfları da öğrendiğimize göre, `const`'ın başka kullanımlarını da tanıtabiliriz: `const`, üye fonksiyonlarda, üye fonksiyon argümanlarında ve nesneler üzerinde kullanılabilir. Bu kavramlar bir araya geldiğinde bazı şaşırtıcı avantajlar sağlarlar.

const Üye Fonksiyonları

Bir `const` üye fonksiyon, sınıfının üye verilerinden hiç birini asla değiştirmeyeceğini garanti eder. `CONSTFU` bunun nasıl gerçekleştiğini gösteriyor:

```
// constfu.cpp
// const uye fonksiyonlar
//
class aClass
   {
   private:
      int alpha;
   public:
      void nonFunc()
         { alpha = 99; }          //const olmayan uye fonksiyon
                                  //tamam

      void conFunc() const        //const uye fonksiyon
         { alpha = 99; }          //HATA: uyeyi degistiremezsiniz!
   };
```

Sabit olmayan bir fonksiyon olan `nonFunc()`, veri üyesi olan `alpha` üzerinde değişiklik yapabilir; fakat `conFunc()` yapamaz. Eğer yapmaya kalkarsa, derleme hatası ortaya çıkar.

Fonksiyon deklaratöründen sonra, fakat gövdesinden önce `const` anahtar kelimesi kullanılarak bir fonksiyon, sabit bir fonksiyon yapılabilir. Eğer ayrı bir fonksiyon deklarasyonu varsa, `const` hem deklarasyonda hem de tanımda kullanılmalıdır. Kullanıcıdan girdi almaktan başka bir iş yapmayan üye fonksiyonlar, veriler üzerinde bir değişiklik gerektirmedikleri için `const` yapılmaya açık birer adaydır.

Bir fonksiyonu `const` yapmak, yani sabit hale getirmek, derleyicinin hatalar yakalamasına yardımcı olur; ayrıca, söz konusu fonksiyonun, nesnelerinde hiçbir değişiklik yapmamasını istediğinizi program listesine bakan birine bildirir. Bir fonksiyonu sabit hale getirmek, ilaveten, `const` nesnelerinin oluşturulmasını ve kullanımını da mümkün kılar. `const` nesnelerini daha sonra ele alacağız.

Bir Distance Örneği

Bir anda çok fazla konuyu ortaya atmamak için, şimdiye kadar `const` üye fonksiyonlarını örnek programlarda kullanmaktan kaçındık. Ancak, `const` üye fonksiyonlarının kullanılması gereken pek çok yer var. Örneğin, birkaç tane programda karşımıza çıkan `Distance` sınıfında, `showdist()` üye fonksiyonu `const` yapılabilir; çünkü çağrıldığı nesne içindeki verilerin hiç biri üzerinde değişiklik yapmaz (aslında kesinlikle yapmamalıdır!). Bu fonksiyon sadece, verileri ekranda göstermekle görevlidir.

Ayrıca, `ENGLRET`'teki `add_dist()` fonksiyonu da, çağrıldığı nesne içindeki verilerin hiç birini değiştirmemelidir. Bu nesne sadece argüman olarak aktarılan nesne ile toplanmalı ve elde edilen toplam fonksiyondan döndürülmelidir. Bu iki sabit fonksiyonun nasıl görüneceğini göstermek için `ENGLRET` programını biraz değiştirdik. Dikkat ederseniz, `const` niteleyicisi `add_dist()`'in hem deklarasyonunda hem de tanımında kullanılır. `ENGCONST`'un program listesi işte şöyledir:

```
// engConst.cpp
// const uye fonksiyonlar ve uye fonksiyonlara const argumanlar
#include <iostream>
using namespace std;
```

```
////////////////////////////////////////////////////////////
class Distance                          //Ingiliz sistemine gore Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:                              //kurucu fonksiyon (argumansiz)
      Distance() : feet(0), inches(0.0)
         { }                            //kurucu fonksiyon (iki argumanli)
      Distance(int ft, float in) : feet(ft), inches(in)
         { }

      void getdist()                    //kullanicidan uzakligi al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }
      void showdist() const             //uzakligi ekranda goster
         { cout << feet << "\'-" << inches << '\"'; }

      Distance add_dist(const Distance&) const;          //topla
   };
//------------------------------------------------------------
//bu uzakligi d2'ye ekle, toplami dondur
Distance Distance::add_dist(const Distance& d2) const
   {
   Distance temp;                       //gecici degisken

//feet = 0;                             //HATA:bunu degistiremezsiniz
//d2.feet = 0;                          //HATA: d2'yi degistiremezsiniz
   temp.inches = inches + d2.inches;    //inches'leri topla
   if(temp.inches >= 12.0)              //eger toplam 12.0'yi gecerse,
      {                                 //inches'leri
      temp.inches -= 12.0;              //12.0 azalt ve
      temp.feet=1;                      //feet'i
      }                                 //1 artir
   temp.feet += feet + d2.feet;         //feet'leri topla
   return temp;
   }
////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist3;               //iki uzakligi topla
   Distance dist2(11, 6.25);            //dist2'yi tanimla, deger ata

   dist1.getdist();                     //kullanicidan dist1'i al
   dist3 = dist1.add_dist(dist2);       //dist3 = dist1 + dist2

                                        //tum uzakliklari ekranda goster
   cout << "\ndist1 = "; dist1.showdist();
   cout << "\ndist2 = "; dist2.showdist();
   cout << "\ndist3 = "; dist3.showdist();
   cout << endl;
   return 0;
   }
```

Bu programda **showdist()** ve **add_dist()**'in her ikisi de birer sabit üye fonksiyonudur. **add_dist()** içindeki açıklama içeren ilk ifadede, yani **feet = 0** ifadesinde, bu fonksiyonun

çağrıldığı nesnenin verilerinin herhangi biri üzerinde değişiklik yaparsanız, bir derleyici hatasının ortaya çıkacağını gösteriyoruz.

const Üye Fonksiyonu Argümanları

Eğer bir argüman bir fonksiyona referans olarak aktarılırsa ve siz bu fonksiyonun bu argümanı değiştirmesini istemiyorsanız argümanın, fonksiyonun deklarasyonunda (ve tanımında) **const** olarak tanımlanması gerektiğini Bölüm 5'te söylemiştik. Bu, üye fonksiyonlar için de geçerlidir. **ENGCONST**'ta **add_dist()**'e aktarılan argüman referans olarak aktarılır. **ENGCONST**'un **main()** içinde **dist2** olarak tanımlanan bu değişkeni değiştirmeyeceğinden emin olmak istiyoruz. Bu nedenle, **add_dist()**'in argümanı olan **d2**, hem deklarasyonda hem de tanımda **const** yapılır. Açıklama içeren ikinci ifade ile, **add_dist()**'in argümanı olan **dist2**'nin herhangi bir üye verisi üzerine **add_dist()** tarafından yapılacak bir değişiklik girişiminin, derleyicinin hata mesajı vermesine neden olacağı bildirilir.

const Nesneler

Birkaç örnek programda **const** niteleyicisinin, **int** gibi basit tipteki değişkenlere, bu değişkenlerin değerlerinin değişmesini önlemek amacıyla uygulanabileceğini görmüştük. Aynı şekilde, **const** niteleyicisini sınıfların nesnelerine de uygulayabiliriz. Bir nesne **const** olarak tanımlandığında, onu değiştiremezsiniz. Buradan şu noktaya geliyoruz: **const** olarak tanımlanmış nesneler ile sadece **const** üye fonksiyonlarını kullanabilirsiniz; çünkü böyle bir nesneyi sadece bu fonksiyonların değiştirmeme garantisi vardır. **CONSTOBJ** programı buna bir örnek veriyor.

```
// constObj.cpp
// const Distance nesneleri
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance                          //Ingiliz sistemine gore Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:                              //2 argumanli kurucu fonksiyon
      Distance(int ft, float in) : feet(ft), inches(in)
         { }
   void getdist()                       //kullanici girdisi; const olmayan fonksiyon
      {
      cout << "\nEnter feet: "; cin >> feet;
      cout << "Enter inches: "; cin >> inches;
      }
   void showdist() const                //uzakligi ekranda goster; const fonksiyon
      { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////
int main()
   {
   const Distance football(300, 0);

//football.getdist();                   //HATA:getdist() const degil
   cout << "football = ";
   football.showdist();                 //tamam
```

```
    cout << endl;
    return 0;
    }
```

Bir Amerikan futbolu sahası tam olarak 300 feet uzunluğundadır. Şayet bir programda böyle bir futbol alanının uzunluğunu kullanacak olursak, bunu **const** yapmak mantıklı olur; çünkü bu uzunluğu değiştirmek, futbol taraftarları için dünyanın sonu anlamına gelebilir. **CONSTOBJ** programı **footbal1** değişkenini **const** olarak tanım la. Artık sadece **const** fonksiyonlar, mesela **showdist()**, bu nesne için çağrılabilir. Sabit olmayan fonksiyonları, mesela kullanıcıdan alınan yeni değeri nesneye atayan **getdist()** fonksiyonunu bu nesne için çağırmak kural dışıdır. Bu şekilde derleyici, **footbal1** değişkeni için sabit (**const**) bir değer kullanılmasını zorlamış olur.

Sınıfları tasarlarken, çağrıldıkları nesnedeki verilerin hiç birini değiştirmeyen fonksiyonların her birini **const** olarak tanımlamak iyi bir fikirdir. Bu, sınıfın kullanıcısının **const** nesneler oluşturmasına imkan verir. Bu nesneler **const** fonksiyonların tümünü kullanabilir; fakat **const** olmayanların hiç birini kullanamaz. Şunu hatırınızdan çıkarmayın: **const** kullanmak derleyicinin size yardımcı olmasına yardımcı olur.

Bütün Bunlar Ne İşe Yarıyor?

Artık sınıflar ve nesnelerle tanıştığınıza göre, bunların gerçekten ne tür avantajlar sağladığını merak ediyor olabilirsiniz. Nitekim, bu bölümdeki birkaç programı Bölüm 4'tekilerle karşılaştırdığınızda, prosedürel bir yaklaşımla yapılanlara benzer şeylerin nesnelerle de yapılabileceğini görebilirsiniz.

Sizin de zaten gözünüze ilişmiş olabilir; nesne yönelimli programlamanın sağladığı bir avantaj, program *tarafından* modellenen gerçek dünyaya ait olguların programın *içinde* kullanılan C++ nesneleri ile çok yakından uyuşuyor olmasıdır. Programdaki bir yedek parça nesnesi gerçek dünyadaki bir yedek parçayı simgeler; bir iskambil kartı nesnesi bir iskambil kartını, bir daire nesnesi ise bir grafik daireyi simgeler. C++'ta bir yedek parçayla ilgili her türlü bilgi, örneğin parça numarası ve diğer veriler; ayrıca, bu verilere erişmek ve bunlar üzerinde işlem yapmak için gerekli fonksiyonlar sınıf tanımına dahil edilir. Böylece bir programlama problemini kavramsal bazda ele almak daha kolay olur. Problemin hangi bölümlerinin nesneler yardımıyla en kullanışlı biçimde simgeleneceğini tespit eder ve bu nesneyle bağlantılı tüm veri ve fonksiyonları o sınıfın içine yerleştirirsiniz. Eğer bir oyun kartını simgelemek için bir C++ sınıfı kullanıyorsanız, bu sınıfın içine kartın değerini simgeleyen veri parçalarını koyarsınız; ayrıca, değer atamak, değere ulaşmak, ekranda görüntülemek, karşılaştırmak vb. işlemler için gerekli fonksiyonları da sınıfın içine yerleştirirsiniz.

Prosedürel bir programda ise, tüm bunların aksine, bir gerçek dünya nesnesi ile bağlantılı global değişkenler ve fonksiyonlar tüm program içinde dağınık olarak yer alırlar; kolaylıkla yakalanabilen tek bir birim oluşturmazlar.

Bazı durumlarda gerçek hayatın hangi parçalarının nesne olarak simgeleneceği belli olmayabilir. Satranç oynayan bir program yazıyorsanız, örneğin, nesneler nedir? Satranç oynayan oyuncu mu, satranç tahtasının üzerindeki kareler mi, yoksa tahta üzerindeki muhtemel pozisyonların tamamı mı?

Bu kitaptaki örneklerin bir çoğu gibi küçük programlarda genellikle deneme-yanılma yoluyla ilerleyebilirsiniz. Problemi bir şekilde nesnelere ayırıp, bu nesneler için deneme amaçlı sınıf tanımları yazarsınız. Şayet sınıflar gerçekle mantıklı bir biçimde örtüşüyor gibi görünüyorsa, devam edersiniz. Örtüşmüyorsa, farklı bütünleri sınıf olarak seçip sil baştan başlamak

zorunda kalabilirsiniz. Nesne yönelimli programlama deneyimiz arttıkça, programlama problemlerini sınıflara ayırmak da kolaylaşacaktır.

Daha büyük problemler bu deneme-yanılma yaklaşımı için fazlasıyla karmaşık kalabilir. Yeni bir alan olan nesne yönelimli tasarım, programlama problemlerinin analizinde ve gerçek dünyadaki durumları (bu, çoğunlukla *problem etki alanı -problem domain-* olarak adlandırılır) simgelemek için hangi nesne ve fonksiyonların kullanılacağının tespitinde artan sıklıkta uygulanır. Bu metodolojiyi, Bölüm 16'da "Nesne yönelimli Yazılım Geliştirme" başlığı altında ayrıntılı olarak ele alacağız.

Nesne yönelimli programlamanın bazı avantajları şu noktada muhtemelen pek belli değildir. Unutmayın ki, nesne yönelimli programlama büyük problemlerin karmaşıklığı ile başa çıkmak için tasarlanmıştır. Bu bölümdeki örnekler gibi daha küçük programlar, nesne yönelimli programlamanın sağladığı organizasyonel güce daha az ihtiyaç duyarlar. Program büyüdükçe, sağlanan avantaj da artar. Fakat, küçük programlarda bile, bir kez nesne yönelimli programlama terimleri ile düşünmeye başladınız mı, nesne yönelimli tasarım yaklaşımı doğal ve şaşırtıcı ölçüde kullanışlı hale gelir. Prosedürel programlara nazaran nesne yönelimli programlarda, derleyicinin daha fazla kavramsal hata yakalayabilmesi nesne yönelimli tasarımın sunduğu bir avantajdır.

Özet

Bir sınıf, birkaç sayıda nesne için bir spesifikasyon veya bir taslaktır. Nesneler hem veri, hem de bu veriler üzerinde işlem yapacak olan fonksiyonları içerirler. Bir sınıf tanımında, üyeler – veri veya fonksiyonlar – **private** (özel) veya **public** (açık) olabilir. private üyeler sadece kendi sınıflarının üye fonksiyonları tarafından, public üyeler ise program içindeki herhangi bir fonksiyon tarafından erişilebilir.

Bir üye fonksiyon, bir sınıfın üyesi olan fonksiyondur. Üye fonksiyonlar, bir nesnenin özel verilerine erişebilir, fakat üye olmayan fonksiyonlar erişemez.

Bir kurucu fonksiyon, kendi sınıfı ile aynı isme sahip bir üye fonksiyondur. Ne zaman bu sınıfa ait bir nesne oluşturulsa, kurucu fonksiyon çalıştırılır. Bir kurucu fonksiyonun dönüş değeri olmaz, fakat kurucu fonksiyon argüman alabilir. Bir kurucu fonksiyon genellikle nesnenin veri üyelerini ilk kullanıma hazırlamak için kullanılır. Kurucu fonksiyonlar aşırı yüklenebilir; bu nedenle, bir nesneye farklı yollardan başlangıç değeri atanabilir.

Bir yok edici fonksiyon, kendi sınıfı ile aynı isme sahip bir üye fonksiyondur. Yok edici fonksiyonda, fonksiyon isminden önce tilde simgesi (~) kullanılır. Yok edici fonksiyon, bir nesne ortadan kaldırıldığında çağrılır. Bir yok edici fonksiyon argüman almaz ve dönüş değeri içermez.

Bilgisayarın belleğinde, bir sınıftan oluşturulmuş her nesne için veri üyelerinin ayrı birer kopyası vardır, fakat sınıfın üye fonksiyonlarının yalnızca tek kopyası vardır. Bir veri parçasını **static** olarak adlandırarak, böyle bir veri parçasını, bir sınıfın tüm nesneleri için tek bir örnek olarak sınırlandırabilirsiniz.

Nesne yönelimli programlama kullanmanın bir sebebi, gerçek dünyaya ait nesneler ile nesne yönelimli programlama sınıflarının birbiriyle yakından örtüşmesidir. Bir program içinde hangi nesne ve sınıfları kullanacağımıza karar vermek karmaşık olabilir. Küçük programlar için deneme-yanılma yeterli olabilir. Büyük programlar için, çoğunlukla daha sistematik bir yaklaşım gerekir.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Bir sınıf tanımının amacı nedir?
2. Basit bir veri tipi ile o tipteki arasındaki ilişki _____ ile _____ arasında vardır.
3. Bir sınıf tanımında **private** olarak belirtilen veri veya fonksiyonların erişilebilirliği ile ilgili aşağıdakilerden hangisi doğrudur?
   a. Program içindeki herhangi bir fonksiyon tarafından erişilebilirler.
   b. Sadece şifreyi biliyorsanız erişebilirsiniz.
   c. O sınıfın üye fonksiyonları tarafından erişilebilirler.
   d. Sadece o sınıfın açık (public) üyeleri tarafından erişilebilirler.
4. **leverage** isminde bir sınıf tanımı yazın. Sınıfın **crowbar** adında, **int** tipinde bir tane özel veri üyesi ve deklarasyonu **void pry()** olan bir tane **public** fonksiyonu olsun.
5. Doğru/Yanlış: Bir sınıf içindeki veri parçaları özel olmalıdır.
6. Soru 4'te tanımlanan **leverage** sınıfının bir nesnesi olan **lever1**'i tanımlayan bir ifade yazın.
7. Nokta operatörü (veya sınıf üyelerine erişim operatörü) aşağıdaki hangi iki olguyu birbirine bağlar (soldan sağa okunarak)?
   a. Bir sınıf üyesi ve bir sınıf nesnesi
   b. Bir sınıf nesnesi ve bir sınıf
   c. Bir sınıf ve o sınıfın bir üyesi
   d. Bir sınıf nesnesi ve o sınıfın bir üyesi
8. Soru 4 ve 6'ta bahsedilen **lever1** nesnesinin fonksiyonu olan **pry()**'ı çalıştıracak bir ifade yazın.
9. Bir sınıfın tanımında tanımlanan üye fonksiyonlar varsayılan durumda _____ .
10. Soru 4'te bahsedilen **leverage** sınıfı için **getcrow()** adında bir üye fonksiyon yazın. Bu fonksiyon **crowbar** verisinin değerini döndürmelidir. Fonksiyonun sınıf tanımı içinde tanımlandığını varsayın.
11. Bir nesne _____ otomatik olarak bir kurucu fonksiyon çalıştırılır.
12. Bir kurucu fonksiyonun ismi _____ ismi ile aynıdır.
13. Soru 4'te bahsedilen **leverage** sınıfının bir üyesi olan **crowbar** verisine 0 değeri atayan bir kurucu fonksiyon yazın. Kurucu fonksiyonun sınıf tanımı içinde tanımlandığını varsayın.
14. Doğru/Yanlış: Bir sınıf içinde aynı ismi taşıyan birden fazla kurucu fonksiyon olabilir.
15. Bir üye fonksiyon _____ içindeki verilere her zaman erişebilir.
    a. üyesi olduğu nesnenin
    b. üyesi olduğu sınıfın
    c. üyesi olduğu sınıfın herhangi bir nesnesinin
    d. kendi sınıfının açık (public) bölümünün
16. Soru 10'daki **getcrow()** fonksiyonunun sınıfı tanımının dışında tanımlandığını varsayalım. Sınıf tanımının içine geçecek bir deklarasyon yazın.
17. Soru 10'daki **getcrow()** fonksiyonunun, sınıf tanımı dışında tanımlanacak şekilde yeni bir versiyonunu yazın.
18. C++'ta yapılar ve sınıflar arasındaki teknik açıdan tek farklılık _____ .

19. Eğer bir sınıfın üç nesnesi tanımlanmışsa, bellekte bu sınıfın veri parçalarının kaç kopyası saklanır? Üye fonksiyonlarının kaç kopyası saklanır?
20. Bir nesneye bir mesaj göndermek _____ ile aynıdır.
21. Sınıflar kullanışlıdır çünkü
    a. kullanımda olmadıkları zaman bellekten çıkarılırlar.
    b. verilerin diğer sınıflardan saklanmasına imkan verirler.
    c. bir bütünle ilgili tüm özellikleri bir araya getirirler.
    d. gerçek dünyadaki nesneleri oldukça yakından modelleyebilirler.
22. Doğru/Yanlış: Gerçek dünyaya ait bir programlama problemini sınıflara ayırmanın basit ve kapsamlı bir yöntemi vardır.
23. Çağrıldığı nesne için bir **const** üye fonksiyon
    a. **const** ve sabit olmayan üye verileri değiştirebilir.
    b. sadece sabit üye verileri değiştirebilir.
    c. sadece sabit olmayan üye verileri değiştirebilir.
    d. ne sabit verileri ne de sabit olmayan verileri değiştiremez.
24. Doğru/Yanlış: Eğer bir **const** nesne tanımlarsanız, bu yalnızca **const** üye fonksiyonlarla kullanılabilir.
25. **float** tipinde **jerry** adında bir tane **const** argüman alan bir **const void** fonksiyon deklarasyonu (tanımı değil) yazın. Fonksiyonun ismi **aFunc()** olsun.

Alıştırmalar

Yıldızla işaretli soruların cevaplarını Ek G'de bulabilirsiniz.

1. Basit bir veri tip olan **int**'in işlevselliğinin bir bölümünü taklit eden bir sınıf oluşturun. Sınıfa **Int** ismini verin (baş harfin büyük olduğuna dikkat edin). Bu sınıfın içindeki tek veri, **int** tipinde bir değişken olmalıdır. Sınıfın içine şu işleri yapan üye fonksiyonlar dahil edin: **Int**'e 0 değeri atayan, bir **int** değerini atayan, içindeki değeri ekranda gösteren ve iki **Int** değerini toplayan üye fonksiyonlar
   Bu sınıfı kullanan bir program yazın. Programınız, bir tane değeri atanmamış, iki tane değeri atanmış **Int** değerleri oluştursun, değeri atanmamış olanların ikisini toplasın, sonucu başlangıç değeri atanmamış değere yerleştirsin ve sonucu ekrana yazsın.
2. Bir köprüdeki para ödeme gişelerini düşünün. Gişeden geçen arabaların geçiş için 50 cent ödemeleri bekleniyor. Genellikle arabalar bu parayı ödüyor; fakat, arada sırada bir araba ödemeden geçebiliyor. Gişe memuru, geçen arabaların sayısını ve toplanan toplam para miktarını takip ediyor.
   Bu para gişesini **tollBooth** isimli bir sınıf kullanarak modelleyin. Sınıfın iki tane veri üyesi olmalı. Bunlardan biri, **unsigned int** tipinde olup, toplam araba sayısını tutmalı; diğeri, **double** tipinde olup, toplanan toplam para miktarını tutmalıdır. Bir kurucu fonksiyon başlangıçta her ikisini de 0'a eşitlemelidir. **payingCar()** isimli bir üye fonksiyon arabaların sayısını bir artırıp, nakit toplamına da 0.5 eklemelidir. **nopayCar()** isimli bir başka fonksiyon, araba sayısını artırmalı ama, nakit toplamına bir şey eklememelidir. Son olarak, **display()** isimli bir üye fonksiyon iki toplamı ekranda göstermelidir. Uygun olan üye fonksiyonları **const** olarak kullanın.
   Bu sınıfı denemek için bir program ilave edin. Bu program, kullanıcının ödeme yapan bir arabayı saymak için bir tuşa, yapmayanı saymak için başka bir tuşa basmasını iste-

sin. Esc tuşuna basmak programın ekrana toplam araba sayısını ve nakit miktarını yazmasına ve çıkmasını sağlamalı.

3. `time` isimli bir sınıf oluşturun. Saat, dakika ve saniye için ayrı ayrı `int` tipinde üye verileri olsun. Bir kurucu fonksiyon bu verilere 0 değerini, başka bir fonksiyon sabit bir değer atamalı. Bir başka üye fonksiyon bu değerleri ekranda 11:59:59 formatında göstermeli. Sonuncu üye fonksiyon ise argüman olarak aktarılan time tipinde iki nesneyi toplamalı.
   Bir `main()` programı iki tane başlangıç değeri olan (`const` olmaları gerekir mi?), bir tane de başlangıç değeri olmayan `time` nesneleri oluşturmalı. Program, değerleri atanmış olan iki nesneyi toplayıp, sonucu üçüncü `time` değişkenine atamalı. Uygun olan üye fonksiyonları `const` olarak kullanın.
4. Bölüm 4'te Alıştırma 4'teki yapıya dayanarak bir `employee` sınıfı oluşturun. Üye veriler, çalışanın (personelin) numarasını saklamak için bir `int` ve çalışanın tazminatını saklamak için de bir `float` içermeli. Üye fonksiyonlar kullanıcının bu verileri girmesini sağlamalı ve bu değerleri ekranda görüntülemeli. Kullanıcının verileri girmesine imkan veren ve bunları ekranda gösteren bir `main()` programı yazın.
5. Bölüm 4, Alıştırma 5'teki `date` yapısı ile başlayın ve bunu bir `date` sınıfına dönüştürün. Bu sınıfın üye verileri üç tane `int`'ten oluşmalı: `month`, `day` ve `year`. Sınıfın ayrıca iki tane üye fonksiyonu olmalı: `getdate()`, kullanıcının tarihi 12/31/02 formatında girmesini sağlamalı; `showdate()` ise tarihi ekranda göstermeli.
6. Alıştırma 4'teki `employee` sınıfını, `date` sınıfını (Alıştırma 5'e bakın) ve bir `etype` `enum` tipini (Bölüm 4, Alıştırma 6'ya bakın) de dahil edecek biçimde geliştirin. `date` sınıfının bir nesnesi, çalışanın işe başlama tarihini tutmak için kullanılmalı. `etype` değişkeni ise çalışanın tipini tutmak için kullanılmalı: işçi, sekreter, yönetici vs. Bu iki unsur, tıpkı çalışanın numarası ve maaşı gibi, employee tanımında özel üye verileri olmalı. `getemploy()` ve `putemploy()` fonksiyonlarını, kullanıcıdan bu yeni bilgileri de alıp ekranda gösterecek biçimde geliştirmeniz gerekir. Bu fonksiyonlar, `etype` değişkenini ele alabilmek için muhtemelen bir `switch`'e ihtiyaç duyacaktır. Üç `employee` değişkeni için kullanıcının veri girmesini sağlayan ve bu değerleri ekranda gösteren bir `main()` programı yazın.
7. Denizcilikte konumlar, enlem ve boylama göre derece ve dakika cinsinden ölçülür. Yani, Tahiti'de Papeete Harbor'un ağzında güneşleniyorsanız, konumunuz 149 derece 38.4 dakika batı boylamı ve 17 derece 31.5 dakika güney enlemidir. Bu, 149° 34.8' W, 17° 31.5' S şeklinde yazılır. Bir derecede 60 dakika vardır. (Eski sistemde bir dakika 60 saniyeye bölünürdü, fakat modern sistemde bunun yerine ondalık dakikalar kullanılır.) Boylam 0 ile 180 derece arasında ölçülür; Greenwich'ten (Londra) başlayarak Pasifik'teki uluslar arası zaman sınırına kadar doğu veya batı olarak adlandırılır. Enlem 0 ile 90 derece arasında ölçülür; ekvatordan kutuplara kadar kuzey veya güney olarak adlandırılır.
   Üç tane üye değişkeni olan `angle` isminde bir sınıf oluşturun: derece için `int` tipinde bir değişken, dakika için `float` tipinde bir değişken ve yönü tayin eden harf için (N, S, E veya W) `char` tipinde bir değişken. Bu sınıf ya bir enlem ya da boylam değişkenini tutabilir. Kullanıcıdan bir açı değerini (derece ve dakika cinsinden) ve yönü alan bir üye fonksiyon yazın. İkinci bir üye fonksiyon bu girilen açı değerini 179°59.9' E formatında ekranda göstersin. Ayrıca, üç argümanlı bir kurucu fonksiyon yazın. Bu kurucu fonksiyonun değer atadığı bir açıyı ekranda gösteren bir `main()` programı yazın. Programda bir döngü içinde kullanıcıdan her hangi bir açı değeri girmesini sağlayın ve değeri ekranda

gösterin. (°) simgesini ekranda bastırmak için `'\xF8'` onaltılık karakter sabitini kullanabilirsiniz.

8. Bir sınıf oluşturun. Bu sınıfın bir üye verisi, bu sınıftan oluşturulan her nesne için bir "seri numarası" tutsun. Yani, ilk oluşturulan nesne 1, ikinci 2, vs. olarak numaralandırılacaktır.
   Bunu yapmak için, o ana dek oluşturulan nesnelerin sayısını kaydeden başka bir veri üyesine ihtiyacınız olacak. (Bu üye, tek tek nesneler üzerinde değil de, sınıfın tümü üzerinde kullanılmalı. Bunu hangi anahtar kelime açıkça belirtir?) Sonra, yeni bir nesne oluşturuldukça, o nesnenin kurucu fonksiyonu, yeni nesneye uygun bir seri numarası belirlemek için bu sayıyı inceleyebilir.
   Bir nesnenin kendi seri numarasını bildirmesine imkan veren bir üye fonksiyon ekleyin. Sonra, üç nesne oluşturan ve her nesneye seri numarasını soran bir `main()` programı yazın. Nesneler şu şekilde yanıt vermeli: `I am object number 2`
9. Bölüm 4, Alıştırma 8'deki `fraction` yapısını, `fraction` sınıfına dönüştürün. Kesrin payı ve paydası üye veriler olsun. Üye fonksiyonlar kullanıcıdan 3/5 formatında girdi alsın ve kesrin değerini aynı formatta ekranda göstersin. Başka bir üye fonksiyon iki kesir değerini toplasın. Kullanıcıdan iki kesir girmesini isteyen, bunların toplamlarını ekranda gösteren ve işlemi tekrarlayan bir `main()` programı yazın. Her işlemden sonra kullanıcıya devam etmek isteyip istemediğini sorun.
10. Bir geminin numarasını ve konumunu belirten `ship` isminde bir sınıf oluşturun. Her `ship` nesnesi oluşturulduğunda onu numaralandırmak için Alıştırma 8'deki yaklaşımı kullanın. Geminin bulunduğu enlem ve boylamı simgelemek için Alıştırma 7'deki `angle` sınıfı tipinde iki değişken kullanın. `ship` sınıfının bir üye fonksiyonu kullanıcıdan konum bilgisini alıp, bu bilgiyi bir nesne içine yerleştirmeli. Başka bir üye fonksiyon seri numarasını ve konumu bildirmeli. Üç tane gemi oluşturan, her birinin pozisyonlarını kullanıcıdan alan ve her birinin numarasını ve konumunu ekranda gösteren bir `main()` programı yazın.
11. Bölüm 5, Alıştırma 12'deki dört fonksiyonlu kesirli işlem yapan hesap makinesini, bir yapı yerine `fraction` sınıfını kullanacak biçimde değiştirin. Dört aritmetik işleminin yanı sıra girdi ve çıktı için de üye fonksiyonlar olmalı. Bu noktada, kesirleri sadeleştirme özelliğini de dahil edebilirsiniz. İşte size bir üye fonksiyon. Bu fonksiyon, üyesi olduğu `fraction` nesnesini sadeleştirmeye yarıyor. Fonksiyon, kesrin payı ve paydasının en büyük ortak bölenini (EBOB) buluyor; sonra payı ve paydayı bölmek için bu EBOB'u kullanıyor.

```
void fraction::lowterms()        // kesri sadelestir
   {
   long tnum, tden, temp, gcd;

   tnum = labs(num);             // negatif olmayan kopyalari kullan
   tden = labs(den);             //    (cmath gerektirir)
   if(tden==0)                   // n/0 durumunu kontrol et
      { cout << "Illegal fraction: division by 0"; exit(1); }
   else if(tnum==0)              // 0/n durumunu kontrol et
      {num=0; den=1; return;}

   //bu 'while' dongusu, tnum ve tden'in EBOB'unu bulur
   while(tnum != 0)
      {
      if(tnum < tden)               //payin daha buyuk oldugundan emin ol
```

```
        { temp=tnum; tnum=tden; tden=temp; }  //yerlerini degistir
    tnum = tnum - tden;           //cikar
    }
gcd = tden;                 //bu, en buyuk ortak bolen
num = num /gcd;             //kesiri sadelestirmek icin
den = den /gcd;             //num ve den'in her ikisini de EBOB'a bol
}
```

Her aritmetik işlemin sonunda veya çıktıyı yazdırmadan hemen önce bu fonksiyonu çağırabilirsiniz. Ayrıca olağan üye fonksiyonlarına da ihtiyacınız olacak: dört aritmetik işlemi için, girdi ve çıktı için. İki argümanlı bir kurucu fonksiyonu kullanışlı bulabilirsiniz.

12. Dikkat ederseniz, nesne yönelimli programlamanın sağladığı bir avantaj, bir sınıfın tamamının, üzerinde hiçbir değişiklik yapmadan farklı bir programda kullanılabilmesidir. Alıştırma 11'deki **fraction** sınıfını, kesirli sayılar için bir çarpım tablosu üreten bir programda kullanın. Kullanıcının paydayı girmesini sağlayın. Bu paydayı kullanarak 0 ve 1 arasındaki iki kesir için olası tüm kombinasyonları üretin ve bunları çarpın. İşte size çıktının bir örneği; payda 6 ise, çıktı şöyle olur:

```
        1/6     1/3     1/2     2/3     5/6
-----------------------------------------
1/6     1/36    1/18    1/12    1/9     5/36
1/3     1/18    1/9     1/6     2/9     5/18
1/2     1/12    1/6     1/4     1/3     5/12
2/3     1/9     2/9     1/3     4/9     5/9
5/6     5/36    5/18    5/12    5/9     25/36
```

YEDİNCİ BÖLÜM

7

DİZİLER VE KARAKTER KATARLARI

Dizilerin Temel Özellikleri

Bir Sınıfın Üye Verisi Olarak Diziler

Nesne Dizileri

C Karakter Katarları

Standart C++ string Sınıfı

Günlük yaşantımızda genellikle benzer nesneleri birimler halinde gruplarız. Bezelyeleri teneke kutu içinde, yumurtaları karton kutuda satın alırız. Bilgisayar dillerinde de aynı tipte veri parçalarını bir araya getirip, gruplamaya ihtiyaç duyarız. C++'ta bunu başaran en temel mekanizma *dizidir*. Diziler, birkaç tane veri parçasını veya on binlercesini saklayabilir. Bir dizi içinde gruplanmış veri parçaları `int` veya `float` gibi temel veri tipleri olabilir; ya da yapılar ve nesneler gibi kullanıcı tarafından tanımlanan tipler olabilir.

Diziler, yapılar gibidir. Şöyle ki, bunların her ikisi de birkaç sayıda veriyi daha büyük bir birim içinde gruplarlar. Fakat bir yapı genellikle farklı tipteki verileri gruplarken, bir dizi aynı tipteki verileri gruplar. Daha da önemlisi, bir yapı içindeki öğelere isim ile erişilirken, dizinin öğelerine bir indeks sayısı ile erişilir. Bir öğeyi açıkça belirtmek için bir indeks numarası kullanmak fazla sayıdaki öğelere kolay erişim imkanı verir.

Diziler neredeyse her bilgisayar dilinde mevcuttur. C++'taki diziler diğer dillerdekine benzer; C'dekilerle ise hemen hemen aynıdır.

Bu bölümde öncelikle, `int` ve `char` gibi temel veri tipindeki dizilere göz atacağız. Daha sonra, sınıflarda veri üyesi olarak kullanılan dizileri ve nesneleri saklamak için kullanılan dizileri inceleyeceğiz. Yani, bu bölüm sadece dizilere bir giriş yapmakla kalmıyor, aynı zamanda nesne yönelimli programlamayı daha iyi anlamınızı tasarlıyor.

Standart C++'ta bir dizi, aynı tipteki öğeleri gruplamanın tek yolu değildir. Standart Şablon Kütüphanesinin (Standard Template Library) bir parçası olan *vektör*, bir başka yaklaşımdır. Vektörlere Bölüm 15'te, "Standart Şablon Kütüphanesi" başlığı altında göz atacağız.

Bu bölümde ayrıca, metinleri saklamak ve üzerlerinde değişiklik yapmak için kullanılan karakter katarlarına (string) iki farklı açıdan bakacağız. Birinci tip karakter katarı, `char` tipinde bir dizi; ikincisi ise Standart C++ `string` sınıfının bir üyesidir.

Dizilerin Temel Özellikleri

Dizileri tanıtmak amacıyla basit bir örnek program kullanacağız. **REPLAY** isimli bu program, dört kişinin yaşlarını simgeleyen dört tamsayılık bir dizi oluşturur. Sonra, kullanıcının dört değer girmesini ister ve bu değerleri diziye yerleştirir. Son olarak, bu dört değeri ekranda gösterir.

```
// replay.cpp
// kullaniciya dort kisinin yasini sorar, bunlari ekranda goruntuler
#include <iostream>
using namespace std;

int main()
   {
   int age[4];                    //4 tamsayi ogeli 'age' dizisi

   for(int j=0; j<4; j++)         //4 kisinin yasini sor
      {
      cout << "Enter an age: ";
      cin >> age[j];              //dizi elemanina eris
      }
   for(j=0; j<4; j++)             //4 yasi ekranda goster
      cout << "You entered " << age[j] << endl;
   return 0;
   }
```

Programla etkileşim şöyle olabilir:


```
Enter an age: 44
Enter an age: 16
Enter an age: 23
Enter an age: 68

You entered 44
You entered 16
You entered 23
You entered 68
```

Birinci `for` döngüsü kullanıcıdan yaşları alır ve bunları diziye yerleştirir; ikinci döngü bu değerleri diziden okur ve ekranda görüntüler.

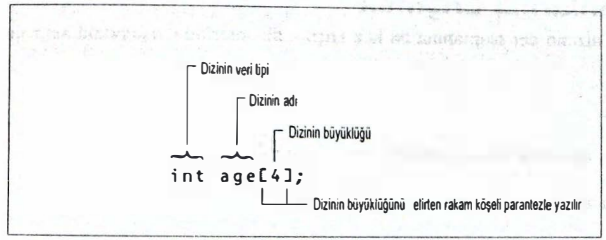
Dizileri Tanımlamak

C++'taki diğer değişkenler gibi bir dizi de, bilgi depolamak amacıyla kullanılmadan önce tanımlanmalıdır. Diğer tanımlar gibi, bir dizi tanımında bir değişken tipi ve ismi açıkça belirtilir. Fakat, dizi tanımı bir özellik daha içerir: Büyüklük. Büyüklük, bir dizinin kaç tane veri öğesi içerebileceğini belirtir. Büyüklük, dizi isminin hemen peşinden gelir ve köşeli parantez ile çevrilidir. Şekil 7.1 bir dizi tanımının söz dizimini gösteriyor.

REPLAY örneğinde dizi `int` tipindedir. Tipin ardından dizinin ismi gelir; ismi, hemen peşinden açılış parantezi, büyüklük ve kapanış parantezi izler. Köşeli parantezlerin içindeki sayı ya bir sabit ya da değeri bir sabit ile sonuçlanan bir deyim olmalıdır. Ayrıca bu sayı bir tamsayı olmalıdır. Örnekte, 4 değerini kullanılmıştır.

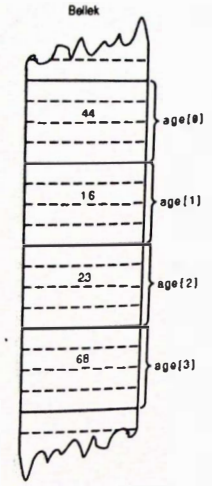
Dizi Elemanları

Bir diziyi oluşturan parçalara *eleman* denir (bu, bir yapının içerdiği parçalardan farklıdır; yapıyı oluşturan parçaların her biri *üye* olarak adlandırılır). Önceden belirttiğimiz gibi, bir dizinin tüm elemanları aynı tiptedir; sadece değerler değişir. Şekil 7.2 **age** isimli bir dizinin elemanlarını gösteriyor.



ŞEKİL 7.1: Dizi tanımının söz dizimi.

Geleneksel yaklaşıma uygun olarak (gerçi geleneksel yaklaşımda bellek kimi zaman geriye doğru büyüse de), şekildeki bellek aşağıya doğru büyür. Yani, dizinin ilk elemanı sayfanın üst tarafında yer alır; diğer elemanlar da aşağıya doğru uzanır. Tanımda da belirtildiği gibi, dizi tam olarak dört eleman içerir.



ŞEKİL 7.2: Dizi elemanları.

Dikkat ederseniz, dizinin birinci elemanı 0 olarak numaralandırılır. Dolayısıyla, toplam dört eleman olduğu için en sondaki, 3 numaralı eleman olur. Bu potansiyel olarak aklınızı karıştırabilecek bir durumdur; dört elemanlı bir dizinin sonuncu elemanının 4 numara olduğunu düşünebilirsiniz ama, öyle değildir.

Dizi Elemanlarına Erişmek

REPLAY örneğinde dizinin her elemanına iki kez erişilir. İlk seferinde, aşağıdaki satır ile diziye bir değer eklenir:

```
cin >> age[j];
```

İkinci seferinde, aşağıdaki satır ile diziden bir değer okunur:

```
cout << "\nYou entered " << age[j];
```

Her iki durumda da dizi elemanları için şu deyim kullanılır:

```
age[j]
```

Bu deyim, dizinin ismini ve peşinden **j** değişkenini sınırlandıran köşeli parantezleri içerir. Bu deyim ile dizinin dört elemanından hangisinin kast edildiği **j**'nin değerine bağlıdır. **age [0]**, birinci elemana; **age [1]**, ikinci elemana; **age [2]**, üçüncü elemana; **age [3]**, dördüncü elemana karşılık gelir. Köşeli parantez içindeki değişken (veya sabit) *dizi indeksi* olarak adlandırılır.

j, her iki **for** döngüsünde de döngü değişkeni olduğu için 0'dan başlar ve 3'e ulaşana kadar birer birer artırılır. Böylece, **j** değişkeni sırayla dizinin her elemanına erişir.

Dizi Elemanlarının Ortalamasını Almak

İşte, iş başındaki dizilere bir örnek daha. **SALES** isimli bu örnek kullanıcıyı, haftanın her günü için (Pazar hariç), yedek parça satışlarını simgeleyen altı değerden oluşan bir seri girdi girmeye davet eder; sonra, bu değerlerin ortalamasını alır. Programda **double** tipinde bir dizi kullanılır; böylece parasal değerler de programa girilebilir.

```
// sales.cpp
// bir haftanin yedek parca satislarinin ortalamasini alir(6 gun icin)
#include <iostream>
using namespace std;

int main()
   {
   const int SIZE = 6;                    //dizinin buyuklugu
   double sales[SIZE];                    //6 degiskenli bir dizi

   cout << "Enter widget sales for 6 days\n";
   for(int j=0; j<SIZE; j++)              //rakamlari diziye yerlestir
      cin >> sales[j];

   double total = 0;
   for(j=0; j<SIZE; j++)                  //rakamlari diziden oku *
      total += sales[j];                  //toplami bulmak icin
   double average = total / SIZE;         //ortalamayi bul
   cout << "Average = " << average << endl;
   return 0;
   }
```

İşte, **SALES** ile örnek bir etkileşim:

```
Enter widget sales for 6 days
352.64
867.70
781.32
867.35
746.21
189.45
Average = 634.11
```

Bu programla ilgili yeni bir ayrıntı, dizi büyüklüğü ve döngü limiti için bir **const** değişken kullanımıdır. Bu değişken program listesinin en başında tanımlanır:

```
const int SIZE = 6;
```

En son örnekte kullanılan 4 gibi, bir sayı yerine bir değişken kullanmak dizi büyüklüğünü değiştirmeyi kolaylaştırır: Dizi büyüklüğünü, döngü limitlerini ve program içinde dizinin geçtiği yerlerin tümünü değiştirmek için sadece tek bir program satırını değiştirmek yeterli olur. Tümü büyük harflerden oluşan isim, değişkenin program içinde değiştirilemeyeceğini bize hatırlatır.

Dizileri İlk Kullanıma Hazırlamak

Bir dizi ilk kez tanımlandığında dizi elemanlarının her birine bir değer verebilirsiniz. DAYS isimli örnekte, **days_per_month** dizisinin 12 elemanına, karşılık gelen ayın kaç gün olduğu değer olarak verilir:

```
// days.cpp
// yilbasindan belirtilen tarihe kadar gecen gunleri gosterir
#include <iostream>
using namespace std;

int main()
   {
   int month, day, total_days;
   int days_per_month[12] = { 31, 28, 31, 30, 31, 30,
                              31, 31, 30, 31, 30, 31 };

   cout << "\nEnter month (1 to 12): ";    //tarihi al
   cin >> month;
   cout << "Enter day (1 to 31): ";
   cin >> day;
   total_days = day;                       //ayri gunler
   for(int j=0; j<month-1 ;j++)            //her ay icin gunleri topla
      total_days += days_per_month[j];
   cout << "Total days from start of year is: " << total_days
        << endl;
   return 0;
   }
```

Program yılbaşından itibaren, kullanıcı tarafından belirlenen bir tarihe kadar geçen günlerin sayısını hesaplar. (Dikkat: Artık yıllar hesaba katılmaz.) Örnek bir etkileşim şöyle olabilir:

```
Enter month (1 to 12): 3
Enter day (1 to 31): 11
Total days from start of year is: 70
```

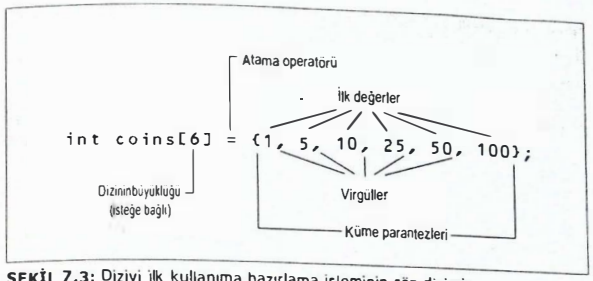
Program ay ve gün değerlerini alır almaz, ilk iş olarak gün değerini **total_days** değişkenine atar. Sonra bir döngü boyunca ilerler ve bu döngü içinde **days_per_month** dizisindeki değerleri toplayarak **total_days**'e değer olarak atar. Bu şekilde toplanacak değerlerin sayısı, ayların sayısından bir eksiktir. Örneğin, kullanıcı eğer 5. ayı girmişse, dizinin ilk dört elemanının değerleri (31, 28, 31 ve 30) toplama eklenecektir.

days_per_month'a başlangıçta atanan değerler köşeli parantezle çevrilidir ve virgülle ayrılmıştır. Bunlar, dizi deyimine eşittir işareti ile bağlanırlar. Şekil 7.3 ilgili söz dizimini gösterir.

Aslında, dizinin tüm elemanlarına ilk değer atadığımız durumlarda dizinin büyüklüğünü kullanmak zorunda değiliz, çünkü derleyici, başlangıç değerlerini sayarak dizinin büyüklüğünü tespit edebilir.

```
int days_per_month[]={ 31, 28, 31, 30, 31, 30,
                       31, 31, 30, 31, 30, 31 };
```

Peki, açıkça bir dizi büyüklüğü belirtirseniz ve bu, ilk değerlerin sayısı ile uyuşmazsa ne olur? Eğer ilk değerler daha azsa, eksik elemanlara 0 değeri atanır. Eğer ilk değerler çok daha fazla ise, hata mesajı verilir.



ŞEKİL 7.3: Diziyi ilk kullanıma hazırlama işleminin söz dizimi.

Çok Boyutlu Diziler

Şimdiye kadar tek boyutlu dizilere göz attık. Tek boyutlu dizilerde, dizi elemanlarının her birini tek bir değişken temsil eder. Fakat, diziler daha yüksek boyutta da olabilir. Aşağıdaki program SALEMON, birkaç bölgenin birkaç aylık satış rakamlarını saklamak amacıyla iki boyutlu bir dizi kullanır:

```
// salemon.cpp
// 2 boyutlu bir dizi kullanarak satis grafigini gosterir
#include <iostream>
#include <iomanip>                          //setprecision vs icin
using namespace std;

const int DISTRICTS = 4;                    //dizi boyutlari
const int MONTHS = 3;

int main()
   {
   int d, m;
   double sales[DISTRICTS][MONTHS];         //iki boyutlu dizi
                                            //tanimi
   cout << endl;
   for(d=0; d<DISTRICTS; d++)               //dizi degerlerini al
      for(m=0; m<MONTHS; m++)
         {
         cout << "Enter sales for district" << d+1;
         cout << ", month " << m+1 <<": ";
         cin >> sales[d][m];                //sayiyi diziye koy
         }
   cout << "\n\n";
   cout << "                  Month \n ";
   cout <<"            1       2       3 ";
   for(d=0; d<DISTRICTS; d++)
      {
      cout << "\nDistrict " << d+1;
      for(m=0; m<MONTHS; m++)               //dizinin degerlerini ekranda goster
         cout << setiosflags(ios::fixed)          //ustel degil
              << setiosflags(ios::showpoint)      //her zaman nokta kullan
              << setprecision(2)            //noktanın sagindaki basamaklar
              << setw(10)                   //alan genisligi
              << sales [d][m];              //diziden sayiyi al
```

```
    }     //for(d)'nin sonu
cout << endl;
return 0;
}     //main'in sonu
```

Bu program satış rakamlarını kullanıcıdan alır ve bunları bir tablo şeklinde ekranda gösterir:

```
Enter sales for district 1, month 1: 3964.23
Enter sales for district 1, month 2: 4135.87
Enter sales for district 1, month 3: 4397.98
Enter sales for district 2, month 1: 867.75
Enter sales for district 2, month 2: 923.59
Enter sales for district 2, month 3: 1037.01
Enter sales for district 3, month 1: 12.77
Enter sales for district 3, month 2: 378.32
Enter sales for district 3, month 3: 798.22
Enter sales for district 4, month 1: 2983.53
Enter sales for district 4, month 2: 3983.73
Enter sales for district 4, month 3: 9494.98
                    Month
             1         2         3
District 1 3964.23   4135.87   4397.98
District 2  867.75    923.59   1037.01
District 3   12.77    378.32    798.22
District 4 2983.53   3983.73   9494.98
```

Çok Boyutlu Dizileri Tanımlamak

Dizi, her biri köşeli parantez içine alınmış iki boyut belirleyicisi ile tanımlanır:

```
double sales[DISTRICTS][MONTHS];
```

İki boyutlu bir dizi olarak **sales**'ı bir satranç tahtası gibi düşünebiliriz. Bir başka ifadeyle, **sales** dizilerin dizisidir, diye de düşünebiliriz. **sales**, **DISTRICT** elemanlarının bir dizisidir; **DISTRICT** elemanlarının her biri ise **MONTHS** elemanlarının bir dizisidir. Şekil 7.4 bunun nasıl göründüğünü açıklıyor.

Elbette, ikiden daha fazla boyutlu diziler de olabilir. Üç boyutlu bir dizi, dizilerin dizilerinin dizisidir. Böyle bir diziye üç indeksle erişilir:

```
elem = dimen3[x][y][z];
```

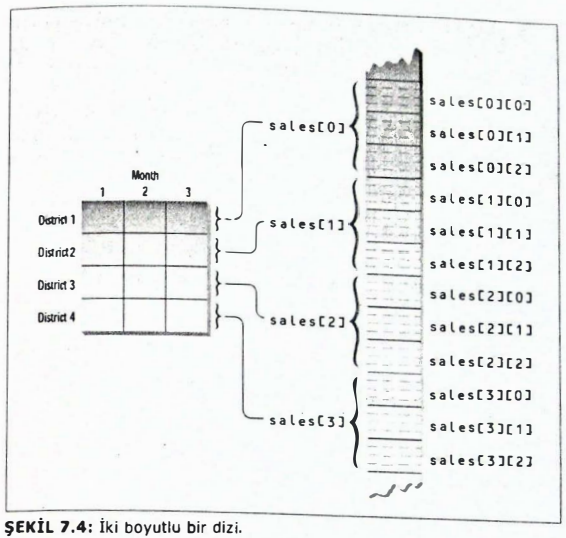
Üç boyutlu bir dizi, bir ve iki boyutlu dizilerin bütünüyle benzeridir.

Çok Boyutlu Dizilerin Elemanlarına Erişmek

İki boyutlu dizilerdeki dizi elemanları iki indeks gerektirir:

```
sales[d][m]
```

Dikkat ederseniz, her indeksin kendisine ait bir çift köşeli parantezi vardır. Virgül kullanılmaz. Şu şekilde yazmayın: **sales[d, m]**; Bu bazı dillerde doğrudur; fakat, C++'ta doğru bir ifade değildir.



ŞEKİL 7.4: İki boyutlu bir dizi.

Sayıların Biçimlendirilmesi

SALEMON programı, dolar değerlerinin bir tablosunu ekranda gösterir. Bu tür değerlerin düzgün olarak biçimlendirilmesi önemlidir. Bu nedenle, bunun C++'ta nasıl yapıldığını anlamak için biraz konu dışına çıkalım. Dolar değerleri söz konusu olduğunda, normal olarak ondalık kesri ifade eden noktanın sağında tam olarak iki basamak olmasını istersiniz; üstelik, bir sütundaki tüm sayıların noktalarının alt alta sıraya dizilmiş olmasını da istersiniz. Ayrıca, sayıları ekranda gösterirken sayıların sonuna 0 eklemek de hoş olur; 79.5 yerine 79.50'yi tercih edersiniz.

C++ I/O akışlarını (stream) tüm bu işlemleri yapmak için ikna etmek az bir iş gerektirir. Çıktının alan genişliğini ayarlamak için kullanılan **setw()** manipülatörünü görmüştünüz. Ondalık sayıları biçimlendirmek için birkaç ilave manipülatör daha lazımdır.

Aşağıdaki ifadeler **fpn** adında bir kayan noktalı sayıyı, 10 karakter genişliğinde ve ondalık noktadan sonra iki basamak olacak biçimde ekrana basar.

```
cout << setiosflags(ios::fixed)          //sabit (ustel degil)
     << setiosflags(ios::showpoint)      //noktayi her zaman goster
     << setprecision(2)                  //noktadan sonra iki ondalik basamak
     << setw(10)                         //alan genisligi 10
     << fpn;                             //son olarak, sayi
```

ios sınıfındaki **long int** tipindeki bir grup 1 bit *biçimlendirme işareti* (*formatting flags*) biçimlendirmenin nasıl yürütüleceğini belirler. Manipülatörleri çalıştırmak için şu aşamada **ios** sınıfının ne olduğunu veya bu sınıfla birlikte kullanılan doğru söz diziminin kullanım sebeplerini bilmemize gerek yoktur.

Biz `ios` işaretlerinden iki tanesi ile ilgileniyoruz: `fixed` ve `showpoint`. İşaretleri ayarlamak için, işaret ismini argüman olarak vererek `setiosflags` manipülatörünü kullanın. Argüman isminden önce sınıfın ismi olan `ios` ve "kapsam çözünürlük operatörü" (::) gelmelidir.

`cout` ifadesinin ilk iki satırı `ios` işaretlerini ayarlar. (Programın daha sonraki bir aşamasında eğer ayarlamayı geri almak, yani işaretleri temizlemek isterseniz `resetiosflags` manipülatörünü kullanabilirsiniz.) `fixed` işareti sayıların üstel biçimde, örneğin 3.45e3 biçiminde, basılmasını önler. `showpoint` işareti, sayının ondalık kısmı olmasa bile her zaman ondalık kesri ifade eden nokta işaretinin sayıda yer alacağını belirtiyor: 123 yerine 123.00 vs.

Ondalık kesri ifade eden noktanın sağında iki basamak olmasını sağlamak için, `setprecision` manipülatörünü basamak sayısını argüman olarak vererek kullanın. `setw` manipülatörünü kullanarak alan genişliğini nasıl ayarladığımızı daha önce görmüştük. Bu manipülatörlerin tümü `cout`'a gönderildikten sonra sayının kendisini de gönderebilirsiniz; sayı ekranda istenilen biçimde görüntülenecektir.

Bölüm 12'de "Akışlar ve Dosyalar" başlığı allında `ios` biçimlendirme işaretlerinden daha uzun bahsedeceğiz.

Çok Boyutlu Dizileri İlk Kullanıma Hazırlamak

Tahmin edeceğiniz gibi, çok boyutlu diziler de ilk kullanıma hazırlanabilir. Bunun tek ön şartı, bir sürü parantez ve virgül yazmaya istekli olmanızdır. Şimdi, girdileri kullanıcıdan sorup almak yerine, ilk kullanıma hazırlanmış bir dizi kullanan `SALEMON` programının bir versiyonuna göz atalım:

```
// saleinit.cpp
// satis grafigini ekranda goruntuler, iki boyutlu diziyi ilk kullanima hazirlar
#include <iostream>
#include <iomanip>                          //setprecision vs. icin
using namespace std;
const int DISTRICTS = 4;                    //dizi boyutlari
const int MONTHS = 3;

int main()
   {
   int d, m;
                                    //dizi elemanlarina baslangic degerleri ver
   double sales[DISTRICTS][MONTHS]
         ={ {  1432.07,    234.50,     654.01 },
            {   322.00, 13838.32,   17589.88 },
            {  9328.34,   934.00,    4492.30 },
            { 12838.29,  2332.63,     32.93 }};
   cout << "\n\n";
   cout << "                    Month\n";
   cout << "            1       2         3";
   for(d=0; d<DISTRICTS; d++)
      {
      cout << "\nDistrict " << d+1;
      for(m=0; m<MONTHS; m++)
         cout << setw(10) << setiosflags(ios::fixed)
              << setiosflags(ios::showpoint) << setprecision(2)
              << sales[d][m];       //dizinin elemanina ulas
      }
   cout << endl;
   return 0;
   }
```


İki boyutlu bir dizinin aslında dizilerin dizisi olduğunu aklınızdan çıkarmayın. Böyle bir diziyi ilk kullanıma hazırlama işleminin biçimi bu gerçeğe dayanır. Her alt dizinin başlangıç değerleri küme parantezi içine alınır ve virgülle birbirinden ayrılır:

```
{ 1432.07, 234.50, 654.01 }
```

Daha sonra her biri asıl dizinin birer elemanı olan bu alt dizilerin tümü, az önceki gibi, küme parantezi içine alınır ve virgülle birbirinden ayrılır. Program listesinde bunu görebilirsiniz.

Dizileri Fonksiyonlara Aktarmak

Diziler, fonksiyonlara argüman olarak kullanılabilir. Şimdi SALES programının bir versiyonuna göz atalım. `SALES`, bir dizi satış rakamını bir fonksiyona aktarır; bu fonksiyonun amacı bu verileri ekranda bir tablo şeklinde görüntülemektir. İşte `SALEFUNC`'ın program listesi:

```
// salefunc.cpp
// diziyi arguman olarak aktarir
#include <iostream>
#include <iomanip>
using namespace std;                                //setprecision vs. icin
const int DISTRICTS = 4;
const int MONTHS = 3;                               //dizi boyutlari
void display( double[DISTRICTS][MONTHS] );          //deklarasyon
//--------------------------------------------------------------
int main()
   {              //iki boyutlu diziyi ilk kullanima hazirla
   double sales[DISTRICTS][MONTHS]
      ={ { 1432.07,    234.50,     654.01 },
         {  322.00,  13838.32,   17589.88 },
         { 9328.34,    934.00,    4492.30 },
         {12838.29,   2332.63,      32.93 }};
   display(sales);      //fonksiyonu cagir; dizi arguman olarak aktarilsin
   cout << endl;
   return 0;
   }  // main sonu
//--------------------------------------------------------------
//display()
//arguman olarak aktarilan 2 boyutlu diziyi goruntuleyen fonksiyon
void display( double funsales[DISTRICTS][MONTHS] )
   {
   int d, m;

   cout << "\n\n";
   cout << "                  Month\n";
   cout << "            1       2         3";

   for(d=0; d<DISTRICTS; d++)
      {
      cout << "\nDistrict " << d+1;
      for(m=0; m<MONTHS; m++)
         cout << setiosflags(ios::fixed) << setw(10)
              << setiosflags(ios::showpoint) << setprecision(2)
              << funsales[d][m];    //dizi elemani
      }     //for(d) dongusunun sonu
   }     //display'in sonu
```

Dizi Argümanlar ile Fonksiyon Deklarasyonu

Bir fonksiyon deklarasyonunda dizi argümanlar, veri tipi ve dizinin boyutu ile gösterilirler. `display()` fonksiyonunun deklarasyonu şöyledir:

```
void display( float[DISTRICTS][MONTHS] );          //deklarasyon
```

Aslında burada tek bir parça gereksiz bilgi vardır. Aşağıdaki ifade de aynı işi görür:

```
void display( float[][MONTHS] );          //deklarasyon
```

Niçin fonksiyon ilk boyutun büyüklüğüne ihtiyaç duymaz? İki boyutlu bir dizinin, dizilerin dizisi olduğunu hatırlayın. Fonksiyon öncelikle argümanın, bölgeleri (district) tutan bir dizi olduğunu düşünür. Fonksiyonun kaç tane bölge olduğunu bilmesi gerekmez, fakat bölge elemanlarının her birinin hangi büyüklükte olduğunu bilmelidir. Böylece, belirli bir elemanın bellekte nerede olduğunu hesaplayabilir (her eleman için gerekli byte'ı indeks ile çarparak elemana erişir). Bu nedenle, her elemanın boyutunu, yani `MONTHS`'ı, bildirmemiz gerekir. Kaç tane eleman olduğunu ise – `DISTRICTS` ile ifade edilir – bildirmemize gerek yoktur.

Buradan bir çıkarım daha yapabiliriz: Bir boyutlu bir diziyi argüman olarak kullanan bir fonksiyon için deklarasyon yapıyorsak, dizi boyutunu kullanmak zorunda değiliz.

```
void somefunc(int elem []));     //deklarasyon
```

Dizi Argümanlar ile Fonksiyon Çağrıları

Fonksiyon çağrıldığında yalnızca dizinin ismi argüman olarak kullanılır.

```
display(sales);      //fonksiyon cagrisi
```

Bu isim (bu örnekte **sales**) aslında dizinin bellek adresini simgeler. Adresler konusunu "İşaretçiler" adlı Bölüm 10'a kadar ele almayacağız; burada gördüklerimiz, bu konuyla ilgili bir-iki ön bilgi niteliğindedir.

Bir dizi argümanın adresini kullanmak referans argümanı kullanmaya benzer. Şöyle ki, dizi elemanlarının değerleri fonksiyonun içine kopyalanmaz. (Bölüm 5'te, "Fonksiyonlar" başlığı altında ele alınan referans argümanlarına bakın.) Bunun yerine, fonksiyon, her ne kadar başka bir isim ile çağrılmış olsa da, orijinal dizi üzerinde işlem yapar. Diziler için bu sistem kullanılır, çünkü diziler çok büyük boyutta olabilir. Dizinin tümünü, kendisini çağıran fonksiyonların her birinin içine kopyalamak hem zaman kaybı hem de bellek israfıdır.

Herşeye rağmen bir adres referansla aynı anlama gelmez. Fonksiyon deklarasyonunda dizi ismi önünde ampersand (**&**) simgesi kullanılmaz. İşaretçilerden bahsedene kadar dizilerin sadece isimleri ile aktarıldığına ve fonksiyonun orijinal dizinin bir kopyasına değil, kendisine eriştiğine inanın.

Dizi Argümanlarla Fonksiyon Tanımı

Bir fonksiyon tanımında deklaratör şöyle görünür:

```
void display( double funsales[DISTRICTS][MONTHS] )
```


Veri tipi, bir isim ve dizi boyutlarının büyüklüğü dizi argümanında kullanılan unsurlardır. Fonksiyon tarafından kullanılan dizi ismi (bu örnekte **funsales**), dizinin tanımında kullanılan isimden (**sales**) farklı olabilir, fakat her ikisi de aynı diziyle ilgilidir. Dizi boyutlarının (bazı durumlarda ilk boyut hariç) tümü açıkça belirtilmelidir. Dizi elemanlarına doğru biçimde erişmek için fonksiyonun bunlara ihtiyacı vardır.

```
funsales[d][m]
```

Ancak, diğer açılardan fonksiyon, dizi elemanlarına sanki dizi fonksiyonun içinde tanımlanmış gibi ulaşabilir.

Yapılardan Oluşan Diziler

Diziler temel veri tiplerinin yanı sıra yapılar da içerebilir. "Yapılar" adlı Bölüm 4'teki **part** yapısını temel alan bir örnek verelim:

```
//partaray.cpp
//dizi elemanlari olarak yapi degiskenleri
#include <iostream>
using namespace std;
const int SIZE = 4;                    //dizi icindeki parca sayisi
////////////////////////////////////////////////////////////
struct part                            //yapiyi belirtelim
   {
   int modelnumber;                    //model numarasi
   int partnumber;                     //parca numarasi
   float cost;                         //parca maliyeti
   };
////////////////////////////////////////////////////////////
int main()
   {
   int n;
   part apart[SIZE];                   //yapilardan olusan bir dizi tanimla

   for(n=0; n<SIZE; n++)               //uyelerin her birinin degerlerini al
      {
      cout << endl;
      cout << "Enter model number: ";
      cin >> apart[n].modelnumber;          //model numarasini al
      cout << "Enter part number: ";
      cin >> apart[n].partnumber;           //parca numarasini al
      cout << "Enter cost:";
      cin >> apart[n].cost;                 //maliyeti al
      }
   cout << endl;
   for(n=0; n<SIZE; n++)               //uyelerin her birinin degerini goster
      {
      cout << "Model " << apart[n].modelnumber;
      cout << " Part " << apart[n].partnumber;
      cout << " Cost " << apart[n].cost << endl;
      }
   return 0;
   }
```

Kullanıcı bir ürünün model numarasını, yedek parça numarasını ve maliyetini programa girer. Program bu verileri bir yapı içine kaydeder. Ancak, bu yapı bir dizi yapı içinde yalnızca bir elemanı oluşturur. Program dört ayrı mal için gerekli veriyi sorar; bunları `apart` dizisinin dört elemanına yerleştirir. Program daha sonra bu bilgileri ekranda gösterir. Program için örnek girdi şöyle olabilir:

```
Enter model number: 44
Enter part number: 4954
Enter cost: 133.45

Enter model number: 44
Enter part number: 8431
Enter cost: 97.59

Enter model number: 77
Enter part number: 9343
Enter cost: 109.99

Enter model number: 77
Enter part number: 4297
Enter cost: 3456.55

Model 44 Part 4954 Cost 133.45
Model 44 Part 8431 Cost 97.59
Model 77 Part 9343 Cost 109.99
Model 77 Part 4297 Cost 3456.55
```

Programda yapılardan oluşan dizi şu ifadeyle tanımlanır:

```
part apart [SIZE];
```

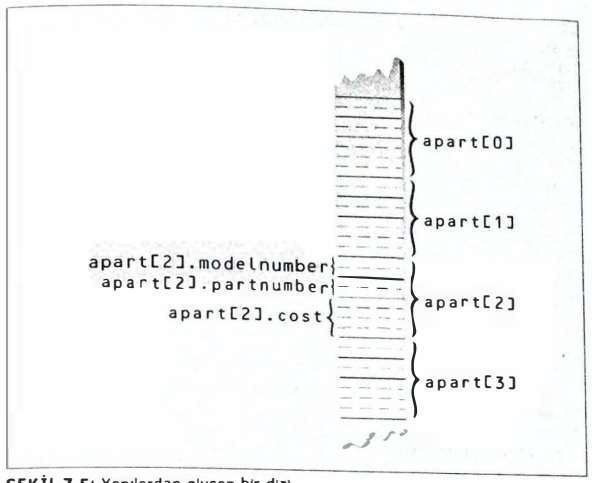
Bu ifade, temel veri tiplerinden oluşan dizilerle aynı sözdizimine sahiptir. Sadece tip ismi, yani **part**, bu dizinin daha karmaşık tipte bir dizi olduğunu gösterir.

Bir dizinin elemanı olan bir yapının bir üyesine erişmek yeni bir söz dizimi gerektirir. Örneğin,

```
apart[n].modelnumber
```

apart dizisinin **n**. elemanı olan yapının **modelnumber** üyesine erişmek için bu ifade kullanılır. Şekil 7.5 bunun nasıl gerçeklendiğini gösteriyor.

Yapılardan oluşan diziler, çok çeşitli durumlarda işe yarayan bir veri tipidir. Araba parçaları dizisini görmüştük; ayrıca personel verilerini (isim, yaş, maaş) içeren dizileri, şehirlerle ilgili coğrafi bilgileri (isim, nüfus, rakım) içeren dizileri ve diğer bir çok tipte verileri içeren dizileri oluşturup saklayabiliriz.



ŞEKİL 7.5: Yapılardan oluşan bir dizi.

Bir Sınıfın Üye Verisi Olarak Diziler

Diziler, sınıflar içinde bir veri parçası olarak da kullanılabilir. Şimdi, sıkça kullanılan bir bilgisayar veri yapısını, yani yığını, modelleyen bir örneğe bakalım.

Bir yığın (stack), kafeteryalardaki tabakların üst üste durduğu yaylı cihazlar gibi çalışır. En üste bir tabak koyduğunuzda yığın bir miktar aşağıya kayar; bir tabak aldığınızda yığın yukarıya yükselir. Yığına en son ilave edilen tabak, her zaman yığından ilk çıkarılacak olan tabaktır.

Yığınlar, bir çok modern bilgisayarda kullanılan mikroişlemcilerin mimarisindeki en temel unsurlardan biridir. Önceden bahsettiğimiz gibi, fonksiyonlar argümanlarını ve dönüş adreslerini yığında saklarlar. Bu tür bir yığın kısmen donanım bazında gerçeklenir ve en elverişli olarak assembly dili ile erişilir. Bununla birlikte, yığınlar tamamen yazılım bazında da gerçeklenebilir. Yazılım yığınları belirli programlama durumlarında, örneğin cebirel deyimlerin "parse" işleminde (analizinde), kullanışlı bir yedekleme aygıtı görevi görürler.

STAKARAY isimli örnek programımız basit bir yığın sınıfı oluşturur:

```
//stakaray.cpp
//bir sinif olarak yigin
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Stack
   {
   private:
      enum { MAX = 10 };            //(standart olmayan soz dizimi)
      int st[MAX];                  //yigin:tamsayi dizisi
      int top;                      //yiginin en ustunun numarasi
   public:
```

```
    Stack()                         //kurucu fonksiyon
       { top = 0; }
    void push(int var)              //sayiyi yigina koy
       { st[++top] = var; }
    int pop()                       //sayiyi yigindan al
       { return st[top--]; }
};
////////////////////////////////////////////////////////////////
int main()
  {
  Stack s1;

  s1.push(11);
  s1.push(22);
  cout << "1: " << s1.pop() << endl;      //22
  cout << "2: " << s1.pop() << endl;      //11
  s1.push(33);
  s1.push(44);
  s1.push(55);
  s1.push(66);
  cout << "3: " << s1.pop() << endl;      //66
  cout << "4: " << s1.pop() << endl;      //55
  cout << "5: " << s1.pop() << endl;      //44
  cout << "6: " << s1.pop() << endl;      //33
  return 0;
  }
```

`st` isimli dizi, yığının önemli bir üyesidir. `top` isimli `int` değişkeni, yığına en son yerleştirilen öğenin indeksini belirtir; bu öğenin konumu yığının *tepesidir* (*top*).

Yığın için kullanılan dizinin boyutu aşağıdaki ifadede `MAX` ile belirtilir:

```
enum { MAX = 10 };
```

`MAX`'in bu şekilde tanımlanması alışılmadık bir durumdur. İlgili verilerin birarada paketlenmesi (encapsulation) felsefine uyarak, bütünüyle bir sınıfın içinde kullanılacak sabitleri, mesela bu örnekteki `MAX`'i bu sınıfın içinde tanımlamak tercih edilen bir yaklaşımdır. Bu nedenle, bu amaç için global `const` değişkenlerinin kullanılması uygun olmaz. Standart C++, `MAX`'i aşağıdaki gibi, sınıfın içinde tanımlamanın mümkün olmasını zorunlu kılar:

```
static const int MAX = 10;
```

Bu ifade, `MAX`'in bir sabit olduğu ve bu sınıf içindeki tüm nesnelere uygulanabileceği anlamına gelir. Ne yazık ki, bazı derleyiciler, Microsoft Visual C++'ın şimdiki sürümü de dahil olmak üzere, bu yeni kabul gören yaklaşıma imkan vermezler.

Çare olarak bu tür sabitleri bir numaralandırma (Bölüm 4'te bahsedilmişti) olarak tanımlayabiliriz. Numaralandırmaya bir isim vermemize gerek yoktur; üstelik, sadece bir tane enümeratöre ihtiyacımız vardır:

```
enum { MAX = 10 };
```

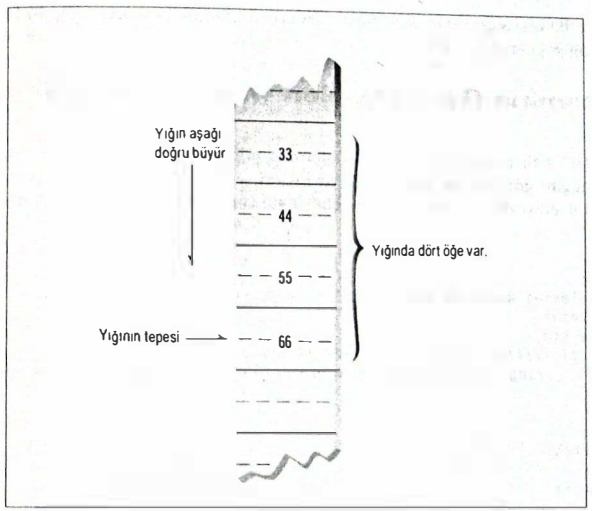
Bu ifade `MAX`'i, değeri 10 olan bir tamsayı olarak tanımlar. Üstelik, tanımın tamamı sınıfın içinde yer alır. Bu yaklaşım çalışır ama, kullanışlı değildir. Eğer derleyiciniz `static const` yaklaşımını destekliyorsa, sabitleri sınıf içinde tanımlamaktansa onu kullanmalısınız.

Şekil 7.6 bir yığını gösterir. Şekilde bellek aşağıya doğru büyüdüğü için yığının en üstü şeklin en altında kalır. Yığına bir öğe eklendiği zaman, `top`'un içindeki indeks yığının yeni elemanına işaret edecek biçimde artırılır. Yığından bir öğe alındığında ise, `top`'un içindeki indeks azaltılır. (Bir öğe yığından alındığında bellekte kalan eski değeri silmemize gerek yoktur; bu değer zaten önemsiz hale gelir.)

Yığına bir öğe eklemek (*push*) için `push()` üye fonksiyonunu çağırırsınız. Yığına eklenecek değer fonksiyonun argümanıdır. Yığının en üstünden bir öğe almak (*pop*) için ise `pop()` üye fonksiyonunu çağırırsınız. `pop()`, yığından alınacak öğenin değerini döndürür.

`STAKARAY` programındaki `main()` programı, `stack` sınıfına ait `s1` isimli bir nesne oluşturarak `stack` sınıfı ile ilgili uygulamalara örnek verir. Program yığının en üstüne iki öğe yerleştirir, bunları yığından alır ve bu öğelerin değerlerini ekranda gösterir. Sonra dört öğeyi daha yığının en üstüne koyar, yığından alır ve ekranda gösterir. Programın çıktısı şöyledir:

```
1: 22
2: 11
3: 66
4: 55
5: 44
6: 33
```



ŞEKİL 7.6: Bir yığın (stack).

Gördüğünüz gibi, öğeler yığından ters sırada alınır; yığına en son konulan ilk önce çıkarılır.

Artırım ve eksiltme operatörlerinde, güçlükle fark edilen önek ve sonek kullanımına dikkat edin.

`push()` üye fonksiyonundaki şu ifade önce `top`'ı bir artırır:

```
st [++top]== var;
```

Böylece **top**, mevcut olan bir sonraki dizi elemanını – sonuncu elemandan bir sonra geleni – göstermiş olur. Sonra bu elemana **var** değerini atar. Böylece **var**, yığının en üstündeki değer olur. Aşağıdaki ifade önce, yığının en üstünde bulduğu değeri döndürür:

```
return st [top--];
```

Sonra, **top**'ı bir azaltır. Böylece **top**, bir önceki elemanı göstermiş olur.

stack sınıfı, nesne yönelimli programlamanın önemli bir özelliğine örnek oluşturur: Sınıf kullanarak bir *konteyner* veya veri yedekleme mekanizması gerçekleştirilebilir. Bölüm 15'te bir yığının birkaç çeşit veri saklama yollarından sadece biri olduğunu göreceğiz. Ayrıca sıralar (queue), kümeler, bağlı listeler (linked list) gibi unsurlar da mevcuttur. Programın spesifik gereksinimlerini karşılayan bir veri yedekleme mekanizması seçilir. Verilerin yedeklenmesini sağlamak için önceden mevcut bir sınıf kullanmak, programcının veri yedekleme mekanizmasının ayrıntılarının kopyasını çıkarmakla zaman kaybetmeyeceği anlamına gelir.

Nense Dizileri

Bir nesnenin bir dizi içerebileceğini görmüştük. Bu durumu tersine çevirebilir ve bir nesneler dizisi oluşturabiliriz. İki olaya göz atacağız: İngiliz sistemine göre ölçülen mesafelerin bir dizisi ve bir takım iskambil kağıdının bir dizisi.

İngiliz Sistemine Göre Ölçülen Mesafelerin Bir Dizisi

"Nesneler ve Sınıflar" adlı Bölüm 6'da, İngiliz sistemine göre uzaklığı temsil eden **Distance** sınıfının birkaç örneğini gördük. Bu sınıf, ayak ve inçleri, yeni bir veri tipini simgeleyen bir nesnenin içine dahil ediyordu. Sıradaki örnek olan **ENGLARAY**, bu tür nesnelerin bir dizisini gösteriyor.

```
// englaray.cpp
// Ingiliz olculerini kullanan nesneler
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance    //Ingiliz sistemine gore uzakliklari simgeleyen Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      void getdist()                        //kullanicidan uzakligi al
         {
         cout << "\n     Enter feet: "; cin >> feet;
         cout << "     Enter inches: "; cin >> inches;
         }
      void showdist() const                 //uzakligi ekranda goster
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
```



```
{
Distance dist[100];                     //uzakliklar dizisi
int n=0;                                //girilenleri say
char ans;                               //kullanicinin cevabi('y' veya 'n')
cout << endl;

do {                                    //kullanicidan uzakliklari al
   cout << "Enter distance number " << n+1;
   dist[n++].getdist();                 //uzakligi dizide sakla
   cout << "Enter another (y/n)?: ";
   cin >> ans;
   } while( ans != 'n' );               //kullanici 'n' girerse programdan cik

for(int j=0; j<n; j++)                  //tum uzakliklari goruntule
   {
   cout << "\nDistance number " << j+1 << " is ";
   dist[j].showdist();
   }
cout << endl;
return 0;
}
```

Bu programda kullanıcı istediği sayıda uzaklık değeri girer. Kullanıcı uzaklıkların her birini girdikten sonra program yeni bir tane daha girmek isteyip istemediğini sorar. Eğer kullanıcı istemezse, program sona erer ve o ana dek girilen tüm uzaklıkları ekranda gösterir. Kullanıcı üç uzaklık girdiğinde programın etkileşimine bir örnek verelim:

```
Enter distance number 1
       Enter feet: 5
       Enter inches: 4
Enter another (y/n)? y
Enter distance number 2
       Enter feet: 6
       Enter inches: 2.5
Enter another (y/n)? y
Enter distance number 3
       Enter feet: 5
       Enter inches: 10.75
Enter another (y/n)? n

Distance number 1 is 5'-4"
Distance number 2 is 6'-2.5"
Distance number 3 is 5'-10.75"
```

Elbette, önceden girilmiş uzaklıkları sadece ekrana basmak yerine program bunların ortalamasını alabilirdi, bunları diskete kaydedebilirdi veya bunlar üzerinde başka türlü işlemler yapabilirdi.

Dizi Sınırları

Bu program kullanıcıdan girdi almak için bir **do** döngüsü kullanır. Bu şekilde kullanıcı, **part** tipinde yapılar için, en fazla dizinin büyüklüğü olan **MAX**'a kadar olmak şartıyla istediği çoklukta veri girebilir. (Programda **MAX**, 100 olarak belirlenir.)

Gerçi birinin bu kadar sabrı olabileceğini hayal etmek zor ama, eğer kullanıcı 100'den fazla uzaklık değeri girdiyse, ne olur? Cevap: Ne olacağı tam olarak kestirilemeyen ama kötü bir şey

olduğu neredeyse kesin olan bir durumla karşılaşılır. C++ dizilerinde sınır kontrolü yoktur. Eğer program dizi sınırının ötesine bir şey eklemeye kalkışırsa ne derleyici ne de çalışma zamanı sistemi buna itiraz etmeyecektir. Ancak, bu hain veri muhtemelen bir başka verinin ya da program kodunun üzerine yazılacaktır. Bu, acayip etkiler doğurabilir veya sistemi tamamen çökertebilir.

İşin özü, dizinin sınır kontrolüyle uğraşmak programcıya kalmış bir iştir. Eğer kullanıcının dizinin saklayabileceğinden çok daha fazla veri girme ihtimali varsa, dizi biraz daha büyütülmelidir veya kullanıcıyı uyarmanın bir yolu geliştirilmelidir. Örneğin, **ENGLARAY**'daki `do` döngüsünün başına aşağıdaki kodu ekleyebilirsiniz:

```
if(n >= MAX)
{
cout << "\nThe array is full!!!";
break;
}
```

Bu kod döngüden çıkış (`break`) sağlar ve dizinin taşmasını önler.

Bir Dizi İçindeki Nesnelere Erişmek

Bu programdaki `Distance` sınıfının deklarasyonu önceki programlarda kullanılanlarla aynıdır. Bunun yanı sıra, `main()` program içinde bu nesnelerden oluşan bir dizi tanımlıyoruz:

```
Distance dist [MAX];
```

Burada `dist` dizisinin veri tipi `Distance` ve dizi `MAX` sayıda elemana sahiptir. Şekil 7.7 bunun nasıl göründüğünü açıklıyor.

Bir dizinin elemanı olan bir sınıf üye fonksiyonu, **PARTARAY** örneğinde olduğu gibi, bir dizinin elemanı olan bir yapının üyesi ile aynı şekilde erişilir. `dist` dizisinin `j`'nci elemanının `showdist()` isimli üye fonksiyonu aşağıdaki gibi çağrılır:

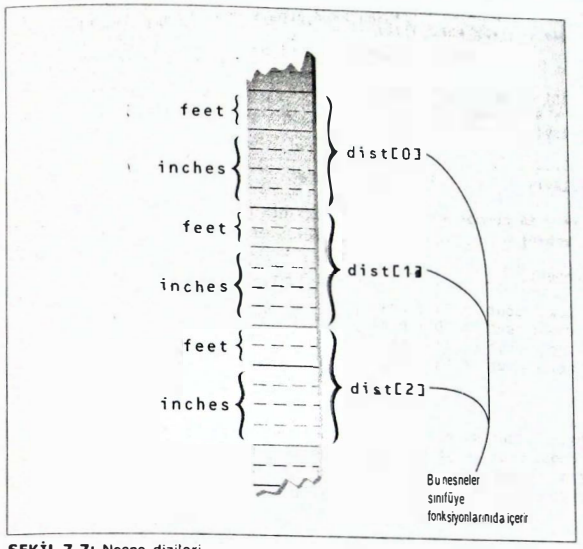
```
dist[j].showdist();
```

Görüldüğü gibi, kendisi bir dizi elemanı olan bir nesnenin bir üye fonksiyonu nokta operatörü kullanılarak erişilir: Dizi ismi ve peşinden gelen köşeli parantez içindeki indeks, nokta operatörü kullanılarak, üye fonksiyonun ismi ve peşinden gelen parantezlerle birleştirilir. Bu, bir yapının (veya sınıfın) üye verilerine erişmeye benzer; tek farkla, veri ismi yerine bu kez, fonksiyon ismi ve parantezler kullanılır.

Dikkat ederseniz, diziye bir uzaklık yerleştirmek için `getdist()` üye fonksiyonunu çağırdığımızda, dizi indeksini (`n`'i) artırma fırsatını da yakalamış oluruz:

```
dist[n++].getdist();
```

Bu şekilde, kullanıcıdan alınan bir sonraki veri grubu, `dist` dizisinin bir sonraki elemanında yer alan yapının içine yerleştirilecektir. `n` değişkeninin gösterildiği gibi elle artırılması gereklidir, çünkü programda `for` döngüsü yerine `do` döngüsü kullanılır. `for` döngüsü içinde otomatik olarak artırılan döngü değişkeni, dizi indeksi olarak da görev yapabilir.



ŞEKİL 7.7: Nesne dizileri.

İskambil Kağıdı Dizileri

Nesne dizilerine işte bir başka, hatta biraz da uzun, örnek daha. Bölüm 6'daki **CARDOBJ** programını şüphesiz hatırlayacaksınız. `card` sınıfını o örnekten ödünç alacağız ve bu tür nesnelerin 52 tanesini bir dizi içinde bir araya gruplayarak bir iskambil destesini oluşturacağız. **CARDARAY** için program listesi şöyledir:

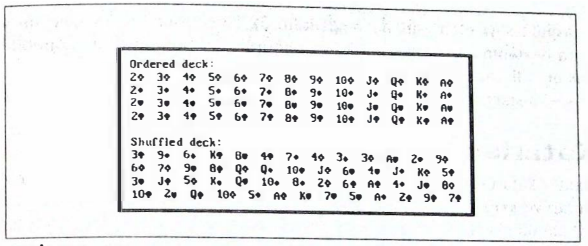
```
// cardaray.cpp
// kartlar birer nesne olarak simgelenir
#include <iostream>
#include <cstdlib>              //srand(), rand() icin
#include <ctime>                //srand() icin gerekli zaman icin
using namespace std;

enum Suit { clubs, diamonds, hearts, spades };
//2'den 10'a kadar ismi olmayan tamsayilar
const int jack = 11;
const int queen = 12;
const int king = 13;
const int ace = 14;
////////////////////////////////////////////////////////////////
class card
   {
   private:
      int number;               //2'den 10'a,vale,kiz,papaz,as
```

```
      Suit suit;                    //maca,karo,kupa,sinek
   public:
      card()                        //kurucu fonksiyon
         { }
      void set(int n, Suit s)       //karti ayarla
         { suit = s; number = n; }
      void display();               //karti goster
   };
//-------------------------------------------------------------
void card::display()                //karti goster
   {
   if( number >= 2 && number <= 10 )
      cout << number;
   else
      switch(number)
         {
         case jack:  cout << "J"; break;
         case queen: cout << "Q"; break;
         case king:  cout << "K"; break;
         case ace:   cout << "A"; break;
         }
   switch(suit)
      {
      case clubs:    cout << static_cast<char>(5); break;
      case diamonds: cout << static_cast<char>(4); break;
      case hearts:   cout << static_cast<char>(3); break;
      case spades:   cout << static_cast<char>(6); break;
      }
   }
////////////////////////////////////////////////////////////////
int main()
   {
   card deck [52];
   int j;

   cout << endl;
   for(j=0; j<52; j++)              //desteyi sirala
      {
      int num=(j % 13) + 2;         //2'den 14'e kadar, 4 kez doner
      Suit su = Suit(j / 13);       //0'dan 3'e kadar, 13 kez doner
      deck[j].set(num, su);         //karti ayarla
      }
   cout << "\nOrdered deck:\n ";
   for(j=0; j<52; j++)              //sirali desteyi ekranda goster
      {
      deck[j].display();
      cout << " ";
      if( !( (j+1) % 13) )          //her 13 karttan sonra yeni satira gec
         cout << endl;
      }
   srand( time(NULL) );             //zamani kullanarak rasgele bir sayi olustur
   for(j=0; j<52; j++)              //destedeki her kart icin
      {
      int k = rand() % 52;          //rasgele bir tane al
      card temp = deck[j];          //ve yerlerini degistir
      deck[j] = deck[k];
      deck[k] = temp;
      }
   cout << "\nShuffled deck:\n";
```



```
   for(j=0; j<52; j++)              //karistirilmis desteyi ekranda goster
      {
      deck[j].display();
      cout << ", ";
      if( !( (j+1) % 13) )          //her 13 karttan sonra yeni satira gec
         cout << endl;
      }
   return 0;
   }        //main'in sonu
```

Bir iskambil destesi hazırladıktan sonra kartları karıştırma isteğine karşı durmak zordur. Programımız destedeki kartları ekranda gösterir, kartları karıştırır ve tekrar ekranda gösterir. Yerden kazanmak amacıyla maça, karo, kupa ve sinek için grafik karakterleri kullanıyoruz. Şekil 7.8 programın çıktısını gösterir. Bu program bünyesinde birkaç yeni fikir içerir; şimdi sırayla bunları inceleyelim.



ŞEKİL 7.8: CARDARAY programının çıktısı.

Grafik Karakterleri

32 numaralı ASCII kodunun aşağısında kalan aralıkta birkaç tane özel grafik karakteri mevcut. (ASCII kodlarının listesi için Ek A'da "ASCII Tablosu"na bakın.) `card`'ın `display()` üye fonksiyonunda, maça, karo, kupa ve sinek karakterlerine erişmek için sırasıyla 5, 4, 3 ve 6 kodlarını kullanılır. Aşağıdaki ifadede gösterildiği gibi, bu sayıları `char` tipine çevirmek, << operatörünün bunları sayı yerine karakter olarak basmasını sağlar.

```
static_cast<char>(5)
```

İskambil Kağıdı Destesi

Bir iskambil destesini meydana getiren bir yapı dizisi aşağıdaki ifade ile tanımlanır:

```
card deck[52];
```

Bu tanım, `card` tipinde 52 nesne içeren, `deck` isminde bir dizi oluşturur. Destedeki j'nci kartı ekranda göstermek için `display()` üye fonksiyonu çağrılır:

```
deck[j].display();
```

Rasgele Sayılar

Rasgele sayılar üretmek her zaman çok eğlenceli, hatta kimi zaman yararlı bir iştir. Bu programda rasgele sayılar, desteyi karıştırmak için kullanılır. Bir rasgele sayı elde etmek için iki adım önemlidir. Birincisi, rasgele sayı üreticisi *tohumlanmalıdır (seeded)*, yani buna bir ilk değer verilmelidir. Bunu gerçekleştirmek için `srand()` kütüphane fonksiyonunu çağrılır. Bu fonksiyon sistem zamanını tohum olarak kullanır; bu nedenle, iki başlık dosyasına gereksinim duyar: `CSTDLIB` ve `CTIME`.

Gerçekten bir rasgele sayı üretmek için `rand()` kütüphane fonksiyonu çağrılır. Bu fonksiyon, rasgele bir tamsayı döndürür. Sayıyı 0 ile 51 arasında elde etmek için, `rand()`'ın döndürdüğü sonuca, kalan operatörü 52 sayısı ile birlikte uygulanır.

```
int k = rand() % 52;
```

Sonuç olarak elde edilen rasgele sayı `k`, iki kartı değiştirmek için indeks olarak kullanılır. Programda `for` döngüsünden geçilir; 0-51 arasındaki sıralı kartları işaret eden indeksin gösterdiği kart, indeksi rasgele sayı olan kart ile değiştirilir. 52 kartın her biri rasgele bir kart ile değiştirildikten sonra destenin karıştırıldığına kanaat getirilir. Bu program bir iskambil oyununun temellerini oluşturabilir; ayrıntıları size bırakıyoruz.

Nesne dizileri C++'ta sıkça kullanılır. Konularda ilerledikçe diğer örnekleri de göreceğiz.

C Karakter Katarları

C++'ta iki tür karakter katarının yaygın olarak kullanıldığını bu bölümün başında vurgulamıştık: C karakter katarları ve `string` sınıfının nesneleri olan karakter katarları. Bu bölümde birinci türü ele alacağız. C karakter katarları bu bölümün konusuna uyar, çünkü C karakter katarları, `char` tipinde birer dizidir. Bu karakter katarlarına, *C karakter katarları* veya *C stilinde karakter katarları* denir, çünkü bunlar C dilinde (ve C++'ın ilk günlerinde) mevcut olan tek karakter katarı tipiydi. Bunlar ayrıca `char*` karakter katarları olarak da adlandırılabilir, çünkü bu karakter katarları `char` tipine işaret eden işaretçiler olarak da simgelenebilir. (Bölüm 10'da öğreneceğimiz gibi, * simgesi bir işaretçiyi belirtir.)

Daha sonra inceleyeceğimiz, `string` sınıfından oluşturulan karakter katarları bir çok durumda C karakter katarlarının yerini alıyor olsa da, C karakter katarları çok çeşitli nedenlerden ötürü önemlerini korurlar. Birincisi; C karakter katarları C kütüphane fonksiyonlarının çoğunda kullanılırlar. İkincisi; önümüzdeki yıllara da miras olarak kalacak bir kod olmaya devam edeceklerdir. Üçüncüsü; C++ öğrencileri için C karakter katarları çok daha ilkeldir, dolayısıyla temel seviyesinde anlaşılmaları çok daha kolaydır.

C Karakter Katarı Değişkenleri

Diğer veri tiplerinde olduğu gibi, karakter katarları da değişken veya sabit olabilirler. Daha karmaşık karakter katarları işlemlerini incelemeye başlamadan önce bu iki olguya bir göz atalım. Şimdi, tek bir karakter katarı değişkeni tanımlayan bir örnek verelim. (Burada *karakter katarı* teriminin C karakter katarı anlamına geldiğini varsayıyoruz.) Program, kullanıcının bir karakter katarı girmesini ister ve bu karakter katarını `string` değişkenine yerleştirir. Daha sonra karakter katarını ekranda gösterir. `STRINGIN` programının listesi şöyledir:

```
// stringin.cpp
// basit karakter katari degiskeni
```

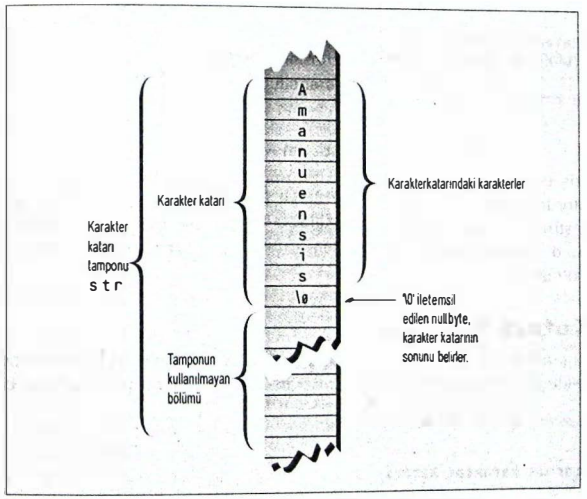


```
#include <iostream>
using namespace std;

int main()
   {
   const int MAX = 80;                  //karakter katarındaki en cok karakter sayısı
   char str[MAX];                       //karakter katarı degiskeni, str

   cout << "Enter a string:";
   cin >> str;                          //karakter katarını str'nin icine koy
                                        //karakter katarini str'den goruntule
   cout << "You entered: " << str << endl;
   return 0;
   }
```

`str` adlı karakter katarı değişkeninin tanımı `char` tipinde bir dizinin tanımına benzer (aslında dizi tanımının aynısıdır):

```
char str[MAX];
```

Klavyeden bir karakter katarı okumak ve bunu `str` adlı karakter katarı değişkeninin içine yerleştirmek için çıkarma operatörünü (>>) kullanıyoruz. Bu operatör karakter katarlarıyla nasıl ilgilenilmesi gerektiğini bilir; karakter katarlarının karakter dizileri olduğunu anlar. Bu programda kullanıcı eğer "Amanuensis" (bu kitabın el yazmalarını kopyalan kişinin ismi) karakter katarını girerse, `str` dizisi Şekil 7.9'dakine benzer biçimde görünür.



ŞEKİL 7.9: Karakter katarı değişkeninde saklı bir karakter katarı.

Her karakter 1 byte bellek kaplar. C karakter katarlarının önemli bir özelliği, bu karakter katarlarının 0 içeren bir byte ile sona ermesi gereğidir. Bu genellikle ASCII değeri 0 olan karakter anlamına gelen '\0' karakter sabiti ile simgelenir. Karakter katarını sona erdiren bu sıfır, *null karakteri* (*null character*) olarak adlandırılır. << operatörü karakter katarını ekranda gösterirken, null karakteriyle karşılaşana kadar karakterlerin her birini ekranda görüntüler.

Tampon Taşmalarını Önlemek

STRINGIN programı kullanıcıyı bir karakter katarı girmeye davet eder. Kullanıcı eğer karakter katarını tutmak için kullanılan diziden daha uzun bir karakter katarı girerse, ne olur? Daha önceden bahsettiğimiz gibi, programların dizinin dışına eleman eklemesini önleyen standart bir mekanizma C++'ta yoktur. Bu nedenle, fazlasıyla hevesli bir kullanıcı sonunda sistemi çökertebilir.

Bununla birlikte, >> operatörüne diziye yerleştireceği karakter sayısını kısıtlamasını bildirmek mümkündür. **SAFETYIN** programı bu yaklaşımı ortaya koyuyor:

```
// safetyin.cpp
// cin.width kullanarak tampon tasmalarini onluyor
#include <iostream>
#include <iomanip>              //setw icin
using namespace std;

int main()
   {
   const int MAX = 20;          //karakter katarindaki maksimum karakter sayisi
   char str[MAX];             //karakter katari degiskeni, str

   cout << "\nEnter a string: ";
   cin >> setw(MAX) >> str;       //MAX karakteri asmadan
                                  // karakter katarini str'nin icine yerlestir,
   cout << "You entered: " << str << endl;
   return 0;
   }
```

Bu program, giriş tamponunun alabileceği maksimum karakter sayısını açıkça belirtmek için **setw** manipülatörünü kullanır. Kullanıcı daha fazla karakter tuşlayabilir, fakat >> operatörü bunları diziye yerleştirmeyecektir. Aslında diziye, belirtilenden bir tane az karakter eklenir. Böylelikle dizi sonunda null karakteri için yer kalmış olur. Yani, **SAFETYIN**'deki diziye maksimum 19 karakter eklenir.

Karakter Katarı Sabitleri

Bir karakter katarını tanımlarken ona sabit bir ilk değer de atayabilirsiniz. Aşağıdaki örnek olan **STRINT** bunu gerçekleştirir (Programda, Shakespear'in sonelerinin birinden alınan bir satır kullanılır):

```
// strinit.cpp
// ilk deger atanmis karakter katari
#include <iostream>
using namespace std;

int main()
   {
```



```
   char str[] = "Farewell! thou art too dear for my possessing.";
   cout << str << endl;
   return 0;
   }
```

Buradaki karakter katarı, çift tırnak işareti ile sınırlandırılarak normal bir İngilizce ifade gibi yazılıyor. Bir karakter katarı, **char** tipinde bir dizi olduğu için bu, biraz şaşırtıcı gelebilir. Önceki örneklerde dizilere küme parantezleriyle çevrili, virgül ile ayrılmış bir seri değer atandığını görmüştünüz. Niçin str'a da aynı şekilde değer atanmıyor? Aslında karakter sabitlerinden oluşan şöyle bir seri de kullanabilirsiniz:

```
char str[] = {'F ','a ','r ','e ','w ','e ','l ','l ','!',' ','t ','h ',
```

ve diğerleri. Neyse ki, C++ (ve C) tasarımcıları bize acıdılar ve **STRINT**'te gösterilen kısa yolu sağladılar. Bu kısa yolun etkisi de aynıdır: Karakterler diziye birbiri arkasından yerleştirilir. Tüm C karakter katarlarında olduğu gibi en son karakter yine null karakteridir.

Metin İçine Gömülü Boşlukları Okumak

STRINGIN programını birden fazla kelime içeren karakter katarları ile çalıştırmayı denerseniz, tatsız bir sürpriz ile karşılaşabilirsiniz. İşte bir örnek:

```
Enter a string: Law is a bottomless pit.
You entered: Law
```

İfadenin kalan kısmı nereye gitti? (Yukarıdaki cümle, İskoçyalı yazar John Arbuthnot'tan [1667-1735] alınmıştır.) Buradan şunu anlıyoruz; çıkarma operatörü >>, boşluk karakterini karakter katarını bitiren karakter olarak ele alır. Yani, bu program yalnızca tek kelimeden oluşan karakter katarlarını okur, boşluğun ardından yazılan ne varsa hepsini kaldırıp atar.

Boşluk içeren metinleri okumak için başka bir fonksiyon kullanılır: **cin.get()**. Bu, **cin**'in bir nesnesi olduğu akış (stream) sınıfının **get()** isimli üye fonksiyonu anlamına gelir. Aşağıdaki **BLANKSIN** örneği bunun nasıl kullanıldığını gösterir:

```
// blanksin.cpp
// bosluk iceren karakter katarlarini okur
#include <iostream>
using namespace std;

int main()
   {
   const int MAX = 80          //karakter katarindaki maksimum karakter sayisi
   char str[MAX];              //karakter katari degiskeni, str

   cout << "\nEnter a string: ";
   cin.get(str, MAX);          //karakter katarini str'a yerlestir
   cout << "You entered:" << str << endl;
   return 0;
   }
```

cin::get() fonksiyonuna aktarılan ilk argüman, girilen karakter katarının yerleştirileceği dizinin adresidir. İkinci argüman dizinin maksimum uzunluğunu belirtir; böylece, tamponun taşması da otomatik olarak önlenmiş olur.

Bu fonksiyon kullanılarak, artık girilen karakter katarının tamamı saklanabilir.

```
Enter a string: Law is a bottomless pit.
You entered: Law is a bottomless pit.
```

`cin.get()`'i, `cin` ve çıkarma operatörü (`>>`) ile karıştırmak potansiyel bir probleme neden olur. Bu problemi çözmek için `ignore()` üye fonksiyonunun kullanımını "Akışlar ve Dosyalar" adlı Bölüm 12'de ele alacağız.

Birden Fazla Satır Okumak

İçinde boşluk içeren karakter katarlarını okuma problemini çözmüş olabiliriz; fakat, birkaç satır uzunluğundaki karakter katarlarını nasıl okuyacağız? Buradan, bu durumda işe yaraması için `cin::get()` fonksiyonunun bir üçüncü argüman alabileceği sonucuna varıyoruz. Bu argüman ile fonksiyona okumayı bitirmesini bildiren karakter belirtilmiş olur. Bu argümanın varsayılan değeri newline (yeni satır) karakteridir (`'\n'`), fakat argüman için başka bir karakter kullanarak da fonksiyonu çağırabilirsiniz. Bu durumda, argüman üzerinden aktarılan değer, varsayılan değerin üzerine yazılacaktır.

Sıradaki örneğimizde, `LINESIN` programı üçüncü argüman olarak dolar işaretini (`'$'`) kullanarak fonksiyonu çağırır:

```
// linesin.cpp
// birden fazla satir okur; '$' isareti girdinin sonunu belirtir
#include <iostream>
using namespace std;

const int MAX = 2000;       //karakter katarindaki maksimum karakter sayisi
char str[MAX];              //karakter katari degiskeni, str

int main()
   {
   cout << "\nEnter a string:\n";
   cin.get(str, MAX, '$');    //$ ile okuma sona ersin
   cout << "You entered:\n" << str << endl;
   return 0;
   }
```

Artık metinlerinizi istediğiniz kadar çok satır kullanarak girebilirsiniz. Fonksiyon, girdiyi sonlandıran karakter girilene kadar (veya dizinin sınırı aşılana kadar) karakterleri kabul etmeye devam edecektir. `'$'` karakterini tuşladıktan sonra da Enter tuşuna basmanız gerektiğini aklınızdan çıkarmayın. Thomas Crew'un [1595-1639] bir şiiri ile programın örnek çıktısı şöyledir:

```
Enter a string:
Ask me no more where Jove bestows
When June is past,the fading rose;
For in your beauty 's orient deep
These flowers,as in their causes,sleep.
$
You entered:
Ask me no more where Jove bestows
When June is past,the fading rose;
For in your beauty 's orient deep
These flowers,as in their causes,sleep.
```


Her satırın sonunda Enter'a basılır, fakat biz `'$'` karakterini girene kadar program girdileri kabul etmeye devam eder.

Bir Karakter Katarını Kopyalamanın Zor Yolu

Karakter katarlarının gerçek yapılarını daha iyi anlamak için onları karakter karakter ele almak en iyi yoldur. Aşağıdaki program bunu gerçekleştirir.

```
// strcopy1.cpp
// for dongusu kullanarak bir karakter katarini kopyalar
#include <iostream>
#include <cstring>                      //strlen() icin
using namespace std;

int main()
   {
   char str1[] = "Oh, Captain, my Captain!"     //ilk degeri atanmis karakter katari
      "our fearful trip is done";

   const int MAX = 80;                  //str2 tamponunun buyuklugu
   char str2[MAX];                      //bos karakter katari

   for(int j=0; j<strlen(str1); j++)    //strlen karakterlerini kopyala
      str2[j] = str1[j];                //str1'den str2'ye
   str2[j] = '\0';                      //en sona null karakterini ekle
   cout << str2 << endl;                //str2'yi goster
   return 0;
   }
```

Bu program bir karakter katarı sabiti ve bir karakter katarı değişkeni oluşturur: sırasıyla `str1` ve `strlen`. Daha sonra bir `for` döngüsü kullanarak karakter katarı sabitini karakter katarı değişkenine kopyalar. Kopyalama işlemi aşağıdaki ifade kullanılarak bir kerede bir karakter kopyalamak suretiyle gerçekleştirilir:

```
str2[j] = str1[j];
```

Hatırlarsanız, derleyici iki komşu karakter katarı sabitini birbirine ekleyerek tek bir karakter katarı oluşturur. Bu özellik bize tırnak içindeki ifadeyi iki satırda yazma imkanı verir.

Bu program ayrıca C karakter katarı kütüphane fonksiyonlarını da tanıtır. C++'ta hazır olarak kullanılabilecek karakter katarı operatörleri bulunmadığı için C karakter katarları çoğunlukla kütüphane fonksiyonları yardımıyla kullanılmalıdırlar. Neyse ki, bu tür fonksiyonlardan bolca vardır. Bu programda kullandığımız `strlen()`, bir C karakter katarının uzunluğunu (yani, içinde kaç karakter olduğunu) bulur. Bu uzunluk `for` döngüsünde limit olarak kullanılır. Böylece, doğru sayıda karakter kopyalanır. Karakter katarı fonksiyonları kullanıldığında `CSTRING` (veya `STRING.H`) başlık dosyası da (`#include` anahtar kelimesi ile birlikte) programa dahil edilmelidir.

Karakter katarının kopyalanan versiyonu null karakteri ile sona ermelidir. Ancak, karakter katarı uzunluğunu hesaplayan `strlen()` fonksiyonu null karakterini dahil etmez. İlave bir karakter daha ekleyebiliriz ama, null karakterini açıkça eklemek daha güvencelidir. Bu, şu satırla gerçekleştirilir:

```
str2[j] = '\0';
```

Eğer bu karakteri karakter katarının sonuna eklemezseniz programın ekranda gösterdiği karakter katarının sonunda her türlü acayip karakter'in karakter katarına dahil olmuş olduğunu fark edersiniz. << operatörü karakterler ne olursa olsun karakterleri basmaya devam eder, ta ki şans eseri bir '\0' karakterine rastlayana kadar.

Bir Karakter Katarını Kopyalamanın Kolay Yolu

Bir karakter katarını kopyalamak için elbette bir for döngüsüne ihtiyacınız yoktur. Tahmin etmiş olabileceğiniz gibi, bir kütüphane fonksiyonu bunu sizin için gerçekleştirecektir. strcpy() fonksiyonunu kullanan STRCOPY2 programı önceki programın düzeltilmiş bir versiyonudur:

```
// strcopy2.cpp
// strcpy() fonksiyonunu kullanarak bir karakter katari kopyalar
#include <iostream>
#include <cstring>                  //strcpy() icin
using namespace std;

int main()
   {
   char str1[] = "Tiger,tiger,burning bright\n"
                 "In the forests of the night";
   const int MAX = 80;              //str2 tamponunun buyuklugu
   char str2[MAX];                  //bos karakter katari

   strcpy(str2, str1);              //str1'den str2'ye kopyala
   cout << str2 << endl;            //str2'yi goster
   return 0;
   }
```

Dikkat ederseniz bu fonksiyon önce hedef (üzerine kopya yapılacak karakter katarı) verilerek çağrılır:

```
strcpy(hedef, kaynak)
```

Sağdan sola sıralaması normal değer atama ifadelerinin formatını hatırlatır: Sağdaki değişken, soldaki değişkene kopyalanır.

Karakter Katarı Dizileri

Şayet dizilerin dizileri oluyorsa, o zaman karakter katarlarının dizileri de elbette olur. Bu aslında çok kullanışlı bir ifadedir. Haftanın günlerinin isimlerini bir diziye yerleştiren STRARAY programı bunun bir örneğini veriyor:

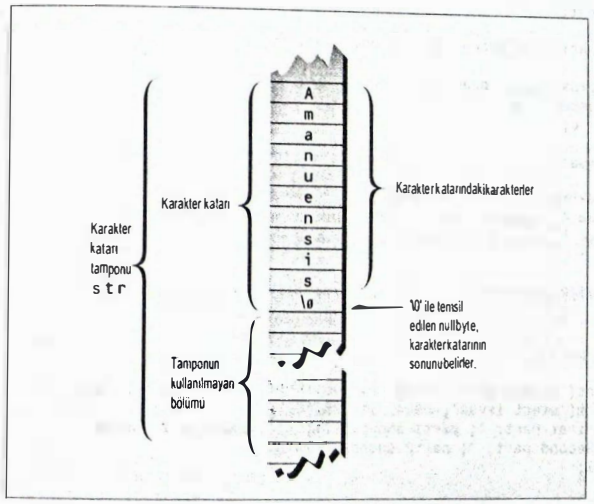
```
// straray.cpp
// karakter katari dizileri
#include <iostream>
using namespace std;

int main()
   {
   const int DAYS = 7;              //dizi icindeki karakter katari sayisi
   const int MAX = 10;              //karakter katarlarinin maksimum boyutu
                                    //karakter katari dizisi
   char star[DAYS][MAX] = {  "Sunday", "Monday", "Tuesday",
                             "Wednesday", "Thursday",
                             "Friday", "Saturday" };
   for(int j=0; j<DAYS; j++)        //karakter katarlarinin her birini goster
      cout << star[j] << endl;
   return 0;
   }
```


Bu program, dizideki karakter katarlarının her birini ekranda gösterir:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Karakter katarı bir dizi olduğu için bir karakter katarı dizisi olan star'ın aslında iki boyutlu bir dizi olduğu doğru olmalıdır. Bu dizinin ilk boyutu olan DAYS, dizide kaç tane karakter katarı olduğunu bildirir. İkinci boyut olan MAX, karakter katarlarının maksimum uzunluğunu belirtir ("Wednesday" için 9 karakter, bir de sonuncu null karakteri, toplam 10 karakter). Şekil 7.10 bunun nasıl göründüğünü açıklar.

Dikkat ederseniz, maksimum uzunluktan daha kısa olan karakter katarlarının sonunda kalan bazı byte'lar ziyan edilmiş olur. İşaretçilerden bahsederken bu verimsizlikten nasıl kurtulacağımızı da öğreneceğiz.



ŞEKİL 7.10: Karakter katarı dizileri.

Belirli bir karakter katarına erişmek için kullandığımız söz dizimi şaşırtıcı gelebilir:

```
star[j];
```

Eğer iki boyutlu bir dizi ile ilgileniyorsak, peki ikinci indeks nerede? İki boyutlu bir dizi, dizilerin dizisi olduğu için "dıştaki" dizinin elemanlarına – ki, bunların da her biri bir dizidir (bu örnekte bir karakter katarı) – tek tek erişebiliriz. Bunu gerçekleştirmek için ikinci bir indekse ihtiyacımız yoktur. Bu nedenle, `star[j]`, karakter katarı dizisi içindeki j'nci karakter katarıdır.

Sınıf Üyesi Olarak Karakter Katarları

Karakter katarları, sınıf üyesi olarak sık sık karşımıza çıkarlar. Bölüm 6'daki **OBJPART** programının bir versiyonu olan aşağıdaki örnek, otomobil parçalarının isimlerini saklamak için C karakter katarlarını kullanır:

```
// strpart.cpp
// yedek parca nesnesinde karakter katarlari
#include <iostream>
#include <cstring>                  //strcpy() icin
using namespace std;
////////////////////////////////////////////////////////////
class part
   {
   private:
      char partname[30];            //yedek parca ismi
      int partnumber;               //yedek parca seri numarasi
      double cost;                  //parcanin maliyeti
   public:
      void setpart(char pname [], int pn, double c)
         {
         strcpy(partname, pname);
         partnumber = pn;
         cost = c;
         }
      void showpart()               //verileri goster
         {
         cout << "\nName=" << partname;
         cout << ", number=" << partnumber;
         cout << ", cost=$" << cost;
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   part part1, part2;

   part1.setpart("handle bolt", 4473, 217.55);     //parcalari ayarla
   part2.setpart("start lever", 9924, 419.25);
   cout << "\nFirst part: "; part1.showpart();     //parcalari goster
   cout << "\nSecond part: "; part2.showpart();
   cout << endl;
   return 0;
   }
```


Bu program, `part` sınıfına ait iki nesne tanımlar ve `setpart()` üye fonksiyonunu kullanarak bu nesnelere değer atar. Daha sonra `showpart()` üye fonksiyonunu kullanarak bu nesneleri ekranda gösterir. Programın çıktısı şöyledir:

```
First part:
Name=handle bolt, number=4473, cost=$217.55
Second part:
Name=start lever, number=9924, cost=$419.25
```

Programın boyutunu küçük tutmak için model numarasını sınıf üyelerinden çıkardık.

`setpart()` üye fonksiyonu içerisinde, `pname` argümanından sınıfın üye verisi olan `partname`'e bir karakter katarı kopyalamak için `strcpy()` kütüphane fonksiyonunu kullanıyoruz. Yani, bir değer atama ifadesinin basit değişkenler üzerinde gerçekleştirdiği işlemin aynısını bu fonksiyon karakter katarları üzerinde gerçekleştirir. (Benzer bir fonksiyon olan `strncpy()`, kopyalayacağı maksimum karakter sayısını üçüncü bir argüman olarak alır. Bu, dizinin sınırlarından taşmayı önlemeye yardımcı olabilir.)

Burada karakter katarıyla ilgili gördüğümüz kütüphane fonksiyonlarının dışında, bir karakter katarı diğerine ekleyen, karakter katarlarını karşılaştıran, karakter katarları içinde belirli karakterleri arayan ve diğer işleri yerine getiren başka kütüphane fonksiyonları da vardır. Bu fonksiyonların tanımları derleyicinizin dokümantasyonunda bulunabilir.

Kullanıcı Tarafından Tanımlanan Bir Karakter Katarı Tipi

C karakter katarları C++'ta normal olarak kullanıldıklarında bazı problemler ortaya çıkar. Mesela, bir karakter katarını diğerine eşitlemek için son derece mantıklı olan bu deyimi kullanamazsınız:

```
strDest = strSrc;
```

(BASIC gibi bazı dillerde bu deyim tamamen doğrudur.) Bir sonraki bölümde inceleyeceğimiz Standart C++ `string` sınıfı bu problemin üstesinden gelecektir, fakat şimdilik bu problemi kendimiz çözmemiz için nesne yönelimli teknolojiyi kullanıp kullanamayacağımıza bir bakalım. Kendi karakter katarı sınıfımızı oluşturmak, karakter katarlarını bir sınıfın nesneleri olarak simgeleme konusunda bize fikir verecektir. Bu, ayrıca Standart C++ `string` sınıfının işleyişini de aydınlatacaktır.

Bir C++ sınıfı kullanarak kendi karakter katarı tipimizi tanımlarsak, bu kez değer atama ifadelerini kullanabiliriz. (Karakter katarlarını birbirinin peşine ekleme gibi diğer bir çok C karakter katarı işlemleri de bu yolla kolaylaşacaktır, fakat bunun nasıl yapıldığını görmek için "Operatörlerin Aşırı Yüklenmesi" adlı Bölüm 8'e kadar beklemek zorundayız.)

STROBJ programı `String` adında bir sınıf oluşturur. (Bu ev yapımı `String`'i Standart C++'ta hazır olarak bulunan ve küçük 's' ile başlayan `string` ile karıştırmayın.) İşte programın listesi:

```
// strobj.cpp
// sinif olarak bir karakter katari
#include <iostream>
#include <cstring>              //strcpy(), strcat() icin
using namespace std;
```

```
//////////////////////////////////////////////////////////
class String
   {
   private:                          //String'lerin maksimum buyuklugu
      enum { SZ = 80; };             //dizi
      char str[SZ];
   public:                           //kurucu fonksiyon, argumansiz
      String()
         { str[0] = '\0'; }          //kurucu fonksiyon, bir argumanli
      String( char s[] )
         { strcpy(str, s); }         // karakter katarini goster
      void display()
         { cout << str; }            //arguman karakter katarini
      void concat(String s2)         //bu karakter katarina ekle
         {
         if( strlen(str)+strlen(s2.str) < SZ )
            strcat(str, s2.str);
         else
            cout << "\nString too long";
         }
   };
//////////////////////////////////////////////////////////
int main()
   {
   String s1("Merry Christmas! ");          //2. Kurucu fonksiyonu kullanir
   String s2 = "Season 's Greetings!";      //2.nin alternatif sekli
   String s3;                               //1. Kurucu fonksiyonu kullanir

   cout << "\ns1="; s1.display();           //hepsini ekranda goster
   cout << "\ns2="; s2.display();
   cout << "\ns3="; s3.display();

   s3 = s1;                                 //atama
   cout << "\ns3="; s3.display();           //s3'u ekranda goster

   s3.concat(s2);                           //ekle
   cout << "\ns3="; s3.display();           //s3'u ekranda goster
   cout << endl;
   return 0;
   }
```

String sınıfı **char** tipinde bir dizi içerir. Yeni tanımlanmış sınıfımız, karakter katarının orijinal tanımıyla tamamen aynı gibi görünebilir: **char** tipinde bir dizi. Ancak diziyi bir sınıfın içinde paketleyerek bazı enteresan avantajlar sağlamış olduk. Bir nesneye, = operatörü kullanarak aynı sınıftan başka bir nesnenin değeri atanabildiği için bir **String** nesnesini diğerine eşitlemek amacıyla **main()**'de şöyle bir ifade kullanabiliriz:

```
s3 = s1;
```

String'lerle (**String** sınıfının nesneleriyle) iş yapmak için ayrıca kendi üye fonksiyonlarımızı da tanımlayabiliriz.

STROBJ programında tüm **String**'ler aynı uzunluğa sahiptir: **SZ** karakter uzunluğunda (**SZ**'ye 80 değeri verilmiştir). Programda iki kurucu fonksiyon vardır. Birinci kurucu fonksiyon, **str**'nin ilk karakterine null karakterini, ('\0'), atar; böylece, bu karakter katarının uzunluğu 0 olur. Bu fonksiyon şu ifade ile çağrılır:


```
String s3;
```

İkinci kurucu fonksiyon, **String** nesnesine "normal" bir karakter katarı (yani, bir C karakter katarı) sabiti atar. Fonksiyon, karakter katarı sabitini nesnenin veri üyesine kopyalamak için **strcpy()** kütüphane fonksiyonunu kullanır. Bu kurucu fonksiyon şu tür ifadelerle çağrılır:

```
String s1("Merry Christmas! ");
```

Bu fonksiyonu çağırmanın alternatif formatı şöyledir:

```
String s1 = "Merry Christmas! ";
```

Bu format bir argümanlı kurucu fonksiyonla çalışır. Hangi formatı kullanırsanız kullanın, bu kurucu fonksiyon etkileyici biçimde bir C karakter katarını bir **String**'e dönüştürür. Yani, normal bir karakter katarı sabitini **String** sınıfının bir nesnesine dönüştürür. **display()** adında bir üye fonksiyon ise **String**'i ekranda gösterir.

String sınıfımızın bir diğer üye fonksiyonu olan **concat()**, bir **String**'i diğerinin sonuna ekler. Orijinal **String**, **concat()**'ın üyesi olduğu nesnedir. Bu **String**'e argüman olarak aktarılan **String** eklenecektir. Yani, **main()**'deki şu ifade, **s2**'nin, mevcut olan **s3**'ün sonuna eklenmesini sağlar:

```
s3.concat(s2);
```

s2'ye başlangıçta "Season's Greetings!" değeri verildiği için ve **s3**'e atanan değer ise **s1**'in değeri, yani "Merry Christmas!" olduğu için, sonuçta **s3**'te elde edilen değer, "Merry Christmas! Season's Greetings!" olur.

concat() fonksiyonu ekleme işlemini gerçekleştirmek için **strcat()** C kütüphane fonksiyonunu kullanır. Bu kütüphane fonksiyonu ikinci argümanda belirtilen karakter katarını, birinci argümanda belirtilen karakter katarına ekler. Programın çıktısı şöyledir:

```
s1=Merry Christmas!
s2=Season's Greetings!
s3=    <==== burada henüz bir şey yok
s3=Merry Christmas! <==== s1'in degerine esitle
s3=Merry Christmas!Season's Greetings!   <==== s2 eklenmis
```

Eğer **concat()** fonksiyonuna verilen iki **String** birlikte maksimum **String** uzunluğunu geçiyorsa, bu durumda ekleme işlemi yapılır ve kullanıcıya bir mesaj gönderilir.

Şimdi basit bir karakter katarı sınıfını incelemiş olduk. Artık aynı yaklaşımın çok daha komplike versiyonlarını göreceğiz.

Standart C++ string Sınıfı

Standart C++, **string** adında yeni bir sınıf içerir. Bu sınıf, geleneksel C karakter katarının bir çok açıdan geliştirilmiş bir versiyonu. Bir defa, karakter katarı değişkenlerini saklamak amacıyla doğru boyutta bir dizi oluşturmak için artık kaygılanmanıza gerek yoktur. **string** sınıfı bellek yönetimiyle ilgili tüm sorumluluğu üstüne alır. Ayrıca, **string** sınıfı, aşırı yüklenmiş

operatörler kullanmanıza da imkan verir. Böylece, string nesnelerini + operatörünü kullanarak ekleyebilirsiniz.

```
s3 = s1 + s2
```

Ayrıca başka avantajları da vardır. Bu yeni sınıfın kullanımı, C karakter katarlarına nazaran daha hızlı ve güvenlidir. Pek çok durumda bu, tercih edilen bir yaklaşımdır. (Yine de, önceden de bahsettiğimiz gibi, hala C karakter katarlarının kullanılmasının zorunlu olduğu pek çok durum söz konusudur.) Bu bölümde **string** sınıfını ve bu sınıfın çeşitli üye fonksiyonlarını ve operatörlerini inceleyeceğiz.

string Nesnelerini Tanımlamak ve string Nesnelerine Değer Atamak

Bir **string** nesnesini birkaç değişik şekilde tanımlayabilirsiniz. Argümanı olmayan bir kurucu fonksiyon tanımlayabilirsiniz; böylece, boş bir karakter katarı oluşturursunuz. Argümanı bir C karakter katarı olan, yani çift tırnak ile çevrelenmiş karakterlerden oluşan, bir argümanlı bir kurucu fonksiyon da kullanabilirsiniz. Ev yapımı **String** sınıfımızda, **string** sınıfının nesneleri basit bir atama operatörü ile birbirlerine atanabilir. **SSTRASS** örneği bunun nasıl göründüğünü açıklar.

```
// sstrass.cpp
// string nesnelerini tanimlamak ve onlara deger atamak
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string s1("Man");                  //ilk kullanima hazirla
   string s2 = "Beast";               //ilk kullanima hazirla
   string s3;

   s3 = s1;                           //atama yap
   cout << "s3 = " << s3 << endl;

   s3 = "Neither " + s1 + " nor ";  //ekle
   s3 += s2;                          //ekle
   cout << "s3 = " << s3 << endl;

   s1.swap(s2);                     //s1 ve s2'yi degistir
   cout << s1 << " nor " << s2 << endl;
   return 0;
   }
```

Bu örnekte kodun ilk üç satırı **string** nesneleri tanımlamanın üç yolunu gösterir. İlk ikisi **string**'lere değer atar; üçüncüsü boş bir **string** oluşturur. Bir sonraki satır, = operatörü ile basit bir atama yapar.

string sınıfı birkaç sayıda aşırı yüklenmiş operatör kullanır. Bir sonraki bölüme kadar aşırı yüklenen operatörlerin sistem bazında işleyişleri hakkında bir şeyler öğrenmeyeceğiz; fakat, bu operatörlerin nasıl oluşturulduklarını bilmeden de operatörleri kullanabilirsiniz.

Aşırı yüklenmiş + operatörü bir karakter katarını diğerine ekler. Aşağıdaki ifade "Neither Man nor " karakter katarını **s3** değişkenine yerleştirir:

```
s3="Neither " + s1 + " nor ";
```

Mevcut bir karakter katarının peşine bir karakter katarı eklemek için += operatörünü de kullanabilirsiniz:

```
s3 += s2;
```

Bu ifade "Beast" değerini içeren **s2**'yi **s3**'ün sonuna ekler. Böylece, "Neither Man nor Beast" karakter katarı oluşturulur ve bu değer **s3**'e atanır.

Bu örnek ayrıca ilk **string** sınıfı üye fonksiyonumuzu da tanıtır: İki string nesnenin değerlerini değiş-tokuş eden **swap()** fonksiyonu. Bir nesne bu fonksiyona argüman olarak aktarılırken, fonksiyon diğer nesne tarafından çağrılır. Bu fonksiyonu **s1** ("Man") ve **s2**'ye ("Beast") uyguluyoruz, sonra **s1**'in artık "Beast" ve **s2**'nin "Man" değerlerini içerdiğini göstermek için karakter katarlarının değerlerini görüntülüyoruz.

SSTRASS'ın çıktısı şöyledir:

```
s3 = Man
s3 = Neither Man nor Beast
Beast nor Man
```

string Nesneleriyle Giriş/Çıkış

Giriş ve çıkışlar C karakter katarlarının giriş ve çıkışlarıyla aynı biçimde ele alınır. << ve >> operatörleri **string** nesneleri ele alabilmek için aşırı yüklenir; **getline()** fonksiyonu ise birden fazla boşluk ve satır içeren girdileri ele alır. **SSTRIO** örneği bunun nasıl göründüğünü açıklar.

```
// sstrio.cpp
// string sinifi giris/cikis
#include <iostream>
#include <string>                     //string sinifi icin
using namespace std;

int main()
   {                                  //string sinifinin nesneleri
   string full_name, nickname, address;
   string greeting("Hello, ");

   cout << "Enter your full name: ";
   getline(cin, full_name);           //metin icine gomulu bosluklari okur
   cout << "Your full name is: " << full_name << endl;

   cout << "Enter your nickname: ";
   cin >> nickname;                   //string nesnesine giris

   greeting += nickname;              //ismi greeting'e ekle
   cout << greeting << endl;          //cikti:"Hello, Jim"

   cout << "Enter your address on separate lines\n";
   cout << "Terminate with '$'\n";
```

```
    getline(cin, address, '$');      //birden fazla satir okur
    cout << "Your address is: " << address << endl;
    return 0;
    }
```

Program, **getline()**'ı kullanarak muhtemelen içinde boşluklar içeren kullanıcının ismini okur. Bu fonksiyon C karakter katarları ile kullanılan **get()** fonksiyonunun aynısıdır; fakat, bu bir üye fonksiyon değildir. Bunun yerine, bu fonksiyonun birinci argümanı, girdinin alınacağı akış nesnesidir (örnekte bu, **cin**); ikinci argüman ise, metnin yerleştirileceği **full_name** adında bir **string** nesnesidir. Bu değişken daha sonra **cout** ve << operatörü kullanılarak ekranda gösterilir.

Program, daha sonra **cin** ve >> operatörünü kullanarak kullanıcının tek kelimelik olduğu varsayılan takma ismini okur. En son olarak program, kullanıcının adresini okumak için **getline()**'ın üç argümanlı bir versiyonunu kullanır; zira, adres birkaç satır uzunluğunda olabilir. Üçüncü argüman, girdinin sonunu bildirmek amacıyla kullanılan karakteri belirtir. Programda '**$**' karakteri kullanılır. Bu, kullanıcının Enter tuşuna basmadan önce girmesi gereken en son karakterdir. Eğer **getline()**'a üçüncü bir argüman sağlanmazsa, ayırıcı karakterin '\n' olduğu farzedilir ki bu Enter tuşu ile simgelenir. Şimdi, **SSTRIO**'nun muhtemel çıktılarına bir göz atalım:

```
Enter your full name: F. Scott Fitzgerald
Your full name is: F. Scott Fitzgerald
Enter your nickname: Scotty
Hello, Scotty
Enter your address on separate lines:
Terminate with '$'
1922 Zelda Lane
East Egg, New York$
Your address is:
1922 Zelda Lane
East Egg, New York
```

string Nesnelerini Bulmak

string sınıfı, **string** nesneleri içinde belirli karakter katarlarını ve alt karakter katarlarını bulmak için çeşitli üye fonksiyonları içerir. **SSTRFIND** örneği bunlardan bazılarını gösterir:

```
// sstrfind.cpp
// string nesneleri icinde alt karakter katarlarini bulur
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string s1 =
      "In Xanadu did Kubla Kahn a stately pleasure dome decree";
   int n;

   n = s1.find("Kubla");
   cout << "Found Kubla at " << n << endl;

   n = s1.find_first_of("spde");
   cout << "First of spde at " << n << endl;
```



```
   n = s1.find_first_not_of("aeiouAEIOU ");
   cout << "First consonant at " << n << endl;
   return 0;
   }
```

find() fonksiyonu, kendisinin çağrıldığı karakter katarı içinde, kendisine argüman olarak aktarılan karakter katarını arar. Bu programda **find()** fonksiyonu, Samuel Taylor Coleridge'in *Kubla Khan* isimli şiirinin ilk dizesinin yer aldığı **s1** karakter katarı içinde "Kubla" sözcüğünü arar. Fonksiyon, bu sözcüğü 14. pozisyonda bulur. C karakter katarlarında olduğu gibi, en soldaki karakter 0 olarak numaralandırılır.

find_first_of() fonksiyonu, bir grup karakterden herhangi birini bir karakter katarı içinde arar; bunlardan hangisini ilk olarak bulursa, onun pozisyonunu döndürür. Örneğimizde fonksiyon, '**s**', '**p**', '**d**' veya '**e**' harflerinden herhangi birini **s1** içinde arar. Bu harflerden ilk olarak bulunan, Xanadu sözcüğünün içindeki ve 7. pozisyonda yer alan '**d**' harfidir.

Benzer bir fonksiyon olan **find_first_not_of()**,karakter katarı içinde, belirtilen gruptaki karakterlerden olmayan ilk karakteri arar. Burada, belirtilen grup küçük ve büyük harf olarak tüm sesli harfleri içerdiği için fonksiyon ilk sessiz harfi arar. Bu, **s1**'deki ikinci harfe karşılık gelir. **SSTRFIND**'ın çıktısı şöyledir:

```
Found Kubla at 14
First of spde at 7
First consonent at 1
```

Bu fonksiyonların bir çoğunun burada göstermediğimiz çeşitli varyasyonları vardır. Örneğin, **rfind()**,karakter katarını tersten tarar; **find_last_of()**, bir grup karakterden biri ile eşlenen karakter katarındaki en son karakteri bulur; **find_last_not_of()** ise, eşlenmeyen en son karakteri bulur. Bu fonksiyonların tümü aranılan bulunmadığı takdirde –1 döndürür.

string Nesneleri Üzerinde Değişiklik Yapmak

string nesneleri üzerinde değişiklik yapmanın çeşitli yolları mevcuttur. Bir sonraki örneğimiz **erase()**, **replace()** ve **insert()** üye fonksiyonlarını kullanım anında gösterir.

```
// sstrchng.cpp
// string nesnelerinin bolumlerini degistirmek
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string s1("Quick! Send for Count Graystone.");
   string s2("Lord");
   string s3("Don't");

   s1.erase(0, 7);              //"Quick! "i cikart
   s1.replace(9, 5, s2);        //"Count "u, "Lord " ile degistir
   s1.replace(0, 1, "s");       //'S'i, 's'ile degistir
   s1.insert(0, s3);            //En basa "Don't " ekle
   s1.erase(s1.size()-1, 1);    //'.' cikart
   s1.append(3, '!');           //sona "!!!" ekle

   int x = s1.find(' ');        //boslugu bul
```

```
while( x < s1.size() )          //bosluk oldugu surece donguye devam et
   {
   s1.replace(x, 1, "/");       //bolu isareti ile degistir
   x = s1.find(' ');            //bir sonraki boslugu bul
   }
cout << "s1: " << s1 << endl;
return 0;
}
```

erase() fonksiyonu, bir karakter katarından bir alt karakter katarını çıkartır. Fonksiyonun ilk argümanı, çıkartılacak olan alt karakter katarının ilk karakterinin, karakter katarı içindeki pozisyonudur. İkinci argüman ise alt karakter katarının uzunluğudur. Örnekte, fonksiyon "Quick " kelimesini karakter katarının başından çıkartır. **replace()** fonksiyonu, bir karakter katarının bir bölümüne başka bir karakter katarını yerleştirir. İlk argüman, yerleştirmenin başlaması gereken pozisyondur. İkinci argüman, orijinal karakter katarında değiştirilmesi gereken karakter sayısıdır. Üçüncü argüman ise, orijinal karakter katarına yerleştirilecek olan karakter katarıdır. Bu örnekte "Count"un yerine "Lord" yerleştirilir.

insert() fonksiyonu, ilk argüman ile belirtilen pozisyona ikinci argümanın belirttiği karakter katarını ekler. Burada, **s1**'in en başına "Don't" karakter katarı eklenir. **erase()** fonksiyonunun ikinci kullanımı, **string** nesnesindeki karakterlerin sayısını döndüren **size()** üye fonksiyonunu çağırır. **size()-1** deyiminin değeri en son karakterin, yani silinmesi gereken noktanın pozisyonudur. **append()** fonksiyonu ifadenin sonuna üç tane ünlem işareti ekler. Fonksiyonun bu versiyonunda ilk argüman, eklenecek karakter sayısı; ikinci argüman ise eklenecek karakterdir.

Programın sonunda, bir alt karakter katarının birden fazla örneğini başka bir karakter katarı ile değiştirmek için kullanabileceğiniz bir kod parçası görülüyor. Burada, bir **while** döngüsü içinde **find()** fonksiyonu kullanılarak boşluk karakteri (' ') aranır ve boşlukların her biri, **replace()** fonksiyonu kullanılarak bölü işareti ile değiştirilir.

"Quick! Send for Count Graystone." ifadesini içeren **s1** ile programa başlanır. Tüm bu değişiklerden sonra **SSTRCHNG**'in çıktısı şöyle olur:

```
s1: Don 't/send/for/Lord/Graystone!!!
```

string Nesnelerini Karşılaştırmak

string nesnelerini karşılaştırmak için aşırı yüklenmiş operatörler veya **compare()** fonksiyonunu kullanabilirsiniz. Bunlar, karakter katarlarının aynı olup olmadığını veya alfabetik açıdan birinin diğerinin önünde mi, arkasında mı olduğunu tespit ederler. **SSTRCOM** programı bazı ihtimalleri gösterir.

```
// sstrcom.cpp
// string nesnelerini karsilastirmak
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string aName = "George";
   string userName;
```



```
cout << "Enter your first name: ";
cin >> userName;
if(userName==aName)
   cout << "Greetings, George\n";          //== operatoru
else if(userName < aName)                  // < operatoru
   cout << "You come before George\n";
else
   cout << "You come after George\n";

                                           //compare() fonksiyonu
int n = userName.compare(0, 2, aName, 0, 2);
cout << "The first two letters of your name ";
if(n==0)
   cout << "match ";
else if(n < 0)
   cout << "come before ";
else
   cout << "come after ";
cout << aName.substr(0, 2) << endl;
return 0;
}
```

Programın ilk bölümünde, kullanıcının girdiği ismin, **George** ismi ile aynı olup olmadığını veya alfabetik olarak önce mi, sonra mı geldiğini belirlemek amacıyla == ve < operatörleri kullanılır. Programın ikinci bölümünde **compare()** fonksiyonu, **"George"** karakter katarının sadece ilk iki harfi ile kullanıcının girdiği ismin (**userName**) ilk iki harfini karşılaştırır. **compare()**'in bu versiyonunun argümanları şöyledir: **userName** içinde karşılaştırmanın başladığı pozisyon, karşılaştırılacak karakter sayısı, karşılaştırma için kullanılacak karakter katarı (**aName**), **aName** içinde karşılaştırılacak karakterlerin başlangıç pozisyonu ve **aName** içinde karşılaştırılacak karakter sayısı. **SSTRCOM** ile etkileşim şu şekilde olabilir:

```
Enter your first name: Alfred
You come before George
The first two letters of your name come before Ge
```

"George"un ilk iki harfi **substr()** üye fonksiyonu kullanılarak elde edilir. Bu fonksiyon kendisini çağıran karakter katarının bir alt karakter katarını döndürür. Fonksiyonun ilk argümanı, alt karakter katarının pozisyonu; ikinci argümanı ise karakter sayısıdır.

string Nesneleri İçindeki Karakterlere Erişmek

Bir **string** nesnesi içindeki karakterlere tek tek, birkaç yoldan erişebiliriz. Bir sonraki örneğimizde **at()** üye fonksiyonunu kullanarak bu tür bir erişimi göstereceğiz. Ayrıca aşırı yüklenmiş **[]** operatörünü de kullanabilirsiniz. Bu operatör, bir **string** nesnesinin bir dizi gibi görünmesini sağlar. Ancak, sınırlar dışındaki bir karaktere (örneğin, karakter katarının sonundan daha ilerideki bir karaktere) erişmeye kalktığınızda **[]** operatörü sizi uyarmaz. **[]** operatörü gerçek dizilerle de bu şekilde davranır ve çok daha kullanışlıdır. Ancak, fark etmesi güç program hatalarına yol açabilir. **at()** fonksiyonunu kullanmak daha emniyetlidir. Eğer karakter katarının sınırları dışında kalan bir indeks kullanırsanız, **at()** fonksiyonu programın durdurulmasını sağlar. (Aslında bunun da bir istisnası vardır; kural dışı durumları Bölüm 14'te "Şablonlar ve Kural Dışı Durumlar" başlığı altında ele alacağız.)

```
// sstrchar.cpp
// string nesneleri icindeki karakterlere erismek
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   char charray[80];
   string word;

   cout << "Enter a word: ";
   cin >> word;
   int wlen = word.length();          //string nesnesinin uzunlugu

   cout << "One character at a time: ";
   for(int j=0; j<wlen; j++)
      cout << word.at(j);             //sinirlarin disindaysa, kural disi bir durum
                                      //sinirlarin disindaysa, uyarmaz
//    cout << word[j];

   word.copy(charray, wlen, 0);       //string nesnesini diziye kopyala
   charray[wlen] = 0;                 // '\0' ile sonlandir
   cout << "\nArray contains: " << charray << endl;
   return 0;
   }
```

Bu programda, bir `string` nesnesi içindeki tüm karakterleri tek tek ekranda göstermek için `at()` fonksiyonu kullanılır. `at()`'e geçirilen argüman, karakter katarı içindeki karakterin konumudur.

Daha sonra bir `string` nesnesini `char` tipinde bir diziye kopyalamak için `copy()` üye fonksiyonunu nasıl kullanabileceği gösterilir. Böylece, `string` nesnesi, etkili bir biçimde C karakter katarına dönüştürülür. Kopyalama işleminin peşinden C karakter katarına dönüştürme işlemini tamamlamak için dizinin son karakterinden sonra null karakteri (`'\0'`) eklenmelidir. `string`'in `length()` üye fonksiyonu, `size()` ile aynı sayıyı döndürür.

`SSTRCHAR`'ın çıktısı şöyledir:

```
Enter a word: symbiosis
One character at a time: symbiosis
Array contains: symbiosis
```

(`string` nesnelerini C karakter katarlarına `c_str()` veya `data()` fonksiyonlarını kullanarak da dönüştürebilirsiniz. Ancak, bu fonksiyonları kullanabilmek için Bölüm 10'da inceleyeceğimiz işaretçileri bilmeniz gerekir.)

Diğer string Fonksiyonları

`size()` ve `length()`'in her ikisinin de, bir `string` nesnesi içinde halihazırda mevcut olan karakterlerin sayısını döndürdüğünü gördük. Bir `string` tarafından kullanılan bellek genellikle karakterler için gerekli olandan bir miktar daha fazladır. (Eğer ilk değer atanmamışsa, karakterler için 0 byte kullanılır. Fakat, buna rağmen gerçekte ayrılan bellek bundan biraz daha fazladır.) `capacity()` üye fonksiyonu kullanılan gerçek bellek miktarını döndürür. Bu limite ulaşana kadar, bir karakter katarına, kendisine tahsis edilmiş belleği genişletmesine gerek kalmadan karakterler ekleyebilirsiniz. `max_size()` üye fonksiyonu, bir `string` nesnesinin

muhtemel en büyük boyutunu döndürür. Bu miktar, sisteminizdeki `int` değişkenlerinin boyutunun 3 byte eksiğine karşılık gelir. 32 bit Windows sistemlerinde bu, 4,294,967,293 byte'tır; fakat, belleğinizin boyutu muhtemelen bu miktarı kısıtlayacaktır.

Ele aldığımız `string` üye fonksiyonlarının bir çoğu, argümanlarının sayısında ve tipinde sayısız çeşitlilikler gösterir. Ayrıntılı bilgi için derleyicinizin dokümantasyonuna bakmalısınız.

`string` nesnelerinin, C karakter katarlarından farklı olarak, null ile sonlanmadığına dikkat etmelisiniz. Bunun yerine, karakter katarının uzunluğu sınıfın bir üyesidir. Bu nedenle, eğer bir karakter katarı üzerinde ilerliyorsanız, karakter katarının sonuna ulaştığınızı size söyleyecek bir null değeriyle karşılaşacağınıza güvenmeyin.

`string` sınıfı aslında, tamamı `basic_string` şablon sınıfından türetilen pek çok muhtemel karakter katarı benzeri sınıftan sadece biridir. `string` sınıfı `char` tipini temel alır; fakat, yaygın olan bir varyantında `char` yerine `wchar_t` tipi kullanılır. Bu, `basic_string`'in, İngilizce'den daha fazla karakter içeren yabancı dillerde de kullanılmasına imkan verir. Derleyicinizin yardım dosyası, `basic_string` başlığı altında `string` üye fonksiyonlarını listeliyor olabilir.

Özet

Diziler, aynı tipte birkaç tane veri öğesi içerirler. Bu tip basit bir veri tipi, bir yapı veya bir sınıf olabilir. Bir dizinin içindeki unsurlara *elemanlar* denir. Elemanlara bir sayı ile erişilir; bu sayıya *indeks* denir. Dizi tanımlanırken elemanlara belirli ilk değerler atanabilir. Diziler çok boyutlu olabilir. İki boyutlu bir dizi, dizilerin dizisidir. Dizinin adresi bir fonksiyonda argüman olarak kullanılabilir; bu durumda, dizinin kendisi kopyalanmaz.

Diziler, sınıflarda veri üyesi olarak kullanılabilir. Bellekte bir dizinin dışına veri yerleştirmeyi önlemek için dikkat edilmelidir.

C karakter katarları `char` tipinde dizilerdir. Bir C karakter katarındaki en son karakter null karakteri (`'\0'`) olmalıdır. C karakter katarı sabitlerinin özel bir şekli vardır, böylece bu sabitler kolaylıkla yazılabilir: Metin çift tırnak ile çevrelenmiştir. C karakter katarlarını ele almak için çeşitli kütüphane fonksiyonları vardır. Bir C karakter katarları dizisi, `char` tipindeki dizilerin bir dizisidir. Bir C karakter katarı değişkeninin, içine yerleştirilecek herhangi bir metni tutabilecek kadar büyük bir dizi olduğu garanti edilmelidir. C karakter katarları, C stilindeki kütüphane fonksiyonlarında argüman olarak kullanılırlar. C karakter katarlarına daha eski programlarda rastlayabilirsiniz. Yeni programlarda C karakter katarlarının genel amaçlı kullanımları normal olarak tavsiye edilmemektedir.

karakter katarlarıyla ilgili tercih edilen yaklaşım, `string` sınıfının nesnelerini kullanmaktır. Bu karakter katarları çok sayıda operatör ve üye fonksiyonuyla ele alınabilir. `string` nesneleri sayesinde kullanıcının bellek yönetimini dert etmesi gerekmez.

Sorular

Soruların cevapları Ek G'de bulabilirsiniz.

1. Bir dizin elemanına aşağıdakilerden hangisi kullanılarak ulaşılır?
   a. ilk giren ilk çıkar yaklaşımı
   b. nokta operatörü
   c. üye ismi
   d. indeks numarası
2. Bir dizinin tüm elemanları _____ veri tipinden olmalıdır.

3. Her biri `double` tipinden 100 eleman içeren `doubleArray` adlı tek boyutlu bir dizi tanımlayan bir ifade yazın.
4. Toplam 10 elemandan oluşan bir dizinin elemanları _____'dan _____'a kadar numaralanırlar.
5. `doubleArray` dizisinin `j` elemanını alıp bunu ekleme operatörü ile `cout`'a yazan bir ifade oluşturun.
6. `doubleArray[7]` elemanı, dizinin kaçıncı elemanıdır?
   a. Altıncı
   b. Yedinci
   c. Sekizinci
   d. Bu bilgiler yeterli değil
7. `int` tipinden `coins` dizisini tanımlayan ve diziyi kullanıma hazırlayan bir ifade yazın.
8. Çok boyutlu bir diziye erişilirken, dizi endeksleri:
   a. virgülle ayrılır
   b. köşeli parantez içine yazılır ve virgülle ayrılır
   c. virgülle ayrılır ve köşeli parantez içine yazılır
   d. köşeli parantez içine yazılır
9. `twoD` adlı iki boyutlu bir dizinin içindeki 2 numaralı altdizinin dördüncü elemanına erişen bir ifade yazın.
10. Doğru/Yanlış: C++'ta dört boyutlu bir dizi olabilir.
11. `float` tipinde ve `flarr` adlı iki boyutlu bir dizi tanımlayan, ilk altdizine 52, 27, 83, ikinciye 94, 73, 49, üçüncüye de 3, 6, 1 ilk değerlerini atayan bir ifade yazın.
12. Kaynak dosyasında kullanılan bir dizi ismi o dizinin _____ temsil eder.
13. Bir dizi ismi bir fonksiyona aktarıldığında o fonksiyon
    a. çağıran programla eriştiği diziye aynen erişir.
    b. programın gönderdiği dizinin bir kopyasına erişir.
    c. dizi için, çağıran programın kullandığı ismin aynısını kullanır.
    d. dizi için, çağıran programın kullandığı isimden başka bir ismi kullanır.
14. Aşağıdaki ifade ne tanımlamaktadır?

```
employee emplist[1000];
```

15. `emplist` adındaki bir dizinin 17. Elemanı olan bir yapı değişkeninin salary adlı üyesine erişen bir ifade yazın.
16. Bir yığında, yığına ilk konulan veri elemanı
    a. herhangi bir indeks numarası almaz.
    b. indeks numarası olarak 0 alır.
    c. yığından çıkacak ilk veri elemanıdır.
    d. yığından çıkacak son veri elemanıdır.
17. `bird` tipinden 50 nesneyi tutacak, manybirds adlı bir dizi tanımlayan bir ifade yazın.
18. Doğru/Yanlış: 10 elemanlı bir dizinin 14. elemanına erişmek isterseniz derleyici hata verecektir.
19. `manybirds` dizininin 27. Elemanı olan bird sınıfından bir nesnenin `cheep()` üye fonksiyonunu çalıştıran bir ifade yazın.
20. C++'ta bir karakter katarı _____ tipinden bir _____ olarak tanımlanır.

21. Uzunluğu 20'ye kadar çıkabilen bir karakter katarını tutabilen `city` adlı bir karakter katarı değişkenini tanımlayan bir ifade yazın (biraz zor bir soru).
22. Değeri "C6H12O6-H2O" olan `dextrose` adlı bir karakter katarı sabitini tanımlayan bir ifade yazın.
23. Doğru/Yanlış: Çıkarma operatörü (>), boşlukla karşılaştığı zaman bir karakter katarını okumayı bırakır.
24. Birden fazla satırdan oluşan bir metin, aşağıdakilerden hangisini kullanarak okunur?
    a. normal `cout <<` kombinasyonu.
    b. bir argümanlı `cin.get()` fonksiyonu.
    c. iki argümanlı `cin.get()` fonksiyonu.
    d. üç argümanlı `cin.get()` fonksiyonu.
25. `name` adlı karakter katarını `blank` adlı karakter katarına kopyalamak için bir karakter katarı kütüphane fonksiyonu kullanan bir ifade yazın.
26. Şu iki veri üyesini içeren `dog` adlı bir sınıfın bildirimini yazın: `breed` adlı bir karakter katarı ve `age` adlı bir `int`. (Herhangi bir üye fonksiyon yazmayın.)
27. Doğru/Yanlış: Yeni programlarda Standard C++ `string` sınıfını kullanmaktansa C karakter katarlarını tercih etmelisiniz.
28. `string` sınıfının nesneleri
    a. null ile sonlandırılmıştır.
    b. atama operatörü ile kopyalanabilir.
    c. bellek yönetimi gerektirmez.
    d. üye fonksiyonları içermez.
29. `"cat"` karakter katarının `s1` karakter katarı içindeki yerini bulan bir ifade yazın.
30. `"cat"` karakter katarını `s1` karakter katarının 12. pozisyonuna ekleyen bir ifade yazın.

Alıştırmalar

Yıldızla işaretli alıştırmaların cevaplarını Ek G'de bulabilirsiniz.

1. Bir C karakter katarı (yani bir `char` dizisini) terse çeviren `reverseit()` adlı bir fonksiyon yazın. Öyle bir döngü oluşturun ki, önce ilk ve son karakterlerin yerini değiştirsin, sonra ikinci ve sondan ikinci karakterleri takas etsin ve o şekilde devam etsin. Karakter katarı `reverseit()`'e argüman olarak geçilmelidir.*

   `reversit()`'i kullanacağınız bir program yazın. Program önce kullanıcıdan bir karakter katarı almalı, `reversit()`'i çağırmalı ve sonucu yazmalıdır. Metin içine gömülü boşlukları kabul edebilen bir giriş yöntemi kullanın. Programı, Napoleon'un ünlü sözü "Able was I ere I saw Elba" ile test edin.
2. Bir isim (`string` sınıfından bir nesne) ve personel numarası (`long` tipi) içeren `employee` adlı bir sınıf tanımlayın. Kullanıcıdan alınan veriyi sınıfa yüklemek için `getdata()`, veriyi göstermek için de `putdata()` adlı üye fonksiyonları da yazın. İsim alanında gömülü boşluk olmadığını varsayın.*

   Bu sınıfı kullanmak üzere bir `main()` program oluşturun. Program, `employee` tipinden bir dizi oluşturmalı, daha sonra kullanıcının en fazla 100 personel verisini girmesini sağlamalıdır. Sonunda da program tüm personel verisinin dökümünü vermelidir.
3. Kullanıcı tarafından girilen ve İngiliz ölçüm sistemine göre ifade edilmiş en fazla 100 adet mesafe ölçümünün ortalamasını hesaplayan bir program yazın. Bu bölümdeki `ENGLARAY` örneğinde kullanılan `Distance` sınıfından nesnelerden oluşan bir dizi oluştu-

run. Ortalamayı hesaplamak için Bölüm 6'daki **ENGLCON** örneğinde kullanılan **add_dist()** üye fonksiyonundan yararlanabilirsiniz. Bir **Distance** değerini tamsayıya bölmek için başka bir üye fonksiyon daha kullanmanız gerekecektir. Örneğin şöyle yapabilirsiniz:

```
void Distance::div_dist(Distance d2, int divisor)
  {
  float fltfeet = d2.feet + d2.inches/12.0;
  fltfeet /= divisor;
  feet = int(fltfeet);
  inches = (fltfeet - feet) * 12.0;
  }
```

4. Kullanıcıdan tamsayıları okuyan ve bunları bir **int** dizisine kaydeden bir program yazmakla başlayın. Daha sonra dizi elemanlarına tek tek bakarak en büyüğünü arayan **maxint()** adlı bir fonksiyon yazın. Fonksiyonun argümanları; dizinin adresi ve dizinin içindeki eleman sayısı olmalıdır. Fonksiyon, en büyük elemanın indeksini döndürmelidir. Program bu fonksiyonu çağırmalı ve en büyük eleman ile onun indeksini göstermelidir. (Bu bölümdeki **SALES** programına bakın).
5. Bölüm 6'nın 11. ve 12. Alıştırmalarında gördüğümüz **fraction** sınıfından başlayın. Kullanıcıdan önceden belirlenmemiş sayıda kesir alan, bunları fraction tipinden bir dizide tutan, kesirlerin ortalamasını bulan ve sonucu gösteren bir **main()** programı yazın.
6. Briç oyununda dört oyuncunun her birine 13 kart verilir. Böylece destenin tamamı dağıtılmış olur. Bu bölümdeki **CARDARAY** programını şu şekilde değiştirin: Deste karıştırıldıktan sonra her biri 13'er karttan dört el oluşturulsun. Ardından da her oyuncunun elindeki kartlar gösterilsin.
7. Ticari program yazımında C++'ın zayıf olan yönlerinden biri $173,698,001.32 gibi parasal değerleri ifade etmeye yarayan tanımlanmış bir tip bulunmamasıdır. Böyle bir para tipinin sabit sayıda ondalık hane içermesi ve toplam 17 haneli olması gereklidir. (Amerika'nın dış borcunu dolar ve sent olarak ifade etmek için bu kadar hassaslık yeterlidir!) Neyse ki, C++'ın **long double** tipi 19 haneli hassasiyet sağlar. Dolayısıyla bir para sınıfını tanımlamak için bu tipi esas alabiliriz. Bu tipin ondalık noktasının kayan nokta olması problem oluşturmaz. Ancak para rakamlarını önünde dolar işaretli olarak ve virgülle ayrılmış üçer basamaklı şekilde okuyup yazabilme yeteneğini eklememiz gereklidir; böylece büyük rakamları daha okunaklı bir şekle sokabiliriz. Böyle bir sınıfı tanımlamanın ilk adımı olarak bir *para karakter katarını*, yani

```
"$1,234,567,890,123.99 "
```

şeklinde bir meblağı argüman olarak alıp karşılık gelen değeri **long double** tipinde döndüren **mstold()** adlı bir fonksiyon yazın.
Para karakter katarını bir karakter dizisi olarak ele almalı ve karakterleri tek tek değerlendirmelisiniz. Sadece (1-9 arasındaki) rakamları kopyalamalı ve bunları başka bir karakter katarına aktarmalısınız. Dolar işareti ve virgüller dahil diğer tüm karakterleri göz ardı etmelisiniz. Elde ettiğiniz saf karakter katarını **_atold()** kütüphane fonksiyonunu kullanarak **long double**'a çevirebilirsiniz. (Fonksiyon adının önündeki alt çizgi "_" karakterine dikkat edin. Başlık dosyası olarak **STDLIB.H** ya da **MATH.H** kullanabilirsiniz.) Para değerlerinin hiçbir zaman negatif olmayacağını varsayın. **mstold()**'u test etmek için bir **main()** programı yazın. Programı sürekli olarak kullanıcıdan bir para karakter katarı alsın ve buna karşılık gelen **long double** değerini göstersin.
8. C++'ın bir başka zayıf yönü de dizi indekslerinin limit dahilinde olup olmadığını otomatik olarak test etmemesidir. (Bu yaklaşım, dizi işlemlerini hızlandırır ama aynı zamanda daha az emniyetli kılar.) Diziye tüm erişimleri kontrol eden bir sınıf tanımlayarak güvenli bir diziyi kendiniz oluşturabiliriz.
Tek veri üyesi sabit boyutlu (bu boyut **LIMIT** olsun) bir tamsayı dizisi olan **safearray** adlı bir sınıf tanımlayın. Sınıfımızın iki üye fonksiyonu olsun. İlk fonksiyon **putel()**, kendisine argüman olarak geçilen indeks numarası ile bir **int** değerini alarak, bu tamsayıyı indeksin belirttiği dizi elemanına atasın. İkinci fonksiyon **getel()**, kendisine argüman olarak geçilen indeks numarasını alarak bu pozisyondaki dizi elemanının değerini **int** olarak döndürsün.

```
safearay sa1;                          //bir safearay nesnesi tanimla
int temp = 12345;                      //bir int degeri tanimla
sa1.putel(7, temp);                    //temp'in degerini dizinin 7'inci elemanina ata
temp = sa1.getel(7);                   //dizinin icindeki 7 indeksli degeri al
```

Her iki fonksiyon da indeks argümanına bakarak bu rakamın 0'dan küçük ya da **LIMIT-1**'den büyük olmadığını kontrol etmelidir. Böyle bir diziniz olduğunda kazara belleğin diğer kısımlarını bozma endişeniz olmadan dizinizi rahatça kullanabilirsiniz.
Dizi elemanlarına fonksiyonla erişmek [] operatörünü kullanmak kadar zarif görünmez. Bölüm 8'de bu operatörü aşırı yükleyerek **safearay** sınıfımızı aşağı-yukarı C++'ın kendi dizileri gibi kullanmanın yolunu öğreneceğiz.
9. Kuyruk, aynı yığın gibi bir veri depolama yapısıdır. İkisi arasındaki fark şudur: Yığında, son gelen eleman ilk çıkar. Kuyrukta ise ilk gelen ilk çıkar. Bir başka ifade ile yığın, LIFO (last-in-first-out) yaklaşımını kullanırken kuyruk, FIFO (first-in-first-out) yöntemi ile çalışır. Burada sözünü ettiğimiz kuyruk, tıpkı bir bankada sıra bekleyen müşterilerin oluşturduğu kuyruğa benzer: Kuyruğa ilk giren ilk hizmeti alır.
Bu bölümdeki **STAKARAY** programını, **stack** adlı sınıf yerine **queue** adlı bir sınıf içerecek şekilde yeniden yazın. Söz konusu sınıfta kurucuya ek olarak iki fonksiyon daha bulunmalıdır: kuyruğa veri eklemek için **put()** adlı bir fonksiyon ve kuyruktan veri çıkarmak için **get()** adlı bir fonksiyon. Bu fonksiyonlar **stack** sınıfındaki **push()** ve **pop()** fonksiyonlarına eşdeğerdir.
Gerek kuyruk gerekse de yığın, veri tutmaya yarayan birer dizidir. Yığının tepesini belirlemek için sadece **top** adlı bir **int** değişkeni kullanmanız yeterlidir. Ancak kuyrukta, kuyruğun başına işaret eden **head** ve sonunu gösteren **tail** adlı olmak üzere iki ayrı değişkene ihtiyacınız vardır. Yeni gelenler kuyruğa arkadan (**tail**) katılırlar (tıpkı bankaya son giren müşterinin kuyruğun sonuna girmesi gibi.) Elemanlar eklenip çıktıkça **tail** ve **head**'in yerleri de değişir. Bu da işi biraz karmaşıklaştırır: Kuyruğun başı ya da sonu kullanılan dizinin sınırına ulaştığında yeniden başa dönmek gerekir. Kuyruk sonunu çevirmek için şu şekilde bir ifadeye ihtiyaç duyarız:

```
if(tail == MAX-1)
tail = -1;
```

Kuyruğun başı için de benzer bir ifade gerekir. Kuyruk için kullanılan diziye bazen dairesel tampon denir. Çünkü kuyruğun başı ve sonu dizindeki veriler etrafında dolaşıp durur.

10. Matris, iki boyutlu bir dizidir. Alıştırma 7'deki dizi sınıfında kullandığımız güvenlik özelliğine sahip olan, yani dizi indeksinin limit dışına taşmamasını sağlayan, `matrix` adlı bir sınıf oluşturun. `matrix` sınıfının üye verileri 10'a 10'luk bir dizi olsun. Bir kurucu fonksiyon, programcının matrisin gerçek boyutlarını belirlemesine imkan sağlamalı (bu boyutlar 10x10'dan küçük olduğu sürece). Matristeki verilere erişim sağlayan üye fonksiyonların iki indeks numarasına ihtiyaçları olacaktır: Bunların her biri dizinin bir boyutuna erişmek için kullanılacaktır. Böyle bir sınıfı kullanan bir `main()` programın parçası aşağıdaki gibi olabilir:

```
matrix m1(3, 4);                //define a matrix object
int temp = 12345;               //define an int value
m1.putel(7, 4, temp);    //insert value of temp into matrix at 7,4
temp = m1.getel(7, 4);   //obtain value from matrix at 7,4
```

11. Alıştırma 6'daki para karakter katarlarını hatırlayın. `long double` olarak temsil edilen bir sayıyı para karakter katarına dönüştüren `ldtoms()` adlı bir fonksiyon yazın. Öncelikle, elinizdeki `long double` sayısının aşırı büyük olmamasını sağlayın. Tavsiyemiz; 9,999,999,999,999,990.00'dan daha büyük sayıları kullanmamanız yönündedir. Sayının bu limitte kaldığına emin olduktan sonra `long double`'ı bellekteki saf bir karakter katarına dönüştürün (yani dolar işareti ya da virgüller olmasın). Bunun için bu bölümde elde aldığımız `ostrstream` nesnesini kullanın. Elde edilen biçimlendirilmiş karakter katarı `ustring` adlı bir tampona koyulabilir. Ardından, başında dolar işareti olan yeni bir karakter katarı oluşturun. Rakamları `ustring`'ten tek tek alarak soldan sağa doğru yeni karakter katarına kopyalayın. Her üç rakamda bir virgül ilave edin. Sayının en solunda bir ya da daha fazla sıfır varsa bunları göz ardı etmeniz gerekecek. Örneğin $0,000,000,000,003,124.95 değil de $3,124.95 sayısını oluşturmak durumundasınız. Karakter katarının sonuna '`\0`' karakterini koymayı da unutmayın. Bu fonksiyonu kullanan bir `main()` program yazın. Kullanıcı sürekli olarak `long double` formatında sayılar girsin, sonuç para karakter katarı olarak çıksın.

12. `bMoney` adlı bir sınıf oluşturun. Bu sınıf, para meblağlarını `long double` olarak tutmalıdır. Daha sonra `long double` olarak girilen bir sayıyı para karakter katarına çevirmek için `mstold()` ve para karakter katarını göstermek üzere dönüştürmek için `ldtoms()` fonksiyonlarını kullanın. (Alıştırma 6 ve 10'a bakın.) Veri okumak ve göstermek için kullanacağınız fonksiyonlara `getmoney()` ve `putmoney()` adlarını verin. İki `bMoney` sayısını ekleyen bir başka üye fonksiyon yazın. Buna `madd()` diyebilirsiniz. `bMoney` nesnelerini eklemek kolaydır: Bunun için iki `bMoney` nesnesinin `long double` veri üyelerini birbiriyle toplamanız yeterlidir. Kullanıcının sürekli olarak iki para string'i girmesini sağlayan ve bunların toplamını gösteren bir `main()` program yazın. Sınıf belirteci aşağıdaki gibi olabilir:

```
class bMoney
  {
  private:
        long double money;
  public:
        bMoney();
        bMoney(char s[]);
        void madd(bMoney m1, bMoney m2);
        void getmoney();
        void putmoney();
  };
```


SEKİZİNCİ BÖLÜM

8

OPERATÖRLERİN AŞIRI YÜKLENMESİ

Tekli (Unary) Operatörlerin Aşırı Yüklenmesi

İkili Operatörlerin Aşırı Yüklenmesi

Veri Dönüşümü

UML Sınıf Şemaları

Operatörleri Aşırı Yüklemenin ve Dönüşümlerin Zorlukları

explicit ve mutable Anahtar Kelimeleri

Operatörleri aşırı yüklemek, nesne yönelimli programlamanın en heyecan verici özelliklerinden biridir. Bu sayede, karmaşık ve anlaşılması güç program listeleri, kişinin sezgileriyle anlaşılabilecek kadar açık bir hale dönüşür. Örneğin, şu tür ifadeler

```
d3.addobject(d1, d2);
```

veya aynı işlevi gören fakat aynı derecede anlaşılması güç olan şu tür ifadeler

```
d3 = d1.addobject(d2);
```

çok daha okunaklı olan şu forma dönüştürülebilir:

```
d3 = d1 + d2;
```

Biraz tehditkar bir ifade olan *operatörleri aşırı yüklemek* (*operator overloading*) ifadesi; normal C++ operatörleri (mesela +, *, <= ve +=), kullanıcı tarafından tanımlanan veri tiplerine uygulandıkları zaman, bu operatörlere ek anlamlar katmak anlamına gelir. Normal olarak

```
a = b + c;
```

ifadesi sadece `int` ve `float` gibi temel veri tipleriyle çalışır. `a`, `b` ve `c` kullanıcı tarafından tanımlanmış bir sınıfın nesneleri iken bu ifadeyi uygulamaya kalkışmak derleyiciden şikayetlerin yükselmesine neden olacaktır. Bununla birlikte, operatörleri aşırı yükleyerek `a`, `b` ve `c`, kullanıcının tanımladığı tipler olsa bile yukarıdaki ifadenin kurallara uygun bir ifade olmasını sağlayabilirsiniz.

Aslında, operatörlerin aşırı yüklenmesi size C++ dilini yeniden tanımlama fırsatı verir. C++ operatörlerinin çalışma biçimini kısıtlı buluyorsanız, bunları istekleriniz doğrultusunda çalışacak şekilde değiştirebilirsiniz. Yeni tipte değişkenler oluşturmak amacıyla sınıfları kullanarak ve operatörler için yeni tanımlar oluşturmak amacıyla aşırı yüklemeyi kullanarak, C++'ı birçok yönden geliştirip kendi tasarladığınız yeni bir dil haline getirebilirsiniz.

Bir diğer işlem türü olan *veri tiplerinin dönüştürülmesi*, operatörlerin aşırı yüklenmesiyle yakından ilişkilidir. C++, `int` ve `float` gibi temel veri tiplerinin dönüşümlerini otomatik olarak ele alır; fakat, kullanıcının tanımladığı tipleri içeren dönüşümler programcı tarafında biraz işlem gerektirir. Veri dönüşümlerine bu bölümün ikinci yarısında göz atacağız.

Aşırı yüklenen operatörlerle hayat pek de öyle güllük gülistanlık değildir. Bunların kullanımlarıyla ilgili bazı tehlikeleri bölümün sonunda ele alacağız.

Tekli (Unary) Operatörlerin Aşırı Yüklenmesi

Bir *tekli operatörü aşırı* yükleyerek konumuza başlayalım. Bölüm 2'den hatırlayabileceğiniz gibi, tekli operatörler yalnızca bir operand üzerinde işlem yapar. (Bir operand, üzerinde bir operatörün işlem yaptığı bir değişkenden ibarettir.) Artırma operatörü (++), eksiltme operatörü (--) ve bir operandlı eksiltme operatörü (-33'teki gibi) birli operatör örneklerindendir.

Bölüm 6'da "Nesneler ve Sınıflar" başlığı altında incelediğimiz `COUNTER` örneğinde, bir sayaçtaki toplamı takip etmek için `Counter` isimli bir sınıf oluşturmuştuk. Bu sınıfın nesneleri bir üye fonksiyon çağrılarak arttırılıyordu:

```
c1.inc_count();
```

Bu işe yaramıştı; fakat, bunun yerine artırma operatörünü (++) kullanmış olsaydık, program listesi çok daha okunaklı olabilirdi:

```
++c1;
```


Tecrübeli C++ (ve C) programcıları bu ifadenin `c1`'i artırdığını hemen tahmin etmişlerdir. Bunu gerçekleştirmek için `COUNTER` programını yeniden yazalım. `COUNTPP1`'in program listesi şöyledir:

```
// countpp1.cpp
// ++ operatorunu kullanarak sayac degiskenini artirir
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;              //count
   public:
      Counter() : count(0)             //kurucu fonksiyon
         { }
      unsigned int get_count()         //count degerini dondur
         { return count; }
      void operator ++()               //artir (onek)
         {
         ++count;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                     //tanimla ve ilk kullanima hazirla

   cout << "\nc1=" << c1.get_count();  //ekranda goster
   cout << "\nc2=" << c2.get_count();

   ++c1;                               //c1'i artir
   ++c2;                               //c2'yi artir
   ++c2;                               //c2'yi artir
   cout << "\nc1=" << c1.get_count();  //tekrar ekranda goster
   cout << "\nc2=" << c2.get_count() << endl;
   return 0;
   }
```

Bu programda `Counter` sınıfına ait iki nesne tanımlanır: `c1` ve `c2`. Nesnelerde tutulan sayılar ekranda gösterilir; sayaçlar başlangıçta 0 değerini taşır. Daha sonra, aşırı yüklenmiş ++ operatörü kullanılarak `c1` bir kez, `c2` iki kez artırılır ve elde dilen değerler ekranda görüntülenir. Programın çıktısı şöyledir:

```
c1=0          <===== sayaclar baslangicta 0
c2=0
c1=1          <===== bir kez artirildi
c2=2          <===== iki kez artirildi
```

Bu işlemlerin gerçekleşmesini sağlayan ifadeler şunlardır:

```
++c1;
++c2;
++c2;
```

++ operatörü c1'e bir kez, c2'ye iki kez uygulanır. Bu örnekte önek notasyonu kullanılır; sonek daha sonra incelenecektir.

operator Anahtar Kelimesi

Normal bir C++ operatörüne, kullanıcı tarafından tanımlanmış bir operand üzerinde çalışması gerektiğini nasıl öğretiyoruz? ++ operatörünü aşırı yüklemek için aşağıdaki deklaratörde **operator** anahtar kelimesi kullanılır:

```
void operator ++()
```

Deklaratörde, ilk önce dönüş değeri (bu örnekte, **void**) gelir, ardından **operatör** anahtar kelimesi, ardından operatörün kendisi (++) ve son olarak da parantezler içinde bir argüman listesi (bu örnekte parantezlerin içi boş) takip eder. Bu deklaratör söz dizimi, derleyici ne zaman bir ++ operatörü ile karşılaşırsa, operandın (++ operatörünün üzerinde işlem yaptığı değişken) **Counter** tipinde olması şartıyla, derleyicinin bu üye fonksiyonu çağırmasını bildirir.

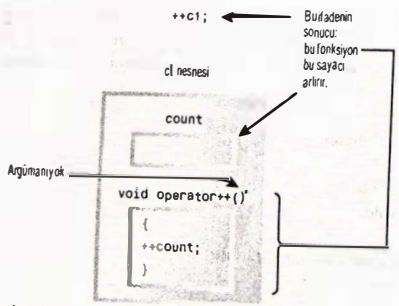
"Fonksiyonlar" adlı Bölüm 5'te, derleyicinin aşırı yüklenmiş fonksiyonları birbirinden ayırt etmesinin tek yolunun, argümanların veri tiplerine ve sayılarına bakması olduğunu görmüştük. Aynı şekilde, derleyicinin aşırı yüklenmiş operatörleri birbirinden ayırt etmesinin tek yolu operandların veri tiplerine bakmasıdır. Eğer operand, şu ifadede olduğu gibi **int** gibi temel bir veri tipinde ise, o zaman derleyici, **int**'i artırmak için kendi hazır rutinini kullanacaktır:

```
++intvar;
```

Fakat, şayet operand bir **Counter** değişkeni ise, derleyici bizim kendi yazdığımız **operator++()** fonksiyonumuzu kullanması gerektiğinin farkına varacaktır.

Operatör Argümanları

++c1 deyiminde görüldüğü gibi, **main()** içinde ++ operatörü belirli bir nesneye uygulanır. Üstelik **operator++()** argüman almaz. Peki, bu operatör neyi artırıyor? Kendisinin üyesi olduğu nesnenin **count** verisini artırır. Üye fonksiyonlar, kendilerini çağıran belirli bir nesneye her zaman erişebildikleri için bu operatör argüman gerektirmez. Bu durum Şekil 8.1'de gösteriliyor.



ŞEKİL 8.1: Aşırı yüklenmiş tekli operatör; argümansız.

Operatörlerin Döndürdükleri Değerler

COUNTPP1 programındaki **operator++()** fonksiyonunun güç fark edilen bir kusuru vardır. **main()**'de şöyle bir ifade kullanırsanız, bu kusuru tespit edebilirsiniz:

```
c1 = ++c2;
```

Bu durumda derleyici şikayet edecektir. Niçin? Çünkü **operator++()** fonksiyonunda ++ operatörünü, dönüş tipi **void** olacak şekilde tanımladık. Halbuki, atama ifadesinde bu operatörün **Counter** tipinde bir değer döndürmesi beklenir. Yani, derleyiciden istenen şu: ++ operatörünün **c2** üzerinde gerçekleştirdiği işlemlerden sonra **c2** hangi değere sahipse, bu değeri döndürmeli ve bunu **c1**'e atamalıdır. **COUNTPP1**'de de aynı şekilde tanımlandığı gibi, ++ operatörünü atama ifadelerinde **Counter** nesnelerini artırmak amacıyla kullanamayız. ++ operatörü her zaman operandı ile birlikte tek başına olmalıdır. Elbette, **int** gibi temel veri tiplerine uygulanan normal ++ operatöründe bu tür problemler yoktur.

Ev yapımı **operator++()** fonksiyonumuzun atama deyimlerinde kullanımını mümkün kılmak için fonksiyonun bir değer döndürmesini sağlayacak bir yol bulmalıyız. Sıradaki program **COUNTPP2** tam bu işi gerçekleştirir.

```
// countpp2.cpp
// ++operatoru kullanarak sayac degiskenini artir,degeri dondur
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;              //count
   public:
      Counter() : count(0)             //kurucu fonksiyon
         { }
      unsigned int get_count()         //count'u dondur
         { return count; }
      Counter operator ++ ()           //count'u artir
         {
         ++count;                      //count'u artir
         Counter temp;                 //gecici bir Counter hazirla
         temp.count = count;           //bu nesne degerinin aynini geciciye ata
         return temp;                  //kopyayi dondur
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                  //c1=0, c2=0

   cout << "\nc1=" << c1.get_count();      //ekranda goster
   cout << "\nc2=" << c2.get_count();

   ++c1;                         //c1=1
   c2 = ++c1;                       //c1=2, c2=2

   cout << "\nc1=" << c1.get_count();      //tekrar goster
   cout << "\nc2=" << c2.get_count() <<endl;
   return 0;
   }
```

Bu programda **operator++()** fonksiyonu **Counter** tipinde, **temp** isminde yeni bir nesne tanımlar. Bu yeni nesne, dönüş değeri olarak kullanılmak üzere tanımlanır. Fonksiyon, önceden olduğu gibi kendi nesnesi içindeki **count** verisini artırır, sonra yeni **temp** isimli nesneyi hazırlar ve bu yeni nesnenin içindeki **count**'a kendi nesnesindeki aynı değeri atar. En son olarak, **temp** nesnesini döndürür. Bu program istenilen sonucu verir. Şu tür deyimler artık bir değer döndürür:

```
++c1
```

Böylece, bu deyimler diğer deyimlerin içinde kullanılabilir. Örneğin,

```
c2 = ++c1;
```

main()'de gösterildiği gibi, **c1++**'tan dönen değer **c2**'ye atanır. Bu programın çıktısı şöyledir:

```
c1=0
c2=0
c1=2
c2=2
```

İsimsiz Geçici Nesneler

COUNTPP2'de **Counter** tipinde, **temp** isminde geçici bir nesne tanımladık. Bu nesnenin tek amacı **++** operatörü için bir dönüş değeri sağlamaktı. Bu işlemler, üç ifade kullanılmasını gerektirdi:

```
Counter temp;          //gecici bir Counter nesnesi hazirla
temp.count = count;    //bu nesnenin degerinin aynini geciciye ver
return temp;           //gecici nesneyi dondur
```

Fonksiyonlardan ve aşırı yüklenmiş operatörlerden geçici nesneleri döndürmenin daha uygun yolları da vardır. Şimdi bir başka yaklaşımı inceleyelim. **COUNTPP3** programında bu yaklaşım ele alınıyor.

```
// countpp3.cpp
// ++operatoru kullanilarak sayac degiskeni artirilir
// isimsiz gecici bir nesne kullanilir
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;              //count
   public:
      Counter() : count(0)             //kurucu fonksiyon, argumansiz
         { }
      Counter(int c) : count(c)        //kurucu fonksiyon,tek argumanli
         { }
      unsigned int get_count()         //count'u dondur
         { return count; }
      Counter operator ++ ()           //count'u artir
         {
```



```
         ++count;                         //count'u once artir, sonra dondur
         return Counter(count);           //isimsiz gecici nesneye
         }                                //bu count ilk degeri atanir
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                                  //c1=0, c2=0

   cout << "\nc1=" << c1.get_count();               //ekranda goster
   cout << "\nc2=" << c2.get_count();

   ++c1;                                            //c1=1
   c2=++c1;                                         //c1=2, c2=2

   cout << "\nc1=" << c1.get_count();               //tekrar goster
   cout << "\nc2=" << c2.get_count() << endl;
   return 0;
   }
```

Bu programda tek bir ifade **COUNTPP2**'de üç ifadenin yaptığı işi yapar:

```
return Counter(count);
```

Bu ifade **Counter** tipinde bir nesne üretir. Bu nesnenin ismi yoktur; bir isme ihtiyaç duyacak kadar uzun bir süre etrafta kalmaz. Bu isimsiz nesneye, **count** argümanı üzerinden sağlanan değer, ilk değeri olarak atanır.

Durun! Bu, tek argüman alan bir kurucu fonksiyon gerektirmez mi? Gerektirir; bu ifadenin düzgün çalışabilmesi için **COUNTPP3**'ün üye fonksiyonları listesine gizlice böyle bir kurucu fonksiyon ekleyiverdik.

```
Counter(int c) : count(c)          //kurucu fonksiyon,tek argumanli
   { }
```

İsimsiz nesneye **count**'un değeri atanır atanmaz, artık bu nesnenin değeri döndürülebilir. Bu programın çıktısı **COUNTPP2**'ninkiyle aynıdır.

COUNTPP2 ve **COUNTPP3**'teki yaklaşımların her ikisi de orijinal nesnenin (fonksiyonun üyesi olduğu nesne) bir kopyasını hazırlamaya ve bu kopyayı döndürmeye dayanır. ("Sanal Fonksiyonlar" adlı Bölüm 11'de öğreneceğimiz gibi, bir başka yaklaşım, **this** işaretçisini kullanarak orijinal nesnenin değerini döndürmektir.)

Sonek Notasyonu

Şu ana kadar artırma operatörünün yalnızca önek formunda kullanımını gösterdik.

```
++c1
```

Peki, sonek ne zaman kullanılır? Sonek, değişkenin değeri deyim içinde kullanıldıktan sonra değişkenin artırılmasını sağlar.

```
c1++
```

Artırma operatörünün her iki versiyonun da çalışmasını sağlamak için **POSTFIX** programında görüldüğü gibi, iki tane aşırı yüklenmiş ++ operatörü tanımlanır:

```
// postfix.cpp
// asiri yuklenmis ++ operatoru, hem onek hem de sonek formunda
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;                    //count
   public:
      Counter() : count(0)                   //kurucu fonksiyon,argumansiz
         { }
      Counter(int c) : count(c)              //kurucu fonksiyon,tek argumanli
         { }
      unsigned int get_count()const          //count'u dondur
         { return count; }

      Counter operator ++ ()                 //count'u artir (onek)
         {                                   //count'u once artir, sonra dondur
         return Counter(++count);            //isimsiz gecici bir nesneye
         }                                   //bu count ilk degeri atanir

      Counter operator ++ (int)              //count'u artir (sonek)
         {                                   //once, bu count ilk degeri atanmis
         return Counter(count++);            //isimsiz gecici nesneyi dondur,
         }                                   //sonra count'u artir
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                           //c1=0,c2=0

   cout << "\nc1=" << c1.get_count();        //ekranda goster
   cout << "\nc2=" << c2.get_count();

   ++c1;                                     //c1=1
   c2 = ++c1;                                //c1=2, c2=2 (onek)

   cout << "\nc1=" << c1.get_count();        //ekranda goster
   cout << "\nc2=" << c2.get_count();

   c2 = c1++;                                //c1=3, c2=2 (sonek)

   cout << "\nc1=" << c1.get_count();        //tekrar ekranda goster
   cout << "\nc2=" << c2.get_count() << endl;
   return 0;
   }
```

++ operatörünün işlevini artırmak için artık iki farklı deklaratör mevcuttur. Önceden sonek notasyonu için gördüğümüz deklaratör şöyle idi:

```
Counter operator ++()
```


Sonek notasyonu için kullandığımız yeni deklaratör ise şöyledir:

```
Counter operator ++(int)
```

İkisi arasındaki tek fark parantez içindeki **int**'tir. Buradaki **int** gerçekte bir argüman değildir. Ayrıca, tamsayı anlamına da gelmez. Bu, sadece derleyiciye operatörün sonek versiyonunu hazırlamasını işaret etmekten ibaret bir ifadedir. C++ tasarımcıları, mevcut operatörleri ve anahtar kelimeleri yeniden kullanarak bunlara birden fazla görev yüklemekten çok hoşlanıyorlar; **int** de tasarımcıların, soneki belirtmesi için seçtikleri bir ifadedir. (Siz daha iyi bir söz dizimi düşünebiliyor musunuz?) Programın çıktısı işte şöyledir:

```
c1=0
c2=0
c1=2
c2=2
c1=3
c2=2
```

Bu çıktı satırlarından ilk dördünü **COUNTPP2** ve **COUNTPP3**'ün çıktılarında görmüştük. Ancak, son iki satır aşağıdaki ifadenin sonucu:

```
c2 = c1++;
```

Bu ifadede, **c1** artırılarak 3 değerine ulaşır; fakat, **c1**'in değeri artırılmadan önce **c2**'ye atanır, böylece c2, 2 değerini muhafaza etmiş olur.

Elbette, aynı yaklaşımı eksiltme operatörü (--) ile de kullanabilirsiniz.

İkili Operatörlerin Aşırı Yüklenmesi

İkili operatörlerin aşırı yüklenmesi, tıpkı tekli operatörlerin aşırı yüklenmesi kadar kolaydır. Şimdi, aritmetik operatörünü, karşılaştırma operatörünü ve aritmetik atama operatörünü aşırı yükleyen örneklere bir göz atalım.

Aritmetik Operatörler

Bölüm 6'daki **ENGLCON** programında, İngiliz ölçü sistemine göre uzaklığı simgeleyen iki **Distance** nesnesinin **add_dist()** üye fonksiyonu kullanılarak nasıl toplanabileceklerini gösterdik:

```
dist3.add_dist(dist1, dist2);
```

+ operatörünü aşırı yükleyerek, bu yoğun görünümlü deyimi şu şekle indirgeyebiliriz:

```
dist3 = dist1 + dist2;
```

İşte tam olarak bunu yapan **ENGLPLUS** programının listesi:

```
// englplus.cpp
// asiri yuklenmis '+' operatoru iki uzakligi toplar
#include <iostream>
```

```
using namespace std;
//////////////////////////////////////////////////////////////
class Distance    //Ingiliz uzaklik olculerine dayanan Distance sinifi
   {
   private:
      int feet;
      float inches;                          //kurucu fonksiyon (argumansiz)
   public:
      Distance() : feet(0), inches(0.0)
         { }                                 //kurucu fonksiyon (iki argumanli)
      Distance(int ft, float in) : feet(ft), inches(in)
         { }
      void getdist()                         //uzakligi kullanicidan al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }
      void showdist() const                  //uzakligi ekranda goster
         { cout << feet << "\'-" << inches << '\"';}
      Distance operator + (Distance) const;       //2 uzakligi topla
   };
//-----------------------------------------------------------
                                             //bu uzakligi d2'ye ekle
Distance Distance::operator + (Distance d2) const        //toplami dondur
   {
   int f = feet + d2.feet;                   //feet'leri topla
   float i = inches + d2.inches;             //inches'leri topla
   if(i >= 12.0)                             //eger toplam 12.0'yi gecerse,
      {                                      //inches'i
      i -= 12.0;                             // 12 azalt ve
      f++;                                   //feet'i 1 artir
      }                                      //degeri sum ilk olarak atanmis
   return Distance(f,i);                     //gecici bir Distance dondur
   }
//////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist3, dist4;             //uzakliklari tanimla
   dist1.getdist();                          //dist1'i kullanicidan al

   Distance dist2(11, 6.25);                 //dist2'yi tanimla ve ilk degeri ata

   dist3 = dist1 + dist2;                    //tek '+' operatoru

   dist4 = dist1 + dist2 + dist3;            //birkac tane '+' operatoru
                                             //tum uzakliklari ekranda goster
   cout << "dist1= "; dist1.showdist(); cout << endl;
   cout << "dist2= "; dist2.showdist(); cout << endl;
   cout << "dist3= "; dist3.showdist(); cout << endl;
   cout << "dist4= "; dist4.showdist(); cout << endl;
   return 0;
   }
```

Bir toplamanın sonucu bir değer atama ifadesi içinde yer alabilir. Bunun yanı sıra, bir toplama sonucunun, bir başka toplama içinde de kullanılabileceğini göstermek için **main()**'de başka bir toplama daha gerçekleştirilir. Aşağıdaki ifade ile **dist4**'ü elde etmek için **dist1**, **dist2** ve **dist3** toplanır. (**dist4**, **dist3**'ün değerinin iki katı olmalıdır.)


```
dist4 = dist1 + dist2 + dist3;
```

Programın çıktısı şöyledir:

```
Enter feet: 10
Enter inches: 6.5
dist1 = 10'-6.5"     <==== kullanıcıdan alınır
dist2 = 11'-6.25"    <==== değeri programda atanır
dist3 = 22'-0.75"    <==== dist1+dist2
dist4 = 44'-1.5"     <==== dist1+dist2+dist3
```

Distance sınıfının içinde, **operator++()** fonksiyonunun deklarasyonu şöyle görünür:

```
Distance operator + ( Distance );
```

Bu fonksiyon **Distance** tipinde bir değer döndürür ve **Distance** tipinde bir tane argüman alır.

Şu tür deyimlerde dönüş değerinin ve operatörün argümanlarının nesnelerle nasıl bir bağlantı içinde olduğunu kavramak önemlidir:

```
dist3 = dist1 + dist2;
```

Derleyici bu deyimi gördüğü zaman argüman tiplerine bakar; argümanların sadece **Distance** tipinde olduğunu fark edince **Distance** sınıfının üye fonksiyonu olan **operator+()** fonksiyonunu kullanması gerektiğini anlar. Fakat bu fonksiyon argüman olarak ne kullanır? **dist1**'i mi yoksa **dist2**'yi mi kullanır? Ayrıca, toplanacak iki sayı olduğuna göre fonksiyonun iki tane argümana ihtiyacı yok mu?

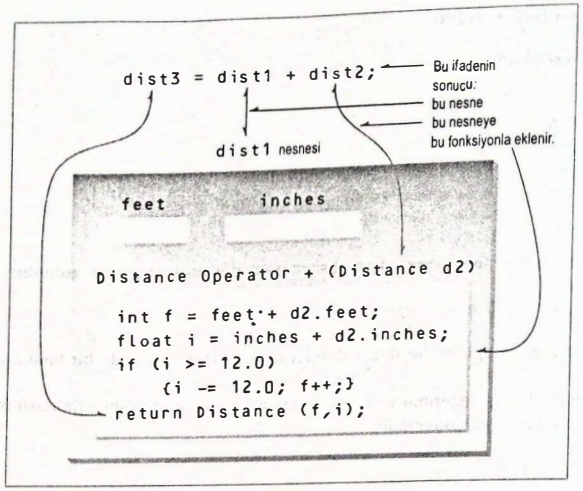
İşte çözüm: Operatörün *sol tarafında* kalan argüman (bu örnekte, **dist1**), operatörün üyesi olduğu nesnedir. Operatörün *sağ tarafında* yer alan nesne ise (**dist2**) , operatöre argüman olarak sağlanmalıdır. Operatör bir değer döndürür; bu değer başka bir değişkene atanabilir veya başka şekillerde kullanılabilir. Bu örnekte, operatörden dönen değer **dist3**'e atanır. Şekil 8.2 bunun nasıl gerçekleştirildiğini gösterir.

operator+() fonksiyonunda, **feet** ve **inches** kullanılarak soldaki operanda doğrudan erişilir; çünkü bu zaten operatörün üyesi olduğu nesnenin kendisi. Sağdaki operanda ise fonksiyonun argümanı olarak erişilir. Burada **d2.feet** ve **d2.inches** ifadeleri kullanılır.

Bunu genelleştirip, şöyle ifade edebiliriz: Aşırı yüklenmiş bir operatör her zaman operand sayısından bir az argüman gerektirir, çünkü operandlardan biri zaten operatörün üyesi olduğu nesnedir. Birli operatörlerin argüman gerektirmemesinin sebebi de budur. (Bu kural arkadaş (friend) fonksiyonlara ve operatörlere uygulanamaz. C++'ın bu özelliklerini Bölüm 11'de ele alacağız.)

ENGLPLUS'taki **operator+()**'ın döndürdüğü değeri hesaplamak için önce her iki operandın **feet** ve **inches** değerleri toplanır (eğer toplamada "elde" oluşuyorsa bu da hesaba katılır). Elde edilen değerler olan **f** ve **i**, daha sonra isimsiz bir **Distance** nesnesine değer atamak için kullanılır. Bu isimsiz **Distance** nesnesi ise aşağıda ifade kullanılarak fonksiyondan döndürülür:

```
return Distance(f, i);
```



ŞEKİL 8.2: Aşırı yüklenmiş ikili operatör: Tek argümanlı.

Bütün bunlar **COUNTPP3**'teki ayarlamalarla aynıdır. Yalnız, burada kurucu fonksiyon bir yerine iki argüman alır. **main()**'deki şu ifade isimsiz **Distance** nesnesinin değerini **dist3**'e atar:

```
dist3 = dist1 + dist2;
```

Anlaşılması çok derece net olan bu ifadeyi, Bölüm 6'daki **ENGLCON** örneğinde aynı işi gören fonksiyon çağrısının kullanımıyla karşılaştırın.

Distance sınıfındaki diğer operatörler de benzer fonksiyonlar tanımlanarak aşırı yüklenebilir. Böylece, bu sınıfa ait nesneler üzerinde yapılan çıkarma, çarpma ve bölme işlemleri daha doğal görünümlü olur.

Karakter Katarlarını Birbirine Eklemek

+ operatörü C karakter kararlarını birbirine eklemek için kullanılamaz. Yani, **str1**, **str2** ve **str3**'ün C karakter katarı (**char** tipinde dizi) değişkenleri olarak yer aldığı şöyle bir ifade yazamazsınız:

```
str3 = str1 + str2;
```

Bu ifade ile elde edilmek istenen, "cat" artı "bird" eşittir "catbird" gibi bir sonuç. Ancak, eğer kendi **String** sınıfımızı (Bölüm 6'da **STROBJ** programında gösterildiği gibi) kullanırsak, böyle bir ekleme gerçekleştirmesi için + operatörünü aşırı yükleyebiliriz. Aslında Standart C++ **string** sınıfının yaptığı da tam anlamıyla budur; fakat, bizim daha az iddialı **String** sınıfımızda, bunun nasıl gerçekleştirildiğini anlamak daha kolaydır. + operatörünü kesinlikle top-

lama olmayan bir iş yapmak için yeni bir işlevle aşırı yüklemek, C++ dilini yeniden tanımlayabilmemizin bir başka örneğini teşkil eder. **STRPLUS**'un program listesi şöyledir:

```
// strplus.cpp
// asiri yuklenmis '+' operatoru karakter katarlarini pespese ekler
#include <iostream>
using namespace std;
#include <string.h>                 //strcpy(), strcat() icin
#include <stdlib.h>                 //exit() icin
////////////////////////////////////////////////////////////
class String      //kullanici tarafindan tanimlanan karakter katari tipi
   {
   private:
      enum { SZ=80 };                       //String nesnelerinin boyutu
      char str[SZ];                         //bir string tutmak icin
   public:
      String()                              //kurucu fonksiyon, argumansiz
         { strcpy(str, ""); }
      String(char s[])                      //kurucu fonksiyon,bir argumanli
         { strcpy(str, s); }
      void display() const                  //String'i ekranda goster
         { cout << str; }
      String operator + (String ss)const    //String'leri ekle
         {
         String temp;                       //gecici bir String hazirla
         if( strlen(str) + strlen(ss.str) < SZ )
            {
            strcpy(temp.str, str);          //bu karakter katarini temp'e kopyala
            strcat(temp.str, ss.str);       //arguman karakter katarini ekle
            }
         else
            { cout << "\nString overflow"; exit(1); }
         return temp;                       //temp String'i dondur
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   String s1 = "\nMerry Christmas! ";       //2. Kurucu fonksiyonu kullanir
   String s2 = "Happy new year!";           //2. Kurucu fonksiyonu kullanir
   String s3;                               //1. Kurucu fonksiyonu kullanir

   s1.display();                            //karakter katarlarini ekranda goster
   s2.display();
   s3.display();

   s3 = s1 + s2;                            //s2'yi s1'e ekle
                                            //ekleme sonucunu s3'e ata
   s3.display();                            //s3'u ekranda goster
   cout << endl;
   return 0;
   }
```

Program ilk önce üç karakter katarını ayrı ayrı ekranda gösterir. (Bu aşamada üçüncü karakter katarı henüz boştur; bu nedenle, üçüncü karakter katarı görüntülenirken ekrana bir şey basılmaz.) Sonra, birinci ve ikinci karakter katarları birbirine eklenir ve üçüncüye yerleştirilir; üçüncü karakter katarı tekrar görüntülenir. Programın çıktısı şöyledir:

```
Merry Christmas! Happy new year!  <==== s1, s2 ve s3 (bos)
Merry Christmas! Happy new year!  <==== s3, ekleme isleminden sonra
```

Şu andan itibaren + operatörünün işlevini artırmakla ilgili temel unsurlar artık biraz tanıdık gelmeye başlamalıdır. Aşağıdaki bildirim + operatörünün **String** tipinde bir argüman aldığını ve aynı tipte bir nesne döndürdüğünü gösterir:

```
String operator + (String ss)
```

operator+() içindeki ekleme işlemi **String** tipinde geçici bir nesne tanımlamakla işe başlar, sonra kendi **String** nesnemizin içindeki karakter katarını bu nesneye kopyalar, **strcat()** kütüphane fonksiyonunu kullanarak argüman olarak aktarılan karakter katarını geçici nesne içindeki karakter katarının peşine ekler ve geçici karakter katarını döndürür. Burada, aşağıdaki gibi, isimsiz geçici bir **String** oluşturan bir yaklaşım kullanamayacağımıza dikkat edin:

```
return String(string);
```

Çünkü hem ilk değer ataması yapmak için hem de argüman karakter katarını peşine eklemek için geçici **String**'e erişmemiz gerekir.

String sınıfında kullanılan sabit uzunluktaki karakter katarlarını taşırmamaya özen göstermemiz gerekir. Bu tür kazaları önlemek için eklenecek iki karakter katarının birleştirilmiş uzunluğunun maksimum karakter katarı uzunluğunu aşıp aşmadığı, **operator+()** fonksiyonunda kontrol edilir. Eğer aşıyorsa, karakter katarlarını ekleme işlemini sürdürmek yerine ekrana bir hata mesajı verilir. (Hataları farklı biçimlerde de ele alabiliriz. Hata olduğunda 0 döndürebiliriz veya daha da iyisi, kural dışı bir durum oluşturabilirsiniz. Kural dışı durumları Bölüm 14'te "Şablonlar ve Kural Dışı Durumlar" konusunda ele alacağız.)

Hatırlarsanız, **SZ** sabitine bir değer atamak için **enum** kullanmak geçici bir çözümdür. Tüm derleyiciler Standart C++ ile uyumlu hale gelince bu ifadeyi aşağıdaki ifade ile değiştirebilirsiniz:

```
static const int SZ=80;
```

Çoklu Aşırı Yükleme

+ operatörünün değişik kullanımlarını gördük: İngiliz ölçü sistemine göre uzaklıkların toplanmasında ve karakter katarlarının peş peşe eklenmesinde. Bu sınıfların her ikisini de birlikte aynı programa koyabilirsiniz; buna rağmen C++, + operatörünü nasıl ele alması gerektiğini bilecektir: "Toplama" işlemini gerçekleştirmek için doğru fonksiyonu, operandın tipine bağlı olarak seçecektir.

Karşılaştırma Operatörleri

Şimdi başka bir tür C++ operatörünün işlevini artırmayı öğrenelim: Karşılaştırma operatörü.

Mesafeleri Karşılaştırmak

İlk örneğimizde **Distance** sınıfı içinde *küçüktür* operatörünü (<) aşırı yükleyeceğiz. Böylece, iki uzaklığı karşılaştırabileceğiz. **ENGLESS** programının listesi şöyledir:


```
// engless.cpp
// asiri yuklenmis '<' operatoru iki uzakligi karsilastirir
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance       //Ingiliz uzaklik olculerine gore Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:                              //kurucu fonksiyon (argumansiz)
      Distance() : feet(0), inches(0.0)
         { }                            //kurucu fonksiyon (iki argumanli)
      Distance(int ft, float in) : feet(ft), inches(in)
         { }
      void getdist()                    //uzakligi kullanicidan al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }
      void showdist() const             //uzakligi ekranda goster
         { cout << feet << "\'-" << inches << '\"'; }
      bool operator < (Distance) const;    //uzakliklari karsilastir
   };
//-----------------------------------------------------------
                                        //bu uzakligi d2 ile karsilastir
bool Distance::operator < (Distance d2) const     //toplami dondur
   {
   float bf1 = feet + inches/12;
   float bf2 = d2.feet + d2.inches/12;
   return (bf1 < bf2) ? true : false;
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1;                      //Distance dist1'i tanimla
   dist1.getdist();                     //dist1'i kullanicidan al

   Distance dist2(6, 2.5);              //dist2'yi tanimla ve deger ata
                                        //uzakliklari ekranda goster
   cout << "\ndist1 = "; dist1.showdist();
   cout << "\ndist2 = "; dist2.showdist();

   if(dist1 < dist2)                    //islevi artirilmis '<' operatoru
      cout << "\ndist1 is less than dist2";
   else
      cout << "\ndist1 is greater than (or equal to)dist2";
   cout << endl;
   return 0;
   }
```

Bu program, kullanıcının girdiği uzaklığı program tarafından atanan bir uzaklıkla (6'-2.5") karşılaştırır. Sonuca bağlı olarak, iki muhtemel ifadeden birini ekrana yazar. Tipik bir program çıktısı şöyle olabilir:

```
Enter feet: 5
Enter inches: 11.5
```

```
dist1 = 5'-11.5"
dist2 = 6'-2.5"
dist1 is less than dist2
```

ENGLESS'teki **operator<()** fonksiyonunda kullanılan yaklaşım, **ENGLPLUS** programındaki + operatörünü aşırı yükleme işlemiyle aynıdır. Yalnızca, buradaki **operator<()** fonksiyonu **bool** tipinde bir değer döndürür. İki uzaklığın karşılaştırılmasına bağlı olarak fonksiyondan dönen değer **true** veya **false** olur. Karşılaştırma işlemi, her iki uzaklığın kayan noktalı ayak değerine dönüştürülmesi ve bu değerlerin normal < operatörü ile karşılaştırılması yoluyla yapılır. Hatırlarsanız, koşul operatörünün kullanımı

```
return (bf1 < bf2) ? true : false;
```

ile aşağıdaki kod aynı işleve sahiptir:

```
if(bf1 < bf2)
   return true;
else
   return false;
```

Karakter Katarlarını Karşılaştırmak

Gelin şimdi de *eşittir* (==) operatörünü aşırı yüklemeye dair bir örnek görelim. Bu operatörü, kendi tanımladığımız **String** nesnelerinden iki tanesini karşılaştırmakta kullanacağız. İki **String** nesnesi eşitse **true**, farklıysa **false** değeri döndüreceğiz. **STREQUAL**'ın kodu şöyledir:

```
// strequal.cpp
// asiri yuklenmis '==' operatorunun karakter katarlarini karsilastirmasi
#include <iostream>
using namespace std;
#include <string.h>                    // strcmp() icin
////////////////////////////////////////////////////////////
class String            //kullanici tarafindan tanimlanan karakter katari tipi
   {
   private:
      enum { SZ = 80 };              // String nesnelerinin buyuklugu
      char str[SZ];                  // karakter katari tutar
   public:
      String()                       //kurucu fonksiyon; argumani yok
         { strcpy(str, ""); }
      String( char s[] )             //kurucu fonksiyon; tek argumani var
         { strcpy(str, s); }
      void display()const            //bir String gostermek icin
         { cout << str; }
      void getstr()                  //karakter katari okumak icin
         { cin.get(str, SZ); }
      bool operator == (String ss) const   //esitlik olup olmadigina bak
         {
         return ( strcmp(str, ss.str)==0 ) ? true : false;
         }
   };
int main()
   {
   String s1 = "yes";
   String s2 = "no";
```



```
   String s3;

   cout << "\nEnter 'yes' or 'no': ";
   s3.getstr();                        //kullanicidan String al

   if(s3==s1)                          // "yes" ile karsilastir
      cout << "You typed yes\n";
   else if(s3==s2)                     // "no" ile karsilastir
      cout << "You typed no\n";
   else
      cout << "You didn't follow instructions\n";
   return 0;
   }
```

Bu programın **main()** kısmında == operatörü iki kez kullanılır. İlkinde, kullanıcının girdiği karakter katarının "yes" olup olmadığına, ikincisinde de "no" olup olmadığına bakılır. Kullanıcının "yes" girdiği durumda programın çıktısı şöyle gerçekleşir:

```
Enter 'yes' or 'no': yes
You typed yes
```

operator==() fonksiyonu, iki C karakter katarını karşılaştırmak için kütüphane fonksiyonu **strcmp()**'i kullanır. Bu fonksiyon, karakter katarları eşit ise 0, birincisi ikincisinden küçükse negatif bir sayı, birincisi ikincisinden büyükse pozitif bir sayı döndürür. Burada "*küçüktür*" ve "*büyüktür*" derken leksikografik düzenden yani alfabetik sıralamadan bahsediyoruz. Bir karakter katarının diğerinden bu anlamda küçük olması demek, sözlükte daha önce gelecek olması demektir.

and ve > gibi karşılaştırma operatörleri de karakter katarlarının leksikografik değerlerini kıyaslamak için kullanılabilir. Alternatif olarak, bu karşılaştırma operatörleri karakter katarı uzunluklarını karşılaştırmak üzere yeniden tanımlanabilirler. Operatörlerin nasıl kullanılacağını kendiniz tanımladığınız için, durumunuza uyan her türlü tanımı kullanabilirsiz.

Aritmetik Atama Operatörleri

Aşırı yüklenmiş ikili operatörlerle ilgili incelememizi aritmetik bir atama operatörü olan += ile noktalayacağız. Hatırlarsanız, bu operatör atama ile toplama işlemlerini tek adımda birleştiriyordu. Şimdi bu operatörü kullanarak İngiliz uzaklık ölçülerine göre ifade edilmiş bir uzaklığı aynı cinsten bir başka ölçü ile toplayacağız ve sonucu da ilk değişkene atayacağız. Daha önce gördüğümüz **ENGLPLUS** örneğine benzeyen bir durum ama burada ince bir fark var. **ENGLPLEQ**'in program listesine bir bakalım:

```
// englpleq.cpp
// asiri yuklenmis atama operatoru '+='
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance          //Ingiliz olcu sistemine dayanan Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:                                    //kurucu fonksiyon (argumansiz)
      Distance() : feet(0), inches(0.0)
```

```
                                        //kurucu fonksiyon (iki argumanli)
      { }
   Distance(int ft, float in) : feet(ft), inches(in)
      { }                               //uzunlugu kullanicidan al
   void getdist()
      {
      cout << "\nEnter feet: "; cin >> feet;
      cout << "Enter inches: "; cin >> inches;
      }
   void showdist() const                //uzakligi goster
      { cout << feet << "\'-" << inches << '\"'; }
   void operator += (Distance);
};
//--------------------------------------------------------
                                        //mesafeyi bu mesafeye ekle
void Distance::operator += (Distance d2)
   {
   feet += d2.feet;                     //feet'i ekle
   inches += d2.inches;                 //inches'i ekle
   if(inches >= 12.0)                   //toplami 12.0'i asiyorsa,
      {                                 //inches'i
      inches -= 12.0;                   //12.0 azalt ve
      feet++;                           //feet'i
      }                                 //1 artir
   }
////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1;                      //dist1'i tanimla
   dist1.getdist();                     //dist1'i kullanicidan al
   cout << "\ndist1 = "; dist1.showdist();

   Distance dist2(11, 6.25);            //dist2'yi tanimla, ilk degeri ata
   cout << "\ndist2 = "; dist2.showdist();

   dist1 += dist2;                      //dist1 = dist1 + dist2
   cout << "\nAfter addition,";
   cout << "\ndist1 = "; dist1.showdist();
   cout << endl;
   return 0;
   }
```

Bu programda, kullanıcıdan bir uzaklık ölçüsü alınır ve buna program tarafından 11'-6.24" başlangıç değeri verilmiş ikinci bir uzaklık ölçüsü eklenir. Programın örnek bir çalışması aşağıdaki gibidir:

```
Enter feet: 3
Enter inches: 5.75
dist1 = 3'-5.75"
dist2 = 11'-6.25"
After addition,
dist1 = 15'-0"
```

Programda toplama işlemi `main()`'de şu ifadede gerçekleşir:

```
dist1 += dist2;
```


Böylece `dist1` ile `dist2`'nin toplamı `dist1`'e atanır. Burada kullanılan `operator+=()` fonksiyonu ile `ENGLPLUS`'ta kullanılan `operator+()` fonksiyonu arasındaki farka dikkat edin. Daha önce kullandığımız `operator+()` fonksiyonunda toplama sonucunun üçüncü bir `Distance` nesnesine atanabilmesi için `Distance` tipinden yeni bir nesne oluşturulması gerekiyordu ki şu işlem gerçekleşebilsin:

```
dist3 = dist1 + dist2;
```

`ENGLPLEQ` örneğindeki `operator+=()` fonksiyonunda ise, toplama işleminin sonucunu atadığımız nesne, fonksiyonun üyesi olduğu bir nesnedir. Dolayısıyla, biz burada `feet` ve `inches`'a değer atıyoruz; sadece bir nesne döndürebilmek için kullandığımız geçici değişkenlere değil... `operator+=()` fonksiyonu herhangi bir değer döndürmez; dönüş tipi `void`'dir. Bu örnekte kullandığımız `+=` gibi aritmetik atama operatörlerinde dönüş değeri gerekli değildir. Çünkü, atama operatörünün sonucu herhangi bir şeye atanmaz.

Programda kullanılan türden deyimlerde operatör tek başınadır:

```
dist1 += dist2;
```

Bu operatörü aşağıdaki gibi karmaşık ifadelerde de kullanmak isteyebilirsiniz:

```
dist3 = dist1 += dist2;
```

İşte böyle bir durumda bir dönüş değerine ihtiyacınız olacaktır. Bunun için `operator+=()` fonksiyonunu şöyle bir deyimle bitirebilirsiniz:

```
return Distance(feet, inches);
```

Bu durumda isimsiz bir nesne, bu nesne ile aynı değere eşitlenir ve döndürülür.

İndeks Operatörü ([])

Normal olarak bir dizinin elemanlarına erişmek için kullanılan indeks operatörü ([ ]) de aşırı yüklenebilir. C++'taki dizilerin çalışma stillerini değiştirmek istiyorsanız bu özellikten yararlanabilirsiniz. Örneğin, "güvenli" bir dizi oluşturmak isteyebilirsiniz: Diziye erişmek için kullandığınız indeks numaralarını otomatik olarak kontrol eden bir dizi oluşturabilirsiniz. Böylece, indeksin dizi sınırları dışına çıkmaması garanti edilmiş olur. (Ayrıca, Bölüm 15'te "Standart Şablon Kütüphanesi" başlığı altında ele alınan `vector` sınıfını da kullanabilirsiniz.)

Aşırı yüklenmiş indeks operatörünü açıklamak için bir başka konuya, ilk kez Bölüm 5'te bahsedilen, fonksiyonlardan referans yoluyla değer döndürme konusuna geri dönmemiz lazım. İşe yaraması için indeks operatörünün referans olarak dönmesi gerekir. Bunun niçin böyle olduğunu anlamak için güvenli dizi uygulaması içeren üç örnek göstereceğiz. Dizi elemanlarını eklemek ve okumak için programların her biri farklı bir yaklaşım izler:

- Ayrı ayrı `put()` ve `get()` fonksiyonları
- Referans olarak dönen bir tek `access()` fonksiyonu
- Referans olarak dönen ve aşırı yüklenmiş `[]` operatörü

Programların üçü de `safearray` isimli bir sınıf tanımlarlar. `safearray` sınıfının tek elemanı, 100 `int` değerinden oluşan bir dizidir. Programların üçü de, diziye yapılan tüm erişimlerin dizi

sınırların içinde kalmasını garanti etmek için erişimleri kontrol ederler. Programların her birindeki `main()`, güvenli diziyi değerlerle doldurarak (dizinin her elemanı kendi indeksinin 10 katına eşit olacak şekilde) sınıfı test ederler, sonra kullanıcıya her şeyin olması gerektiği gibi çalıştığını kanıtlamak için dizi elemanlarını ekranda gösterirler.

Ayrı Ayrı get() ve put() Fonksiyonları

Birinci program, dizi elemanlarına erişmek için iki fonksiyon sunar: `putel()`, diziye bir eleman eklemek için; `getel()`, bir dizi elemanının değerini bulmak için kullanılır. Her iki fonksiyon da, dizi sınırları dışına çıkmamasını garanti etmek için indeks numarasının değerini kontrol eder. Yani, 0'dan küçük veya dizi büyüklüğünden (dizi büyüklüğü eksi 1) büyük olup olmamasını incelerler. **ARROVER1** programının listesi şöyledir:

```
// arrover1.cpp
// guvenli bir dizi tanimlar
// (dizi elemanina erismeden once indeks degeri kontrol edilir)
// ayri ayri put ve get fonksiyonlari kullanir
#include <iostream>
using namespace std;
#include <process.h>                    //exit() icin
const int LIMIT = 100;
////////////////////////////////////////////////////////////
class safearay
   {
   private:
      int arr[LIMIT];
   public:
      void putel(int n, int elvalue)       //elemanin degerini ayarla
         {
         if( n<0 || n>=LIMIT)
            { cout << "\nIndex out of bounds"; exit(1); }
         arr[n] = elvalue;
         }
      int getel(int n) const               //elemanin degerini al
         {
         if( n<0 || n>=LIMIT)
            { cout << "\nIndex out of bounds"; exit(1); }
         return arr[n];
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   safearay sa1;

   for(int j=0; j<LIMIT; j++)             //elemanlari ekle
      sa1.putel(j, j*10);

   for(j=0; j<LIMIT; j++)                 //elemanlari ekranda goster
      {
      int temp = sa1.getel(j);
      cout << "Element " << j << " is " << temp << endl;
      }
   return 0;
   }
```


Veriler güvenli diziye `putel()` üye fonksiyonu ile eklenir; `getel()` fonksiyonu ile de eklanda gösterilir. Bu program, bir güvenli dizi uygulaması gerçekleştirir. Dizi sınırları dışında kalan bir indeks numarası kullanmaya kalkışırsanız hata mesajı ile karşılaşırsınız. Her şeye rağmen, kullanılan format bir parça baştan savmadır.

Referans Olarak Dönen Bir Tek access() Fonksiyonu

Görülen o ki, bir güvenli diziye hem veri eklemek hem de diziden veri okumak için aynı üye fonksiyonu kullanabilirsiniz. Bunun sırrı, fonksiyondan referans yoluyla değer döndürmekte yatar. Bu şu anlama gelir: Fonksiyonu eşittir işaretinin sol tarafına yerleştirebiliriz; böylece, sağ taraftaki değer, fonksiyonun döndürdüğü değişkene atanacaktır (Bölüm 5'te anlatılmıştı). **ARROVER2** için program listesi şöyledir:

```
// arrover2.cpp
// bir guvenli dizi tanimlar
// (indeks degerleri diziye erismeden once kontrol edilir)
// put ve get'in her ikisi icin bir tek access() fonksiyonu kullanir
#include <iostream>
using namespace std;
#include <process.h>                    //exit() icin
const int LIMIT = 100;                  //dizi buyuklugu
////////////////////////////////////////////////////////////
class safearay
   {
   private:
      int arr[LIMIT];
   public:
      int& access(int n)                //dikkat:referans olarak doner
         {
         if(n<0 || n>=LIMIT)
            {cout << "\nIndex out of bounds"; exit(1);}
         return arr[n];
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   safearay sa1;

   for(int j=0; j<LIMIT; j++)           //elemanlari ekle
      sa1.access(j) = j*10;             //esittir isaretinin *sol* tarafi

   for(j=0; j<LIMIT; j++)               //elemanlari goster
      {
      int temp = sa1.access(j);         //esittir isaretinin *sag* tarafi
      cout << "Element " << j << " is " << temp << endl;
      }
   return 0;
   }
```

Aşağıdaki ifade **`j*10`** değerinin, fonksiyonun döndürdüğü değer olan **`arr[j]`** içine yerleşmesini sağlar:

```
sa1.access(j) = j*10;          //esittir isaretinin *sol* tarafi
```

sınırların içinde kalmasını garanti etmek için erişimleri kontrol ederler. Programların her birinde ki `main()`, güvenli diziyi değerlerle doldurarak (dizinin her elemanı kendi indeksinin 10 katına eşit olacak şekilde) sınıfı test ederler, sonra kullanıcıya her şeyin olması gerektiği gibi çalıştığını kanıtlamak için dizi elemanlarını ekranda gösterirler.

Ayrı Ayrı get() ve put() Fonksiyonları

Birinci program, dizi elemanlarına erişmek için iki fonksiyon sunar: `putel()`, diziye bir eleman eklemek için; `getel()`, bir dizi elemanının değerini bulmak için kullanılır. Her iki fonksiyon da, dizi sınırları dışına çıkmamasını garanti etmek için indeks numarasının değerini kontrol eder. Yani, 0'dan küçük veya dizi büyüklüğünden (dizi büyüklüğü eksi 1) büyük olup olmamasını incelerler. **ARROVER1** programının listesi şöyledir:

```
// arrover1.cpp
// guvenli bir dizi tanımlar
// (dizi elemanına erısmeden once indeks degeri kontrol edilir)
// ayri ayri put ve get fonksiyonlari kullanir
#include <iostream>
using namespace std;
#include <process.h>                    //exit() icin
const int LIMIT = 100;
////////////////////////////////////////////////////////////
class safearay
   {
   private:
      int arr[LIMIT];
   public:
      void putel(int n, int elvalue)          //elemanin degerini ayarla
         {
         if( n<0 || n>=LIMIT)
            { cout << "\nIndex out of bounds"; exit(1); }
         arr[n] = elvalue;
         }
      int getel(int n) const                  //elemanin degerini al
         {
         if( n<0 || n>=LIMIT)
            { cout << "\nIndex out of bounds"; exit(1); }
         return arr[n];
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   safearay sa1;

   for(int j=0; j<LIMIT; j++)             //elemanlari ekle
      sa1.putel(j, j*10);

   for(j=0; j<LIMIT; j++)                 //elemanlari ekranda goster
      {
      int temp = sa1.getel(j);
      cout << "Element " << j << " is " << temp << endl;
      }
   return 0;
   }
```


Veriler güvenli diziye `putel()` üye fonksiyonu ile eklenir; `getel()` fonksiyonu ile de ekranda gösterilir. Bu program, bir güvenli dizi uygulaması gerçekleştirir. Dizi sınırları dışında kalan bir indeks numarası kullanmaya kalkışırsanız hata mesajı ile karşılaşırsınız. Her şeye rağmen, kullanılan format bir parça baştan savmadır.

Referans Olarak Dönen Bir Tek access() Fonksiyonu

Görülen o ki, bir güvenli diziye hem veri eklemek hem de diziden veri okumak için aynı üye fonksiyonu kullanabilirsiniz. Bunun sırrı, fonksiyondan referans yoluyla değer döndürmektir. Bu şu anlama gelir: Fonksiyonu eşittir işaretinin sol tarafına yerleştirebiliriz; böylece, sağ taraftaki değer, fonksiyonun döndürdüğü değişkene atanacaktır (Bölüm 5'te anlatılmıştı). **ARROVER2** için program listesi şöyledir:

```
// arrover2.cpp
// bir guvenli dizi tanımlar
// (indeks degerleri diziye erisimeden once kontrol edilir)
// put ve get'in her ikisi icin bir tek access() fonksiyonu kullanir
#include <iostream>
using namespace std;
#include <process.h>                    //exit() icin
const int LIMIT = 100;                  //dizi buyuklugu
////////////////////////////////////////////////////////////
class safearay
   {
   private:
      int arr[LIMIT];
   public:
      int& access(int n)                //dikkat:referans olarak doner
         {
         if(n<0 || n>=LIMIT)
            {cout << "\nIndex out of bounds"; exit(1);}
         return arr[n];
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   safearay sa1;

   for(int j=0; j<LIMIT; j++)           //elemanlari ekle
      sa1.access(j) = j*10;             //esittir isaretinin *sol* tarafi

   for(j=0; j<LIMIT; j++)               //elemanlari goster
      {
      int temp = sa1.access(j);         //esittir isaretinin *sag* tarafi
      cout << "Element " << j << " is " << temp << endl;
      }
   return 0;
   }
```

Aşağıdaki ifade `j*10` değerinin, fonksiyonun döndürdüğü değer olan `arr[j]` içine yerleşmesini sağlar:

```
sa1.access(j) = j*10;          //esittir isaretinin *sol* tarafi
```

Bir güvenli dizinin girdi ve çıktı işlemleri için aynı fonksiyonu kullanmak, ayrı ayrı fonksiyonlar kullanmaktan belki biraz daha uygun olabilir; bu durumda, bir isim daha az hatırlamanız gerekir. Fakat, bundan daha da iyi bir yol daha vardır: Hiç isim hatırlamamak.

Referans Olarak Dönen Aşırı Yüklenmiş [] Operatörü

Normal C++ dizilerinde kullanılan indeks operatörünün ([ ]) aynını kullanarak bir güvenli diziye erişmek için indeks operatörüne **safearray** sınıfı içinde yeni bir işlev yüklenir. Ancak, bu operatör genellikle eşittir işaretinin sol tarafında kullanıldığı için aşırı yüklenmiş operatör referans yoluyla değer döndürmelidir (önceki örnekte gösterildiği gibi). **ARROVER3**'ün program listesi şöyledir:

```
// arrover3.cpp
// bir guvenli dizi tanimlar
// (indeks degerleri diziye erismeden once kontrol edilir)
// put ve get'in her ikisi icin asiri yuklenmis [] operatoru kullanilir

#include <iostream>
using namespace std;
#include <process.h>                    //exit() icin
const int LIMIT = 100;                  //dizinin buyuklugu
////////////////////////////////////////////////////////////
class safearay
   {
   private:
      int arr[LIMIT];
   public:
      int& operator [](int n)           //dikkat:referans olarak doner
         {
         if( n<0 || n>=LIMIT )
            {cout << "\nIndex out of bounds"; exit(1);}
         return arr[n];
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   safearay sa1;

   for(int j=0; j<LIMIT; j++)           //elemanlari ekle
      sa1[j] = j*10;                    //esittir isaretinin *sol* tarafi

   for(j=0; j<LIMIT; j++)               //elemanlari ekranda goster
      {
      int temp = sa1[j];                //esittir isaretinin *sag* tarafi
      cout << "Element " << j << " is " << temp << endl;
      }
   return 0;
   }
```

Bu programda güvenli dizinin girdi ve çıktıları için doğal indeks deyimleri kullanabiliriz:

```
sa1[j] = j*10;
```

ve

```
temp = sa1[j];
```


Veri Dönüşümü

= operatörünün aşağıdaki gibi ifadelerde bir değişkeni diğerine atamak amacıyla kullanıldığını zaten biliyorsunuz.

```
intvar1 = intvar2;
```

Bu ifadede **intvar1** ve **intvar2** birer tamsayı değişkendir. Ayrıca, = operatörünün, aynı tipte olmaları şartıyla kullanıcı tarafından tanımlanmış bir nesnenin değerini diğerine atadığını da fark etmişsinizdir. Şu tür ifadelerde olduğu gibi:

```
dist3 = dist1 + dist2;
```

Distance tipinde olan toplamanın sonucu, **Distance** tipinde bir başka nesne olan **dist3**'e atanır. Bir nesnenin değeri aynı tipte bir başka nesneye atandığında, normal olarak, üye veri öğelerinin tümünün değerleri yeni nesneye kopyalanır. **Distance** nesneleri gibi kullanıcının tanımladığı nesnelerin değerlerinin atanmasında, derleyicinin = operatörünü kullanmak için özel bir takım komutlar alması gerekmez.

Bu nedenle, tipler arası atamalar – söz konusu tiplerin temel veri tipi veya kullanıcı tarafından tanımlı veri tipi olması fark etmez – eşittir işaretinin her iki tarafında aynı veri tipinin kullanılması şartıyla, bizim çabamıza gerek kalmadan derleyici tarafından ele alınır. Peki, = işaretinin her iki tarafındaki değişkenler farklı tipte ise ne olur? Bu çok daha zor bir sorudur. Öyle ki, bu bölümün yarısını bu sorunun cevabına ayırdık. Öncelikle, derleyicinin temel veri tiplerini nasıl dönüştürdüğünü gözden geçireceğiz. Bu, derleyicinin otomatik olarak yaptığı bir iş. Sonra, derleyicinin otomatik olarak yapmadığı ve derleyiciye ne yapılması gerektiğini bizim bildirmemiz gereken birkaç durumu inceleyeceğiz. Temel tipler ve kullanıcı tarafından tanımlanan tipler arasındaki dönüşümler ve kullanıcı tarafından tanımlanan tipler arasındaki dönüşümler inceleyeceğimiz konular kapsamında yer alıyor.

Rutin olarak bir tipten diğerine dönüştürme işleminin zayıf programlama becerisini temsil ettiğini düşünebilirsiniz. Yine de, Pascal gibi diller sizi bu tür dönüşümler yapmaktan alıkoymak için hatırı sayılır miktarda zahmetin içine girerler. Yine de, C++'ın (ve C'nin) felsefesi şudur: Dönüşümlere izin vererek sağlanan esneklik, tehlikelerden daha ağır basar.

Temel Veri Tipleri Arasındaki Dönüşümler

Şu tür bir ifade yazdığımızda

```
intvar=floatvar;
```

intvar'ın **int** tipinde, **floatvar**'ın da **float** tipinde olduğunu düşünürsek, derleyicinin, kayan noktalı formatta ifade edilen **floatvar** değişkenini bir tamsayı formatına dönüştürmek için özel bir rutin çağıracağını farz ederiz. Böylece, **floatvar**'ın değeri **intvar**'a atanabilecektir. Elbette pek çok bu tür dönüşüm mevcuttur: **float**'tan **double**'a, **char**'dan **float**'a vs. Bu tür dönüşümlerin her birinin derleyicide standart olarak hazır bulunan kendi rutinleri vardır. Eşittir işaretinin farklı tarafındaki veri tipleri bu rutinlerin çağrılması gerektiğini bildirdiklerinde rutinler çağrılır. Bu tür dönüşümler *kapalıdır* (*implicit*), çünkü program listesinde açıkça görünmezler.

Kimi zaman derleyiciyi bir tipten diğerine dönüşüm yapması için zorlarız. Bunun için tip ataması (cast) operatörünü kullanırız. Örneğin, **float**'tan **int**'e dönüştürmek için, şöyle diyebiliriz

```
intvar = static_cast<int>(floatvar);
```

Tip atamaları *açık* (explicit) dönüşüm sağlarlar: **static_cast<int>()** ifadesinin **float**'tan **int**'e dönüşüm yapmayı planladığı program listesinde açıkça görülüyor. Her şeye rağmen, bu tür açık dönüşümler, örtülü dönüşümlerle aynı hazır rutinleri kullanırlar.

Nesneler ve Temel Veri Tipleri Arasındaki Dönüşümler

Kullanıcı tarafından tanımlanan veri tipleri ve temel veri tipleri arasında dönüşüm yapmak istediğimizde hazır dönüşüm rutinlerine güvenemeyiz. Çünkü, derleyici kullanıcının tanımladığı tipler hakkında bizim söylediklerimizden başkasını bilmez. Bunun yerine, bu rutinleri kendimiz yazmamız gerekir.

Sıradaki örneğimiz, bir temel veri tipi ile kullanıcı tarafından tanımlanan bir veri tipi arasında dönüşümün nasıl yapıldığını gösteriyor. Bu örnekte kullanıcı tarafından tanımlanan tip, önceki örneklerde gördüğümüz İngiliz uzaklık ölçülerine dayanan **Distance** sınıfı; temel veri tipi ise, metrik ölçme sisteminde bir birim uzunluğu simgeleyen metreyi temsil etmek için kullandığımız **float** tipidir.

Örnekte hem **float**'tan **Distance**'a hem de **Distance**'tan **float**'a yapılan dönüşümler yer alıyor. **ENGLCONV**'un program listesi şöyledir:

```
// englconv.cpp
// donusumler: Distance'tan metreye, metreden Distance'a
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance          //Ingiliz uzaklik sistemine dayanan Distance sinifi
   {
   private:
      const float MTF;                    //metreden feet'e
      int feet;
      float inches;
   public:                                //kurucu fonksiyon (argumansiz)
      Distance(): feet(0), inches(0.0), MTF(3.280833F)
         { }                              //kurucu fonksiyon (tek argumanli)
      Distance(float meters) : MTF(3.280833F)
         {                                //metreden Distance'a donustur
         float fltfeet = MTF * meters;    //float feet'e donustur
         feet = int(fltfeet);             //feet, tamsayi kismi
         inches = 12*(fltfeet-feet);      //inches, kalan kisim
         }                                //kurucu fonksiyon (iki argumanli)
      Distance(int ft, float in) : feet(ft),
                                   inches(in), MTF(3.280833F)
         { }
      void getdist()                      //uzakligi kullanicidan al
         {
         Cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }
```



```
      void showdist() const               //uzakligi ekranda goster
         {cout << feet << "\'-" << inches << '\"';}

      operator float() const              //donusum operatoru
         {                                //Distance'ı metreye donusturur
         float fracfeet = inches/12;      //inches'ı donustur
         fracfeet += static_cast<float>(feet);    //feet'ı ekle
         return fracfeet/MTF;             //metreye donustur
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   float mtrs;
   Distance dist1 = 2.35F;       //metreyi Distance'a donusturmek
                                 //icin 1 argumanli kurucu fonksiyon kullanir
   cout << "\ndist1 = "; dist1.showdist();

   mtrs = static_cast<float>(dist1);  // Distance'dan metreye donusturmek icin
                                      // donusum operatorunu kullanir
   cout << "\ndist1 = " << mtrs << " meters\n";

   Distance dist2(5, 10.25);          //2 argumanli kurucu fonksiyon

   mtrs = dist2;                      //yine donusum operatorunu kullaniyor
   cout << "\ndist2 = " << mtrs << " meters\n";
// dist2 = mtrs;                      //hata, = donusumu gerceklesmez
   return 0;
   }
```

main() içinde, program önce metre cinsinden uzaklığı belirten sabit bir **float** değerini tek argümanlı bir kurucu fonksiyon kullanarak feet ve inç'e çevirir:

```
Distance dist1 = 2.35F;
```

Daha sonra ters yönde ilerleyerek, bir **Distance**'ı aşağıdaki iki ifadeyi kullanarak metreye çevirir:

```
mtrs = static_cast<float>(dist2);
```

ve

```
mtrs = dist2;
```

Programın çıktısı şöyledir:

```
dist1 = 7'-8.51949"        <==== bu 2.35 metreye esit
dist1 = 2.35 meters        <==== bu 7'-8.51949"
dist2 = 1.78435 meters     <==== bu 5'-10.25"
```

Dönüşümlerin **main()** içinde basit atama ifadeleri kullanılarak nasıl yapıldığını öğrendik. Şimdi perdenin arkasında, **Distance** üye fonksiyonları içinde neler olduğuna bir göz atalım. Kullanıcının tanımladığı bir veri tipini bir temel veri tipine dönüştürmek, bir temel veri tipini kullanıcının tanımladığı bir tipe dönüştürmekten daha farklı bir yaklaşım gerektirir. Her iki dönüşümün de nasıl ele alındığını **ENGLCONV** programında göreceğiz.

Temel Veri Tipinden Kullanıcı Tanımlı Tipe Dönüşüm

Temel veri tipinden (bu örnekte, **float**) kullanıcı tanımlı veri tipine (mesela, **Distance**) geçmek için tek argümanlı bir fonksiyon kullanılır. Bunlara kimi zaman *dönüşüm kurucuları* (*conversion constructors*) deniliyor. **ENGLCONV**'daki kurucu fonksiyon işte şöyle görünür:

```
Distance(float meters)
   {
   float fltfeet = MTF * meters;
   feet = int(fltfeet);
   inches = 12 * (fltfeet-feet);
   }
```

Tek argümanlı **Distance** tipinde bir nesne oluşturulurken bu fonksiyon çağrılır. Fonksiyon bu argümanın metreyi simgelediğini farz eder. Argümanı ayak ve inç'e çevirir; elde edilen sonucu nesneye atar. Böylece, metreden **Distance**'a dönüşüm, aşağıdaki ifade kullanılarak bir nesne oluşturulmasıyla başlar ve devam eder.

```
Distance dist1 = 2.35;
```

Kullanıcı Tanımlı Tiplerden Temel Tiplere Dönüşüm

Peki, diğer türlü bir dönüşüme, yani kullanıcının tanımladığı tipten temel veri tipine doğru bir dönüşüme ne dersiniz? Buradaki incelik, *dönüşüm operatörü* (*conversion operator*) denilen bir şey oluşturmakta yatar. **ENGLCONV**'da bunu gerçekleştirdiğimiz kod parçası şöyledir:

```
operator float()
   {
   float fracfeet = inches/12;
   fracfeet += float(feet);
   return fracfeet/MTF;
   }
```

Bu operatör, kendisinin bir üyesi olduğu **Distance** nesnesinin değerini alır, bunu metreyi simgeleyen **float**'a çevirir ve bu değeri döndürür.

Bu operatör açık bir tip ataması ile:

```
mtrs = static_cast<float>(dist1);
```

ya da basit bir atama ifadesiyle de çağrılabilir:

```
mtrs = dist2;
```

Bu formların her ikisi de **Distance** nesnesini, metre cinsinden karşılık gelen **float** değerine çevirir.

C Karakter Katarları ve String Nesneleri Arasında Dönüşüm

İşte, tek argümanlı bir kurucu fonksiyon ve dönüşüm operatörü kullanan bir başka örnek daha. Bu program, bu bölümde daha önce gördüğümüz **STRPLUS** örneğindeki **String** sınıfı üzerinde işlem yapar.


```
// strconv.cpp
// siradan karakter katarlariyla String sinifi arasinda donusum
#include <iostream>
using namespace std;
#include <string.h>                //strcpy() vs icin
////////////////////////////////////////////////////////////
class String                       //kullanici tanimli karakter katari tipi
   {
   private:
      enum { SZ = 80 };            //tum String nesnelerinin buyuklugu
      char str[SZ];                //bir C karakter katari tutar
   public:
      String()                     //argumansiz kurucu fonksiyon
         { str [0] = '\0'; }
      String(char s[])             //1-argumanli kurucu fonksiyon
         { strcpy(str, s); }       //C karakter katarini String'e donustur
      void display()const          //String'i ekranda goster
         { cout << str; }
      operator char*()             //donusum operatoru
         { return str; }              //String'i C karakter katarina donustur
   };
////////////////////////////////////////////////////////////
int main()
   {
   String s1;                      //argumansiz kurucu fonksiyon kullan
                                   //C karakter katarini tanimla ve ilk deger ata
   char xstr[] = "Joyeux Noel! ";

   s1 = xstr;                      //C karakter katarindan String'e donusturmek
                                   //icin 1-argumanli kurucu fonksiyon kullan
   s1.display();                   //String'i ekranda goster

   String s2 = "Bonne Annee!";     //String'e ilk deger atamak icin
                                   //1-argumanli kurucu fonksiyon kullanir
   cout << static_cast<char*>(s2); //<< operatorune yollamadan once
   cout << endl;                      //String'i C karakter katarina donusturmek
   return 0;                          //icin donusum operatoru kullan
   }
```

Bu programdaki tek argümanlı kurucu fonksiyon, normal bir karakter katarını (**char** dizisini) **String** sınıfına ait bir nesneye dönüştürür:

```
String(char s[])
   { strcpy(str, s); }
```

s ismindeki C karakter katarı, kurucu fonksiyona argüman olarak aktarılır ve **strcpy()** kütüphane fonksiyonu kullanılarak yeni oluşturulan **String** nesnesinin **str** veri üyesine kopyalanır.

Aşağıdaki ifadedeki gibi, bir **String** tanımlandığı zaman bu dönüşüm uygulanabilir:

```
String s2="Bonne Annee!";
```

Ya da bu dönüşüm, aşağıdaki gibi atama ifadelerinde uygulanabilir:

```
s1 = xstr;
```

Burada `s1`, `String` tipindedir, `xstr` ise bir C karakter katarıdır.
Bir `String` tipini C karakter katarına dönüştürmek için bir dönüşüm operatörü kullanılır:

```
operator char*()

   {return str;}
```

Bu deyimdeki asteriks *işaret eder* anlamındadır. Bölüm 10'a kadar işaretçileri ele almayacağız; fakat, buradaki kullanımının anlaşılması zor değildir. Buradaki işaretçi, `char` tipine *işaret eder* anlamındadır; bu tanım, char *tipinde bir dizi* ifadesine çok benzer. Yani, `char*` ifadesi `char[]` ile aynıdır. Bu ifade, C karakter katarı veri tipini belirtmenin bir başka yoludur.

Derleyici tarafından kullanılan dönüşüm operatörü şu ifadede yer alır:

```
cout << static_cast<char*>(s2);
```

Burada `s2` değişkeni, aşırı yüklenmiş << operatörüne sağlanan bir argümandır. << operatörünün kendi tanımladığımız `String` tipi hakkında hiçbir bilgisi olmadığı için derleyici, `s2`'yi << operatörünün tanıyabileceği bir tipe dönüştürmenin bir yolunu arar. Dönüştürülmek istenen tip, `char*` kalıbı ile belirtilir. Böylece derleyici, `String`'den C karakter katarına bir dönüşüm gerçekleştirmenin yolunu arar, `operator char*()` fonksiyonun bulur ve bir C karakter katarı üretmek için bu fonksiyonu kullanır. Elde edilen C karakter katarı daha sonra, ekranda gösterilmek üzere << operatörüne gönderilir. (Bu, `String::display()` fonksiyonunu çağırmakla aynı etkiye sahiptir, fakat << operatörünün sağladığı kolaylığın ve anlaşılırlığın yanında `display()` fonksiyonu gereksiz kalır. Bu nedenle programdan çıkartılabilir.)

`STRCONV` programının çıktısı şöyledir:

```
Joyeux Noel! Bonne Anneel
```

`STRCONV` örneği, dönüşümlerin sadece atama ifadelerinde değil, uygun olan diğer yerlerde, mesela operatörlere (<< operatörü gibi) gönderilen argümanlarda veya fonksiyonlarda da otomatik olarak kullanıldığını gösterir. Eğer bir operatöre veya fonksiyona yanlış tipte argümanlar sağlarsanız, böyle bir dönüşüm tanımlamış olmanız şartıyla, bu argümanlar kabul edilebilir tipte argümanlara dönüştürülecektir.

Bir `String`'i bir C karakter katarına dönüştürmek için açık atama ifadeleri kullanmayacağınıza dikkat etmelisiniz:

```
xstr = s2;
```

Bir C karakter katarı olan `xstr` aslında bir dizidir ve dizilere normal şartlarda değer atayamazsınız (gerçi Bölüm 11'de göreceğimiz gibi, atama operatörünü aşırı yüklediğiniz takdirde her türlü ihtimal mümkün hale gelir).

Farklı Sınıfların Nesneleri Arasındaki Dönüşümler

Peki, kullanıcı tarafından tanımlanan farklı sınıfların nesneleri arasında bir dönüşüm mümkün mü? Temel veri tipleri ve kullanıcının tanımladığı tipler arasındaki dönüşümler için az önce gösterilen iki metot, kullanıcı tarafından tanımlanan iki tip arasındaki dönüşümlere de uygulanabilir. Yani, tek argümanlı bir kurucu fonksiyon ya da dönüşüm operatörü kullanabilirsiniz. Tercih, dönüşüm rutinini kaynak veya hedef nesnenin sınıf deklarasyonlarından hangisinin içine koymak istediğinize bağlıdır. Diyelim ki, şöyle bir ifade kullanıyorsunuz:


```
objecta = objectb;
```

Burada `objecta`, `A` sınıfının bir üyesi; `objectb` ise `B` sınıfının bir üyesi olsun. Dönüşüm rutini `A` (`objecta` değer aldığı için bu, hedef sınıftır) sınıfında mı, yoksa `B` (kaynak sınıf) sınıfında mı yer alıyor? Her iki durumu da inceleyeceğiz.

Zamanın İki Türü

Örnek programlarımız zamanı ölçmenin iki yöntemi arasında dönüşüm gerçekleştirecektir: 12-saat bazında ve 24-saat bazında. Zamanı bildirmeye yarayan bu yöntemler bazen *sivil* ve *askeri* zaman olarak adlandırılırlar. `time12` sınıfımız, dijital saatlerde ve hava alanlarındaki uçak kalkış göstergelerinde kullanılan sivil zamanı temsil ediyor. Bu bağlamda saniyeye gerek olmadığını varsayacağız; bu nedenle, `time12` sadece saat (1'den 12'ye kadar), dakika ve "a.m" veya "p.m" göstergelerini kullanır. `time24` sınıfımız ise havacılıkta ve denizcilikte yön tayini gibi çok daha hassas uygulamalar için saat (00'dan 23'e kadar), dakika ve saniyeyi kullanır. Tablo 8.1'de her ikisi arasındaki farklar gösteriliyor.

TABLO 8.1: 12-Saat ve 24-Saat Bazında Zaman

12-Saat Bazında Zaman	24-Saat Bazında Zaman
12:00 a.m. (gece yarısı)	00:00
12:01 a.m.	00:01
1:00 a.m.	01:00
6:00 a.m.	06:00
11:59 a.m	11:59
12:00 p.m. (öğle)	12:00
12:01 p.m.	12:01
6:00 p.m.	18:00
11:59 p.m.	23:59

Dikkat ederseniz, sivil zamanda 12 a.m (gece yarısı) askeri zamanda 00 saate karşılık geliyor. Sivil zamanda ise 0 saat diye bir ölçü yok.

Kaynak Nesne İçindeki Rutin

İlk örnek programımız kaynak sınıf içinde yer alan bir dönüşüm rutinini gösterir. Dönüşüm rutini, kaynak sınıf içinde yer alınca genellikle dönüşüm operatörü olarak uygulanır. `TIMES1`'in program listesi şöyledir:

```
// times1.cpp
// time24 icindeki bir operatoru kullanarak time24'ten time12'ye donusturur
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////
class time12
   {
   private:
      bool pm;                          //true = pm, false = am
```

```
      int hrs;                          //1'den 12'ye
      int mins;                         //0'dan 59'a
                                        //argumansiz kurucu fonksiyon
   public:
      time12() : pm(true), hrs(0), mins(0)
         { }
                                        //3-argumanli kurucu fonksiyon
      time12(bool ap, int h, int m) : pm(ap), hrs(h), mins(m)
         { }
      void display() const              //bicim:11:59 p.m.
         {
         cout << hrs << ':';
         if(mins < 10)
            cout << '0';                //"01" icin fazladan 0
         cout << mins << ' ';
         string am_pm = pm ? "p.m." : "a.m.";
         cout << am_pm;
         }
   };
////////////////////////////////////////////////////////////
class time24
   {
   private:
      int hours;                        //0'dan 23'e
      int minutes;                      //0'dan 59'a
      int seconds;                      //0'dan 59'a
   public:                              //argumansiz kurucu fonksiyon
      time24() : hours(0), minutes(0), seconds(0)
         { }
      time24(int h, int m, int s):      //3-argumanli kurucu fonksiyon
            hours(h), minutes(m), seconds(s)
         { }
      void display() const              //bicim:23:15:01
         {
         if(hours < 10)    cout << '0';
         cout << hours << ':';
         if(minutes < 10)  cout << '0';
         cout << minutes << ':';
         if(seconds < 10)  cout << '0';
         cout << seconds;
         }
      operator time12() const;          //donusum operatoru
   };
//--------------------------------------------------------
time24::operator time12() const         //donusum operatoru
   {
   int hrs24 = hours;
   bool pm = hours < 12 ? false : true;    // am/pm, hangisi bul
                                           //saniyeyi yuvarla
   int roundMins = seconds < 30 ? minutes : minutes+1;
   if(roundMins == 60)                     //dakikada elde var mi?
      {
      roundMins=0;
      ++hrs24;
      if(hrs24 == 12 || hrs24 == 24)       //saatte elde var mi?
         pm = (pm == true) ? false : true; //am/pm, digerine cevir
      }
   int hrs12 = (hrs24 < 13) ? hrs24 : hrs24-12;
   if(hrs12==0)
      { hrs12=12; pm=false; }               //00= 12 a.m.
```



```
   return time12(pm, hrs12, roundMins);
   }
////////////////////////////////////////////////////////////
int main()
   {
   int h, m, s;

   while(true)
      {                                 //kullanicidan 24-saat formatinda zamani al
      cout << "Enter 24-hour time: \n";
      cout << " Hours (0 to 23): "; cin >> h;
      if(h > 23)                        //eger saat > 23 ise cik
         return(1);
      cout << " Minutes: "; cin >> m;
      cout << " Seconds: "; cin >> s;

      time24 t24(h, m, s);              //bir time24 hazirla
      cout << "You entered: ";          //time24'u ekranda goster
      t24.display();

      time12 t12 = t24;                 //time24'u time12'ye cevir

      cout << "\n12-hour time: ";        //karsilik gelen time12'yi ekranda goster
      t12.display();
      cout << "\n\n";
      }
   return 0;
   }
```

`TIMES1`'in `main()` bölümünde `time24` tipinde ve `t24` olarak adlandırılan bir nesne tanımlanır. Kullanıcıdan alınan saat, dakika ve saniye değerleri `t24`'e atanır. Bir de, `time12` tipinde ve `t12` olarak adlandırılan bir nesne daha tanımlanır. Bu nesneye aşağıdaki ifade yardımıyla başlangıç değeri olarak `t24` atanır:

```
time12 t12 = t24;
```

Bu nesneler farklı sınıflardan olduğu için değer atama işlemi bir dönüşüm içerir. Önceden belirttiğimiz gibi, bu programda dönüşüm operatörü `time24` sınıfının bir üyesidir. Deklaratör şöyle ifade edilir:

```
time24::operator time12() const         //donusum operatoru
}
```

Bu fonksiyon, üyesi olduğu nesneyi bir `time12` nesnesine dönüştürür ve bu nesneyi döndürür. Daha sonra `main()`, bu nesneyi `t12`'ye atar. `TIMES1`'le etkileşim şöyle olabilir:

```
Enter 24-hour time:
Hours (0 to 23): 17
Minutes: 59
Seconds: 45
You entered: 17:59:45
12-hour time: 6:00 p.m.
```

Saniye değeri yuvarlanır; bu nedenle, 12-saat bazında ifade edilen zaman 5:59 p.m.'den 6:00 p.m.'e yükseltilir. 23'ten daha büyük bir saat değerinin girilmesi programdan çıkmaya neden olur.

Hedef Nesne İçindeki Rutin

Dönüşüm rutini hedef sınıfın içinde yer aldığı zaman yeni dönüşümün nasıl yürütüldüğüne bir bakalım. Böyle bir durumda tek argümanlı bir kurucu fonksiyon kullanmak yaygın olan yaklaşımdır. Bununla birlikte, hedef sınıf içindeki kurucu fonksiyonun dönüşümü gerçekleştirmek için kaynak sınıf içindeki verilere erişebilir olması gerçeği işleri güçleştirir. `time24` için deki veriler – `hours`, `minutes` ve `seconds` – private'tır; bu nedenle, bu verilere doğrudan erişim sağlamak için `time24` içinde private üye fonksiyonlar temin etmemiz gerekir. Bu fonksiyonlar `getHrs()`, `getMins()` ve `getSecs()` olarak adlandırılır.

```
// times2.cpp
// time12'de bir kurucu fonksiyon kullanarak time24'ten time12'ye donustur
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class time24
   {
   private:
      int hours;                          //0 - 23
      int minutes;                        //0 - 59
      int seconds;                        //0 - 59
   public:                                //argumansiz kurucu fonksiyon
      time24(): hours(0), minutes(0), seconds(0)
         { }
      time24(int h, int m, int s) :       //3-argumanli kurucu fonksiyon
                hours(h), minutes(m), seconds(s)
         { }
      void display() const                //format 23:15:01
         {
         if(hours < 10)    cout << '0';
         cout << hours << ':';
         if(minutes < 10)  cout << '0';
         cout << minutes << ':';
         if(seconds < 10)  cout << '0';
         cout << seconds;
         }
      int getHrs() const      { return hours; }
      int getMins() const     { return minutes; }
      int getSecs() const     { return seconds; }
   };
////////////////////////////////////////////////////////////////
class time12
   {
   private:
      bool pm;                            //true=pm,false=am
      int hrs;                            //1 - 12
      int mins;                           //0 - 59
   public:                                //argumansiz kurucu fonksiyon
      time12() : pm(true), hrs(0), mins(0)
         { }
      time12(time24);                     //1-argumanli kurucu fonksiyon
```



```
                                       //3-argumanli kurucu fonksiyon
      time12(bool ap, int h, int m) : pm(ap), hrs(h), mins(m)
         { }
      void display() const
         {
         cout << hrs << ':';
         if(mins < 10) cout << '0';       // "01 " icin fazladan 0
         cout << mins << ' ';
         string am_pm = pm ? "p.m." : "a.m.";
         cout << am_pm;
         }
   };
//--------------------------------------------------------------
time12::time12(time24 t24)             //1-argumanli kurucu fonksiyon
   {                                   //time24'u time12'ye donusturur
   int hrs24 = t24.getHrs();           //saati al
                                       //am/pm, hangisi bul
   pm = t24.getHrs() < 12 ? false : true;

   mins = (t24.getSecs() < 30) ?       //saniyeyi yuvarla
                  t24.getMins() : t24.getMins()+1;
   if(mins == 60)                      //dakikada elde var mi?
      {
      mins=0;
      ++hrs24;
      if(hrs24 == 12 || hrs24 == 24)   //saatte elde var mi?
         pm = (pm==true) ? false : true;  //am/pm, digerine cevir
      }
   hrs = (hrs24 < 13) ? hrs24 : hrs24-12; //hrs'i donustur
   if(hrs==0)                          //00= 12 a.m.
      { hrs=12; pm=false; }
   }
////////////////////////////////////////////////////////////////
int main()
   {
   int h, m, s;

   while(true)
      {                          //kullanicidan 24-saat formatinda zamani al
      cout << "Enter 24-hour time: \n";
      cout << "   Hours (0 to 23): "; cin >> h;
      if(h > 23)                          //eger saat > 23 ise cik
         return(1);
      cout << "   Minutes: "; cin >> m;
      cout << "   Seconds: "; cin >> s;

      time24 t24(h, m, s);                //bir time24 hazirla
      cout << "You entered: ";            //time24'u ekranda goster
      t24.display();

      time12 t12 = t24;                   //time24'ten time12'ye donustur

      cout << "\n12-hour time: ";          //karsilik gelen time12'yi goster
      t12.display();
      cout << "\n\n";
      }
   return 0;
   }
```

Dönüşüm rutini `time12` sınıfında yer alan tek argümanlı bir kurucu fonksiyondur. Bu fonksiyon, argüman olarak aldığı nesnenin `time24` değerlerine karşılık gelen değerlerini, üyesi olduğu nesneye yükler. Bu, `TIMES1`'deki dönüşüm operatörü ile çok benzer biçimde çalışır; sadece bu fonksiyon, `time24` nesnesi içindeki verilere erişmek için `getHrs()` ve benzeri fonksiyonları kullanarak biraz daha zor bir işin altına girer.

`TIMES2`'nin `main()` bölümü `TIMES1`'inki ile aynıdır. Tek argümanlı kurucu fonksiyon yine aşağıdaki ifade ile, `time24`'den `time12`'ye dönüşümün gerçekleştirilmesine imkan verir:

```
time12 t12 = t24;
```

Ayrıca, çıktı da aynıdır. İkisi arasındaki fark, perdenin arkasında gizlidir: Bu kez dönüşüm, kaynak nesne yerine hedef nesne içinde yer alan dönüşüm operatörü tarafından gerçekleştirilir.

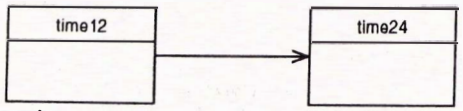
Dönüşümler: Neyi Ne Zaman Kullanmak

Kaynak sınıf içindeki dönüşüm operatörüne karşı, hedef sınıf içindeki tek argümanlı kurucu fonksiyonu ne zaman kullanmalısınız? Çoğunlukla tercihinizi kendiniz yapabilirsiniz. Yine de, kimi zaman sizin için tercih yapılmıştır. Eğer bir kütüphane sınıfı satın aldıysanız bunların kaynak koduna erişemeyebilirsiniz. Dönüşümde kaynak olarak böyle bir sınıfın nesnesini kullanıyorsanız sadece hedef sınıfa erişim hakkına sahip olacaksınız; bu nedenle, tek argümanlı bir kurucu fonksiyon kullanmaya ihtiyacınız olacaktır. Şayet kütüphane sınıfına ait nesne hedefte ise, dönüşüm operatörünü kaynakta kullanmalısınız.

UML Sınıf Şemaları

UML'i Bölüm 1'de "Genel Görünüm" başlığı altında tanıtmıştık. Artık sınıflar hakkında biraz bir şeyler öğrendiğinize göre ilk UML özelliğimize bir göz atabiliriz: *Sınıf şeması*. Bu şema, nesne yönelimli programlara yeni bir bakış açısı sunar. Ayrıca, `TIMES1` ve `TIMES2` programlarının çalışmalarına ilave bir ışık da tutabilir.

`TIMES1`'in program listesine baktığımızda programda iki sınıf olduğunu görüyoruz: `time12` ve `time24`. Bir UML sınıf şemasında sınıflar dikdörtgenlerle simgelenir. Şekil 8.3'te bu gösteriliyor.



ŞEKİL 8.3: `TIMES1` programının UML sınıf şeması.

Sınıf dikdörtgenlerinin her biri yatay çizgilerle bölümlere ayrılır. Sınıfın ismi en üstteki bölüme yerleştirilir. Burada göstermiyoruz ama, şemaya üye veriler (UML'de bunlara *nitelikler* – *attributes* – denir) ve üye fonksiyonlar (bunlar *işlemler* – operations – olarak adlandırılır) için de bölümler dahil edebilirsiniz.

Bağlantılar

Sınıfların kendi aralarında çok çeşitli ilişkileri olabilir. `TIMES`'deki sınıflar *bağlantı* (*association*) yoluyla ilişkilendirilirler. Bu ilişkiyi, bu sınıfların dikdörtgenlerini birleştiren bir çizgi ile belirtiyoruz. (Sınıf ilişkilerinin bir başka türü olan genelleştirmenin nasıl bir özellik olduğunu Bölüm 9'da "Kalıtım" bahsinde göreceğiz.)

Bir bağlantıyı meydana getiren nedir? Kavramsal olarak, program içinde sınıf olarak simgelenen gerçek dünyaya ait varlıklar bir çeşit belirgin bir ilişki içindedirler. Sürücüler arabalarla, kitaplar kütüphane görevlileriyle, yarış atları yarış kulvarlarıyla ilişkilidir. Eğer bu tür varlıklar bir program içinde yer alsalardı bağlantı yoluyla ilişkilendirilirlerdi.

`TIMES2.CPP` programında `time12` sınıfının `time24` ile ilişkili olduğunu anlayabiliyoruz, çünkü bir sınıfın nesnelerini diğerinin nesnelerine dönüştürüyoruz.

Bir sınıf bağlantısı, sınıfların kendilerinden ziyade aslında sınıfların nesnelerinin bir çeşit ilişki içinde olduğunu gösterir. Tipik olarak, eğer bir sınıfın bir nesnesi diğer sınıfın bir nesnesine ait bir üye fonksiyonu (bir *işlemi*) çağırırsa iki sınıf birbiriyle ilişkilidir. Ayrıca, bir sınıfın bir niteliği, diğer bir sınıfın nesnesi ise yine bağlantıdan söz edilebilir.

`TIMES1` programında, `time12` sınıfının `t12` olarak adlandırılan bir nesnesi, `time24` sınıfının `t24` isimli nesnesinde bulunan `operator time12()` dönüşüm rutinini çağırır. Bu, `main()`'de şu ifade ile gerçekleştirilir:

```
time12 t12 = t24;          //time24'ten time12'ye donustur
```

Böyle bir çağrı, iki sınıf arasında bir bağlantı çizgisi ile simgelenir.

Yönlendirilebilirlik

Bağlantının yönünü ya da *yönlendirilebilirliğini* (*navigability*) belirtmek için açık ok ucu ekleyebiliriz. (Sonradan ele alacağımız gibi, kapalı ok uçlarının farklı anlamı vardır.) `time12`, `time24`'ü çağırdığı için ok, `time12`'den `time24`'e, `time24`'ü gösterecek şekildedir. Buna *tek yönlü* bağlantı denir, çünkü ok sadece tek bir yönü gösterir. Eğer sınıfların her ikisi de bir diğerindeki işlemi çağıracak olsaydı, çizginin her iki ucunda da ok ucu olurdu ve böyle bir bağlantı *çift yönlü* bağlantı olarak adlandırılırdı. UML'deki diğer birçok şey gibi yönlendirilebilirlik okları da isteğe bağlıdır.

Operatörleri Aşırı Yüklemenin ve Dönüşümlerin Zorlukları

Operatörlerin aşırı yüklenmesi ve tip dönüşümleri, bütünüyle yeni bir dil oluşturmak için fırsat verir. `a`, `b` ve `c` kullanıcı tarafından tanımlanan sınıfların birer nesnesi iken ve `+` operatörünün işlevi artırıldığında, şu ifade, `a`, `b` ve `c` temel veri tipinde değişkenler iken elde edilen anlamdan oldukça farklı bir anlam içerebilir:

```
a = b + c;
```

Dilin yapı taşlarını yeniden tanımlama becerisi bir lütuf olarak düşünülebilir, çünkü bu beceri program listelerinizin çok daha anlaşılır ve okunaklı olmasını sağlayabilir. Fakat, bu becerinin ters etkisi de olabilir; program listeleriniz çok daha karışık ve anlaşılması güç hale gelebilir. İşte konunun ana noktalarından bazıları.

Benzer Anlamlar Kullanın

Aşırı yüklenmiş operatörleri, temel veri tipleri üzerinde gerçekleştirilen işlemlere mümkün olduğunca benzeyen işlemleri gerçekleştirmek için kullanın. + işaretini örneğin, çıkarma yapacak şekilde aşırı yükleyebilirsiniz; fakat bu, program listenizin daha anlaşılır olmasını zorlaştıracaktır.

Bir operatörün aşırı yüklenmesi, belirli bir sınıfın nesneleri üzerinde belirli işlemler gerçekleştirmenin bir manası olduğunu varsayar. Eğer `X` sınıfındaki + operatörünü aşırı yükleyeceksek, `X` sınıfından iki nesnenin toplanmasıyla elde edilen sonuç, en azından toplamayı biraz andırır bir anlam içermelidir. Örneğin, bu bölümde İngiliz uzaklık ölçülerine dayanan `Distance` sınıfı için + operatörünün nasıl aşırı yüklendiğini gördük. İki uzaklığı toplamak elbette anlamlıdır. `String` sınıfı için de ayrıca, + operatörünü aşırı yükledik. Burada iki karakter katarının toplanmasını, bir karakter katarını diğerinin sonuna ekleyerek bir üçüncü karakter katarı oluşturmak olarak yorumladık. Bu da anlaşılırlık açısından tatmin edici bir yorumdur. Fakat, bir çok sınıf için nesnelerinin "toplanması"ndan bahsetmek mantıklı olmayabilir. Örneğin, personel verilerini tutan `employee` adında bir sınıfın nesnelerini muhtemelen toplamak istemeyeceksiniz.

Benzer Söz Dizimi Kullanın

Aşırı yüklenmiş operatörleri, temel veri tiplerini kullandığınız şekilde kullanın. Örneğin, eğer `alpha` ve `beta` temel veri tipinde ise, şu ifadedeki atama operatörü, `alpha`'ya `alpha` ve `beta`'nın toplamını yükler.

```
alpha += beta;
```

Bu operatörün, aşırı yüklenmiş herhangi bir versiyonu buna benzer bir iş gerçekleştirmelidir. Muhtemelen aynı işlemi şu şekilde de yapar:

```
alpha = alpha + beta;
```

Burada + aşırı yüklenmiştir.

Eğer bir aritmetik operatörünü aşırı yüklediyseniz, tutarlılık açısından tümünü aşırı yüklemeyi isteyebilirsiniz. Bu karışıklığı önleyecektir.

Operatörlerin bazı sözdizimsel özellikleri değiştirilemeyebilir. Tespit etmiş olabileceğiniz gibi, ikili operatörü, tekli operatör olacak şekilde aşırı yükleyemezsiniz. Bunun tersi de geçerlidir.

Kendinize Hakim Olun

Şunu hatırınızdan çıkarmayın: Eğer + operatörünün işlevini artırmışsanız, program listenize yabancı birinin aşağıdaki gibi bir ifadenin gerçekten ne anlama geldiğini bulmak için hatırı sayılır bir araştırma yapması gerekecektir:

```
a = b + c;
```

Eğer aşırı yüklenmiş operatörlerin sayısı çok artmışsa ve operatörler anlaşılmaz şekilde kullanılıyorsa, bunları kullanmanın tüm mantığı kaybolmuş demektir; listeyi okumak kolaylaşacağına, daha da güç hale gelir. Aşırı yüklenmiş operatörleri tutumlu kullanın ve sadece kullanımı çok belli olduğunda kullanın. Şüpheli durumlarda, aşırı yüklenmiş bir operatör kullanmak yerine bir fonksiyon kullanın; çünkü, fonksiyonun ismi amacını belirtebilir. Eğer bir karakter katarının sol tarafını bulmak için bir fonksiyon yazıyorsanız, mesela, bir operatörü (diyelim ki `&&`) aşırı yüklemeye çalışmaktansa `getleft()` diye bir fonksiyon çağırmak çok daha iyidir.

Belirsizlikleri Önleyin

Diyelim ki, aynı dönüşümü gerçekleştirmek için (örneğin, `time24`'ten `time12`'ye) hem tek argümanlı bir kurucu fonksiyon hem de bir dönüşüm operatörü kullanıyorsunuz. Derleyici hangi dönüşümü kullanması gerektiğini nasıl bilecek? Bilemez. Derleyici ne yapması gerektiğini bilmediği bir durumda kalmaktan hoşlanmaz ve bir hata mesajı verir. Yani, aynı dönüşümü birden fazla yoldan yapmayı önleyin.

Her Operatörü Aşırı Yüklenemeyebilir

Şu operatörler aşırı yüklenemez: Üye erişim veya nokta operatörü (`.`), kapsam çözünürlük operatörü (`::`) ve koşul operatörü (`?:`). Ayrıca, henüz karşılaşmadığımız, üyelere işaret eden (`->`) operatör de aşırı yüklenemez. Şayet merak ediyorsanız, hayır, yeni (mesela, `*&`) operatörler oluşturup, onları aşırı yüklemeye kalkışamazsınız; sadece mevcut operatörler aşırı yüklenebilir.

explicit ve mutable Anahtar Kelimeleri

İki acayip anahtar kelimeye bir bakalım: `explicit` ve `mutable`. Bunlar oldukça farklı bir etkiye sahiptirler, fakat her ikisi de sınıfın üyeleri üzerinde değişiklik yaptıkları için burada birlikte gruplanmışlardır. `explicit` anahtar kelimesi veri dönüşümü ile ilgili, ama `mutable`'ın daha güç algılanan bir amacı vardır.

explicit Sayesinde Dönüşümleri Engellemek

İyi bir iş olarak karar verdiğiniz bazı spesifik dönüşümler olabilir. Bunları gerçekleştirmek için gerekli adımları atarsınız: `TIME1` ve `TIME2` örneklerinde gösterildiği gibi, uygun dönüşüm operatörlerini ve tek argümanlı kurucu fonksiyonları kurarsınız. Bununla birlikte, gerçekleştirilmesini istemediğiniz diğer dönüşümler de olabilir. İstemediğiniz herhangi bir dönüşümü aktif olarak engelleyebilirsiniz. Böylece tatsız sürprizleri önlemiş olursunuz.

Dönüşüm operatörü ile gerçekleştirilen bir dönüşümü önlemek kolaydır: Bu operatörü tanımlamayın. Fakat, kurucu fonksiyonlarla işler bu kadar kolay değildir. Bir başka tipteki tek bir değeri kullanan nesneler tanımlamak isteyebilirsiniz, fakat tek argümanlı bir kurucu fonksiyonun diğer durumlarda mümkün kıldığı kapalı dönüşümleri istemeyebilirsiniz. Bu durumda ne yapılabilir?

Standart C++ bu problemi çözmek için `explicit` adında bir anahtar kelime içerir. Bu, tek argümanlı kurucu fonksiyonun hemen önüne yerleştirilir. `EXPLICIT` örnek programı (`ENGLCON` programına dayanır) bunun nasıl yapıldığını gösterir:

```
// explicit.cpp
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance    //Ingiliz uzaklik olculerine dayanan Distance sinifi
   {
```

```
   private:
      const float MTF;                    //metreden feet'e
      int feet;
      float inches;                       //argumansiz kurucu fonksiyon
   public:
      Distance(): feet(0), inches(0.0), MTF(3.280833F)
         { }                              //EXPLICIT tek argumanli kurucu fonksiyon
      explicit Distance(float meters) : MTF(3.280833F)
         {
         float fltfeet = MTF * meters;
         feet = int(fltfeet);
         inches = 12*(fltfeet-feet);
         }
      void showdist()                     //uzakligi ekranda goster
         {cout << feet << "\'-" << inches << '\"';}
   };
////////////////////////////////////////////////////////////
int main()
   {
   void fancyDist(Distance);              //deklarasyon
   Distance dist1(2.35F);                 //metreyi Distnace'a donusturmek icin
                                          //1 argumanli kurucu fonksiyon kullanir
// Distance dist1 = 2.35F;                //HATA, eger kurucu fonksiyon explicit ise
   cout << "\ndist1= "; dist1.showdist();

   float mtrs = 3.0F;
   cout << "\ndist1 ";
// fancyDist(mtrs);                       //HATA, eger kurucu fonksiyon explicit ise
   return 0;
   }
//-----------------------------------------------------------
void fancyDist(Distance d)
   {
   cout << "(in feet and inches) = ";
   d.showdist();
   cout << endl;
   }
```

Bu program, her ayak ve inç rakamının önüne "(in feet and inches)" ifadesini ekleyerek **Distance** nesnesinin çıktısını güzelleştiren, **fancyDist()** adında bir fonksiyon içerir. Bu fonksiyonun argümanı bir **Distance** değişkenidir. Bu tür bir değişkenle **fancyDist()**'i problemsiz olarak çağırabilirsiniz.

Buradaki tuzak şudur: Eğer önlemini almadıysanız **fancyDist()**'i **float** tipinde bir değişkeni argüman olarak kullanarak da çağırabilirsiniz:

```
fancyDist(mtrs);
```

Derleyici bunun hatalı bir tip olduğunun farkına varıp bir dönüşüm operatörü arayacaktır. **float** tipini argüman olarak alan bir **Distance** kurucu fonksiyonu bulunca, bu kurucu fonksiyonun **float**'tan **Distance**'a dönüşümü gerçekleştirmesini ayarlayacak ve **Distance** değerini yukarıdaki fonksiyona aktaracaktır. Bu bir *kapalı* (*implicit*) dönüşümdür. Bu, belki de gerçekleşmesini düşünmediğiniz bir dönüşümdür.

Bunun yanı sıra, eğer kurucu fonksiyonu *açık* (explicit) yaparsak kapalı dönüşümleri önlemiş oluruz. Programda **fancyDist()** çağrısının önündeki açıklama simgesini kaldırarak bu

deneyebilirsiniz: Derleyici size dönüşümü gerçekleştiremeyeceğini bildirecektir. **explicit** anahtar kelimesi olmasa bu çağrı tamamen kurallara uygundur.

explicit kurucu fonksiyonunun bir yan etkisi olarak, eşittir işareti kullanarak bir nesneyi ilk kullanıma hazırlama işlemini burada gerçekleştiremeyeceğinize dikkat edin:

```
Distance dist1 = 2.35F;
```

Yalnız, parantezli ifade her zaman olduğu gibi bu durumda da çalışır.

```
Distance dist1(2.35F);
```

mutable Kullanarak const Nesne Verilerini Değiştirmek

Her zamanki gibi, bir **const** nesne tanımladığınız zaman (Bölüm 6'da anlatıldı), bu nesnenin üye verilerinin hiçbirinin değiştirilemeyeceğini garanti etmek istersiniz. Bununla birlikte, **const** nesneler tanımlanırken arada sırada şöyle bir durum ortaya çıkar: **const** nesneler tanımlamak istersiniz; fakat, aynı zamanda, nesnenin **const** olmasına rağmen birkaç spesifik veri üyesini de değiştirmeniz gerekiyordur.

Örnek olarak, bir pencere hayal edelim (Windows programlarının ekrana çizdiği türden bir pencere). Pencerenin bazı özellikleri, mesela kaydırma çubukları ve menüler, pencere tarafından *sahiplenilmiş* olabilir. Sahiplenme, çeşitli programlama problemlerinde yaygın olarak kullanılır. Bir nesnenin bir diğerinin bir niteliği olduğu duruma nazaran sahiplenme, daha yüksek derecede bir bağımsızlığa işaret eder. Böyle bir durumda bir nesne değişmeden kalabilir; yalnızca sahibi değişir. Kaydırma çubuğu aynı boyut, renk ve yön bilgilerine sahip olabilir; fakat, mülkiyeti bir pencereden diğerine aktarılabilir. Bu, bankanızın sizin ipoteğinizi bir başka bankaya satmasına benzer. İpoteğin tüm şartları aynıdır ama sahibi farklıdır.

Diyelim ki, **const** kaydırma çubuklarının niteliklerinin değişmeden kaldığı, sadece sahipliğinin değiştiği bir tanımın mümkün olmasını istiyoruz. **mutable** anahtar kelimesi işte böyle bir anda ortaya çıkar. **MUTABLE** isimli program bunun nasıl kullanılabileceğini gösterir.

```
// mutable.cpp
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////
class scrollbar
   {
   private:
      int size;                        //const olmasiyla ilgili
      mutable string owner;            //const olmasiyla ilgili degil
   public:
      scrollbar(int sz, string own) : size(sz), owner(own)
         { }
      void setSize(int sz)             //boyutu degistirir
         { size = sz; }
      void setOwner(string own) const  //sahibi degistirir
         { owner = own; }
      int getSize() const              //boyutu dondurur
         { return size; }
      string getOwner() const          //sahibi dondurur
         { return owner; }
   };
```

```
//////////////////////////////////////////////////////////
int main()
   {
   const scrollbar sbar(60, "Window1");

// sbar.setSize(100);                    //bunu sabit bir nesneye yapamazsiniz
   sbar.setOwner("Window2");             //bu, tamam
                                         //bunlar da tamam
   cout << sbar.getSize() << ", " << sbar.getOwner() << endl;
   return 0;
   }
```

size olarak adlandırılan nitelik, kaydırma çubuğu verilerinin **const** nesneler içinde değiştirilemeyeceğini simgeler. Fakat, **owner** adındaki nitelik, nesne **const** bile olsa değiştirilebilir. Buna izin vermek için **owner** niteliği **mutable** anahtar kelimesi ile kullanılır. **main()** içinde, **sbar** adında sabit bir nesne tanımlanır. Bu nesnenin boyutu değiştirilemez; fakat, sahibi **setOwner()** fonksiyonu kullanılarak değiştirilebilir. (Sabit olmayan nesnelerde elbette her iki nitelik de değiştirilebilir.) Bu durum için, **sbar** *mantıksal* **const** olma durumuna sahip, denilir. Bunun anlamı, **sbar** teorik olarak değiştirilemez, pratik olarak ise kısıtlı ölçüde değiştirilebilir demektir.

Özet

Bu bölümde, normal C++ operatörlerinin kullanıcı tarafından tanımlanan veri tiplerine uygulandıklarında nasıl yeni anlamlar yüklenebileceklerini gördük. Bir operatörü aşırı yüklemek için **operator** anahtar kelimesi kullanılır; aşırı yüklenmiş operatör programcı tarafından sağlanan anlamı kabul eder.

Operatörleri aşırı yüklemekle yakından alakalı bir başka konu ise *tip dönüşümleridir*. Bazı dönüşümler kullanıcının tanımladığı tipler ve temel veri tipleri arasında gerçekleşir. Bu tür dönüşümlerde iki yaklaşım kullanılır: Tek argümanlı bir kurucu fonksiyon, bir temel veri tipini kullanıcının tanımladığı tipe çevirir; bir dönüşüm operatörü ise kullanıcının tanımladığı tipi bir temel veri tipine dönüştürür. Kullanıcı tarafından tanımlanan bir tip yine bu tür bir tipe dönüştürülecekse iki yaklaşımdan herhangi biri kullanılabilir.

TABLO 8.2: Tip Dönüşümleri

	Hedef Nesne İçindeki Rutin	Kaynak Nesne İçindeki Rutin
Temel veriden Temel veriye	*(Standart olarak hazır bulunan dönüşüm operatörleri)*	
Temel veriden Sınıfa	Kurucu fonksiyon	yok
Sınıftan Temel veriye	yok	Dönüşüm operatörü
Sınıftan Sınıfa	Kurucu fonksiyon	Dönüşüm operatörü

explicit anahtar kelimesi ile tanımlanan bir kurucu fonksiyon, kapalı veri dönüşümlerinde kullanılamaz. **mutable** anahtar kelimesi ile tanımlanan bir veri üyesi, const bir nesne olsa bile değiştirilebilir.

UML sınıf şemaları, sınıfları ve sınıflar arası ilişkileri gösterir. Bir "bağlantı" gerçek dünyaya ait nesnelerle programın sınıflarının simgeledikleri arasındaki kavramsal bir ilişkiyi temsil eder. Bağlantılar bir sınıftan diğerine yönlendirilebilir; buna yönlendirilebilirlik (navigability) denir.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Operatörleri aşırı yüklemek,
   a. C++ operatörlerinin nesnelerle çalışmasına imkan verir.
   b. C++ operatörlerine kaldırabileceklerinden fazlasını yükler.
   c. mevcut C++ operatörlerine yeni anlamlar verir.
   d. yeni C++ operatörleri hazırlar.
2. **x** isimli bir sınıfın aşırı yüklenmiş operatör kullanmadığını varsayarak, **x** sınıfının bir nesnesi olan **x1**'i yine aynı sınıfın bir nesnesi olan **x2**'den çıkaran ve sonucu **x3**'e yerleştiren bir ifade yazın.
3. - operatörünü aşırı yüklemeye yarayan bir rutinin **x** sınıfı içinde yer aldığını varsayarak, Soru 2'de belirtilen işin aynısını gerçekleştirecek bir ifade yazın.
4. Doğru/Yanlış: >= operatörünü aşırı yüklenebilir.
5. **COUNTPP1** örneğindeki **Counter** sınıfındaki aşırı yüklenmiş operatör için komple bir tanım yazın. Operatör, **count**'u artırmak yerine azaltsın.
6. Aşırı yüklenmiş bir tekli operatörün tanımında kaç tane argüman gerekir?
7. Nesneleri **obj1**, **obj2** ve **obj3** olan bir sınıf olduğunu varsayın. **obj3 = obj1 - obj2** ifadesinin doğru çalışması için, aşırı yüklenmiş operatör
   a. iki argüman almalıdır.
   b. bir değer döndürmelidir.
   c. ismi olan bir geçici nesne tanımlamalıdır.
   d. kendisinin üyesi olduğu nesneyi operand olarak kullanmalıdır.
8. **ENGPLUS** örneğindeki **Distance** sınıfı için aşırı yüklenmiş bir **++** operatörünün komple tanımını yazın. Bu operatör, **feet** üye verisine 1 eklemelidir ve şu tür ifadeleri mümkün kılmalıdır.

   ```
   dist1++;
   ```

9. Soru 8'i tekrar edin; fakat, şu tür ifadelere de izin verin:

   ```
   dist2 = dist1++;
   ```

10. **++** operatörü önek formunda kullanıldığında sonek formundaki kullanımdan farklı olarak neler gerçekleştirir?
11. İki karakter katarı nesnesini birbirine eklemek için değişik iki yöntem sunan iki deklaratör şöyledir:

    ```
    void add(String s1, String s2)
    String operator + (String s)
    ```

    Aşağıda birinci deklaratör için verilenleri ikinci deklaratördeki uygun karşılıkları ile eşleyin.

    fonksiyon ismi (**add**) _____ ile eşlenir.
    dönüş tipi (**void**) _____ ile eşlenir.
    ilk argüman (**s1**) _____ ile eşlenir.

ikinci argüman (`s2`) _____ ile eşlenir.
fonksiyonun üyesi olduğu nesne _____ ile eşlenir.

a. argüman (`s`)
b. operatörün üyesi olduğu nesne
c. operatör (`+`)
d. dönüş değeri (`String`)
e. bu öğenin eşlenebileceği bir karşılık yok

12. Doğru/Yanlış: Aşırı yüklenmiş bir operatör her zaman operand sayısından bir eksik argüman gerektirir.
13. Aritmetik atama operatörüne yeni bir işlev yüklerseniz, işlemin sonucu

a. operatörün sağında kalan nesneye gider.
b. operatörün solunda kalan nesneye gider.
c. operatörün üyesi olduğu nesneye gider.
d. geri döndürülmelidir.

14. `STRPLUS` örneğindeki `String` sınıfı ile çalışan, aşırı yüklenmiş bir `++` operatörünün tam tanımını yazın. Bu operatörün etkisi operandını büyük harfe çevirmek şeklinde olmalıdır. `toupper()` kütüphane fonksiyonunu kullanabilirsiniz (`CCTYPE` başlık dosyasında). `toupper()`, değiştirilecek karakteri tek argüman olarak alır ve değiştirilmiş karakteri döndürür (veya değiştirmek gerekmiyorsa argümanın kendisini döndürür.)
15. Kullanıcı tarafından tanımlanan bir tipten temel bir veri tipine dönüşüm yapmak için aşağıdakilerden hangisini kullanmanız en muhtemeldir?

a. standart olarak derleyicide bulunan bir dönüşüm operatörü.
b. tek argümanlı bir kurucu fonksiyon.
c. aşırı yüklenmiş bir = operatörü.
d. söz konusu sınıfın üyesi olan bir dönüşüm operatörü.

16. Doğru/Yanlış: `objA = objB;` ifadesi eğer nesneler farklı sınıfa ait ise, derleyicinin hata mesajı vermesine neden olur.
17. Bir temel veri tipinden kullanıcının tanımladığı bir tipe dönüşüm yapmak için aşağıdakilerden hangisini kullanmanız en muhtemeldir?

a. standart olarak derleyicide bulunan bir dönüşüm operatörü.
b. tek argümanlı bir kurucu fonksiyon.
c. aşırı yüklenmiş bir = operatörü.
d. söz konusu sınıfın üyesi olan bir dönüşüm operatörü.

18. Doğru/Yanlış: Eğer `aclass obj = intvar;` gibi tanımları ele alacak bir kurucu fonksiyon tanımlamışsanız, `obj = intvar;` gibi ifadeler de kullanabilirsiniz.
19. Eğer `objA`, `A` sınıfının ve `objB`, `B` sınıfının içinde ise, `objA = objB` demek istiyorsanız; ayrıca, dönüşüm rutininin `A` sınıfında yer almasını da istiyorsanız, ne tür bir dönüşüm rutini kullanabilirsiniz?
20. Doğru/Yanlış: `*` operatörünü bölme yapacak şekilde aşırı yüklerseniz, derleyici buna karşı çıkmaz.
21. Bir UML şemasında hangi durumda bir bağlantı söz konusudur?

a. iki sınıf aynı programdaysa
b. bir sınıf diğerinden türetiliyorsa
c. iki sınıf aynı global değişkeni kullanıyorsa
d. bir sınıf, diğer bir sınıf içinde yer alan bir üye fonksiyonu çağırıyorsa

22. UML'de üye verilere _____ denir; üye fonksiyonlara _____ denir.
23. Doğru/Yanlış: Sınıfları sembolize eden dikdörtgenlerin yuvarlatılmış köşeleri vardır.
24. A sınıfından B sınıfına bir yönlendirme ne anlama gelir?

a. A sınıfının bir nesnesi B sınıfının bir nesnesindeki işlemi çağırabilir
b. A sınıfı ve B sınıfı arasında bir ilişki vardır
c. nesneler A sınıfından B sınıfına gidebilir
d. B sınıfından gönderilen mesajlar A sınıfı tarafından alınır

Alıştırmalar

Yıldızla işaretli alıştırmaların cevaplarını Ek G'de bulabilirsiniz.

1. Bu bölümdeki `ENGLPLUS` programında gördüğümüz `Distance` sınıfına iki uzaklık ölçüsü arasındaki farklı bulacak şekilde aşırı yüklenmiş bir operatör ekleyin. Aşırı yüklenmiş operatörümüz şu tür ifadeleri mümkün kılmalıdır:*

```
dist3 = dist1 - dist2;
```

Operatörün hiçbir zaman büyük sayıları küçük sayılardan çıkarmayacağını (yani negatif mesafe ölçülerine izin verilmeyeceğini) varsayın.

2. Yine bu bölümde ele aldığımız `STRPLUS` programına bakarak, bu örnekteki aşırı yüklenmiş `+` operatörünün yerine aşırı yüklenmiş bir `+=` operatörü koyan bir program yazın. Bu program, aşağıdakine benzer ifadelerin kullanımını sağlamalıdır:*

```
s1 += s2;
```

Burada `s2`, `s1`'e eklenmekte ve sonuç `s1`'e atanmaktadır. Operatör, işlem sonucunun şu gibi diğer hesaplamalarda kullanılmasına da izin vermelidir:

```
s3 = s1 += s2;
```

3. Bölüm 6'nın 3. alıştırmasındaki `time` sınıfını, iki zaman değişkenini `add_time()` fonksiyonu ile değil de aşırı yüklenmiş `+` operatörü ile toplayacak şekilde değiştirin. Bu sınıfı test edecek bir program yazın.*
4. Bölüm 6'nın 1. alıştırmasındaki gibi bir `Int` sınıfı oluşturun. Tamsayı aritmetik operatörlerini (`+`, `-`, `*` ve `/`) `Int` tipinden nesneler üzerinde işlem yapabilecek şekilde aşırı yükleyin. Bu tür bir aritmetik işlemin sonucu `int`'in normal sınırından öteye taşarsa, yani (32 bitlik bir ortamda) 2,147,483,648 ile -2,147,483,648 arasında değilse, operatör bir uyarı vererek programı durdursun. Böyle bir veri tipi, aritmetik taşma durumlarının hata çıkardığı ve kabul edilemez olduğu durumlarda işe yarayabilir. İpucu: Taşma olup olmadığını kontrol etmek için işlemleri `long double` tipini kullanarak yapın. Son olarak bu sınıfı test edecek bir program hazırlayın.*
5. Yukarıdaki 3. alıştırmada bahsettiğimiz `time` sınıfını, hem önek hem de sonek notasyonu üzerinde işlem yapan ve değer döndüren, aşırı yüklenmiş artırma (`++`) ve eksiltme (`--`) operatörleri ile geliştirin. Yeni operatörleri test etmek için `main()` programa uygun ifadeler ekleyin.

6. Bir önceki alıştırmada ele aldığımız `time` sınıfını şöyle aşırı yükleyin: aşırı yüklenmiş (-) operatörü ile iki `time` değeri arasındaki farkı hesaplamak; aşırı yüklenmiş (*) operatörünü kullanarak bir `time` değerini `float` tipi bir sayı ile çarpmak.
7. Bölüm 6'nın 11. alıştırmasında kesirler üzerinde dört işlem yaparken kullandığımız `fraction` sınıfını şu şekilde değiştirin: Toplama, çıkarma, çarpma ve bölme işlemleri, aşırı yüklenmiş operatörlerle yapılsın. ("Döngüler ve Kararlar" adlı Bölüm 3'ün 12. alıştırmasında gördüğümüz kesir aritmetiği kurallarını hatırlayın.) Karşılaştırma operatörleri `==` ve `=!`'in de işlevlerini artırın. Bu operatörleri, kullanıcının programın istediği iki kesir için `0/1, 0/1` girmesi durumda programdan çıkılması için de kullanın. `lowterms()` fonksiyonunu da argümanında verileri kesiri en sade haline çevirip geri göndermesi için değiştirebilirsiniz. Böylece `lowterms()` aritmetik fonksiyonlarda daha yararlı bir şekil alır, çünkü sonuç döndürülmeden hemen önce kullanılabilir.
8. "Diziler ve Karakter Dizileri" başlıklı Bölüm 7'nin 12. alıştırmasında geçen `bMoney` sınıfını, aşırı yüklenmiş operatörlerle yapılan aşağıdaki aritmetik işlemleri de kapsayacak şekilde geliştirin:

```
bMoney = bMoney + bMoney
bMoney = bMoney - bMoney
bMoney = bMoney * long double          //parca birim fiyati * parca sayisi
long double = bMoney / bMoney          //toplam fiyat bolu parca birim fiyati
bMoney = bMoney / long double          //toplam fiyati bolu parca sayisi
```

/operatörünün iki kez aşırı yüklendiğine dikkat edin. Derleyici, argümanlar farklı olduğu için, ikisini birbirinden ayırabilir. Hatırlayacağınız gibi, aynı işlemi kendi `long double` verileri üzerinde yaparak `bMoney` nesneleri üzerinde aritmetik işlemler gerçekleştirmek gayet kolaydır.
`main()` programınız kullanıcıdan iki para karakter katarı ve bir `float` sayı istesin. Bu veriler üzerine beş işlemin hepsi uygulansın ve sonuç ekrana getirilsin. Bunu da bir döngü halinde yapın ki, istendiği sürece yeni işlem yapılabilsin.
Bazı para işlemleri anlamlı değildir. Örneğin, paranın karesi diye bir şey olmadığı için `bMoney * bMoney` ifadesinin gerçek bir karşılığı yoktur. Aynı şekilde `bMoney` ile `long double` bir sayıyı toplayamayız (şu kadar dolar artı şu kadar parça ne anlam ifade eder ki?). Bu tür geçersiz işlemleri önlemek için `bMoney`'nin `long double`'a dönüşümünü ve tersini sağlayan operatörleri işe katmayın. Bunları işe dahil ederseniz ve aşağıdaki gibi bir ifade yazarsanız, derleyici parçaları otomatik olarak `bMoney`'ye çevirir ve toplamayı yapar:

```
bmon2 = bmon1 + widgets; //anlamsiz
```

Operatörleri baştan işe katmazsanız, derleyici bu tür dönüşümleri hata olarak işaretler ve böyle anlamı hatalarının yakalanmasını kolaylaştırır. Aynı şekilde herhangi bir dönüşüm kurucusu kullanırsanız bunu açıkça belirtin.
Operatörün sağında bir nesne olmasını gerektiren ama sol taraf için böyle bir zorunluluğu olmayan, bu yüzden de aşırı yüklenmiş operatörlerle nasıl kullanacağımızı henüz bilmediğimiz başka para işlemleri de olabilir.

```
long double * bMoney          //Bunu henuz yapamayiz: bMoney yalnizca sagda
long double / bMoney          //Bunu henuz yapamayiz: bMoney yalnizca sagda
```


Bu durumu nasıl çözeceğimizi Bölüm 11'de arkadaş (friend) fonksiyonlarını işlerken öğreneceğiz.

9. Bu bölümdeki `ARROVER3` programına kullanılan `safearray` sınıfını, kullanıcının diziye hem alt hem de üst sınır belirleyeceği şekilde geliştirin (örneğin dizi elemanlarının indeksi 100'den 200'e kadar gitsin). İşlevi artırışmış indeks operatörü dizi elemanlarına her erişimde indeksi kontrol ederek sınırların dışına taşmasını önlesin. Bunu sağlamak için üst ve alt limitleri belirleyen iki argümanlı bir kurucu eklemeniz gerekecek. Dinamik olarak bellek ayırmayı henüz öğrenmediğimizden, üye verileri yine 0'dan 99'a kadar giden bir dizi olacak ama belki `safearray`'in indekslerini gerçek `int` dizisindeki farklı indekslerle eşleştirebilirsiniz. Örneğin, 100'den 175'e kadar bir aralık seçtirse bunu `arr[0]`'dan `arr[75]`'e kadar olan aralıkla eşleştirebilirsiniz.
10. Bu soru sadece matematik meraklıları için: Düzlemdeki noktaları kutupsal (polar) koordinatlar (yarıçap ve açı) olarak temsil eden `Polar` adlı bir sınıf oluşturun. İki kutupsal veriyi toplamak için aşırı yüklenmiş bir + operatörü yazın. Düzlemde iki noktayı toplamak için söz konusu noktaların X ve Y koordinatları ayrı ayrı toplanır. Böylece "cevabın" X ve Y koordinatları bulunur. Dolayısıyla, kutupsal koordinatlarla ifade edilen noktalarınızı önce kartezyen koordinatlara çevirmeniz, sonra bunların X ve Y koordinatlarını ayrı ayrı toplayarak sonuç noktasının kartezyen koordinatlarını bulmanız gerek. Nihayet, sonucu yeniden kutupsal sistemde ifade etmeniz isteniyor.
11. `sterling` yapısını hatırlıyor musunuz? Bu yapıyı örneğin "C++ Programlama Esasları" başlıklı Bölüm 2'nin 10. alıştırmasında ve Bölüm 5'in 11. alıştırmasında görmüştük. Bu yapıyı bir sınıfa dönüştürün: Veri elemanları `pounds` (`long` tipi), `shillings` (`int` tipi) ve `pence` (`int` tipi) olsun.
Aşağıdaki üye fonksiyonları ekleyin:

- argümansız kurucu fonksiyon
- `double` tipi tek argümanı olan kurucu fonksiyon (ondalık düzendeki `pound` verilerini dönüştürmek için)
- `pounds`, `shillings` ve `pence` olmak üzere üç argümanlı kurucu fonksiyon
- `getSterling()` kullanıcıdan £9.19.11 formatında bir veriyi `pounds`, `shillings` ve `pence` olarak almak için
- `putSterling()` `pounds`, `shillings` ve `pence`'den oluşan veriyi £9.19.11 formatında göstermek için
- toplama (`+sterling`) aşırı yüklenmiş + operatörünü kullanın
- çıkarma (`-sterling`) aşırı yüklenmiş - operatörünü kullanın
- çarpma (`*double`) aşırı yüklenmiş * operatörünü kullanın
- bölme (`/sterling`) aşırı yüklenmiş / operatörünü kullanın
- bölme (`/double`) aşırı yüklenmiş / operatörünü kullanın
- `double` operatörü (`double`'a dönüşüm için)

Bu işlemleri yapmak için (örneğin) her nesnenin verisini ayrı ayrı ekleyebilirsiniz. `Pence` hanelerini toplayıp, elde kalan olursa bunu `shillings`'e aktarıp sonra da `shillings` verilerini toplayarak devam etmek şeklinde hareket edebilirsiniz. Ancak, her iki `sterling` nesnesini `double` tipine dönüştürerek toplamak ve sonucu `sterling`'e çevirmek daha kolaydır. Bunun için gerekli aşırı yüklenmiş + operatörü şöyledir:

```
sterling sterling::operator + (sterling s2)
  {
```

```
    return sterling( double(sterling(pounds, shillings, pence))
                                                    +double(s2) );
}
```

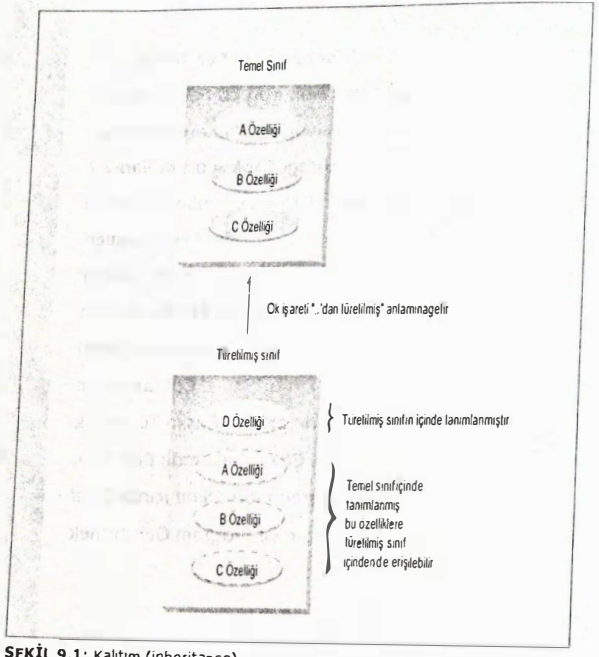
Bu ifadeler iki geçici **double** değişken oluşturur. Değişkenlerin biri, üyesi olduğu fonksiyonun nesnesinden gelir. Diğeri de **s2** adlı argümandan alınır. **double** tipinden olan bu değişkenler toplanır, sonuç yeniden **sterling**'e çevrilir ve geri döndürülür. Dikkat ederseniz, **sterling** sınıfını kullanırken **bMoney**'deki yaklaşımımızdan farklı bir felsefe kullandık. **sterling** örneğinde dönüşüm operatörlerini kullanarak geçersiz matematiksel işlemleri yakalama fırsatı elde ettik. Aşırı yüklenmiş matematiksel işlemleri yazarak da kodumuzu basitleştirdik.

12. Hem 8. alıştırmadaki **bMoney** sınıfını hem de 11. alıştırmadaki **sterling** sınıfını içeren bir program yazın. Bir pound'un ($1.00) elli dolara ($50.00) eşit olduğunu varsayarak, **bMoney** ve **sterling** arasında dönüştürme işlemi yapacak dönüşüm operatörlerini yazın. Bu, 19 .yüzyılda Britanya İmparatorluğu en parlak dönemindeyken ve pound-shilling-pence formatı kullanımdayken yaklaşık çapraz kur oranıydı. Kullanıcının bu para birimlerinden birini kullanarak bir para miktarı girdiği bir **main()** programı yazın. Program daha sonra bu para mjktarını diğer para birimine çevirmeli ve sonucu ekranda göstermeli. **bMoney** ve **sterling** sınıfları üzerinde yapacağınız değişiklikleri en aza indirgeyin.

9

KALITIM

Kalıtım (inheritance), nesne yönelimli programlamanın sınıflardan sonra muhtemelen en güçlü özelliğidir. Kalıtım, mevcut olan sınıflardan veya *temel sınıflardan, türetilmiş sınıf* denilen yeni sınıflar oluşturma işlemidir. Türetilmiş sınıf, temel sınıfın tüm beceritelerine sahip olur; üstelik, kendisine ait özellikler ve ilaveler katabilir. Temel sınıf bu işlemlerden etkilemez. Kalıtım ilişkisi Şekil 9.1'de gösterilmiştir.



ŞEKİL 9.1: Kalıtım (inheritance).

Şekil 9.1'deki ok, umduğunuzun aksine bir yönü gösteriyor. Ok eğer aşağı göstermiş olsaydı, oka *kalıtım* etiketini yapıştırırdık. Ancak, okun türetilmiş sınıftan temel sınıfa doğru yukarıya işaret etmesi ve "türetilmiştir" anlamını taşıyan bir ok olarak düşünülmesi çok daha yaygın bir yaklaşımdır.

Kalıtım, nesne yönelimli programlamanın başlıca parçasıdır. En büyük avantajı ise kodun *yeniden kullanılmasına* izin vermesidir. Bir temel sınıfı yazdıktan ve hatalarını giderdikten sonra bu sınıfa dokunmanıza gerek yoktur. Bununla birlikte, kalıtım kullanarak bu sınıf değişik durumlarda çalışmak üzere ayarlanabilir. Mevcut kodu yeniden kullanmak zamandan ve paradan tasarruf etmeyi sağlar; üstelik, programın okunurluğunu artırır. Kalıtım ayrıca bir programlama probleminin başlangıç aşamasında kavramsal olarak incelenmesine ve programın genel tasarımına yardımcı olabilir.

Yeniden kullanılabilirliğin önemli bir sonucu, sınıf kütüphanelerinin dağıtımının kolaylaşmasıdır. Bu sayede bir programcı, bir başka kişi ve kurum tarafından geliştirilmiş bir sınıfı kullanabilir; üstelik, bu sınıfı değiştirmeden belirli durumlara uygun düşecek başka sınıflar türetebilir.

Kalıtımın kullanımıyla ilgili bazı spesifik örneklere baktıktan sonra kalıtımın bu özelliklerini daha ayrıntılı inceleyeceğiz.

Türetilmiş Sınıf ve Temel Sınıf

Bölüm 8'de "Operatörlerin Aşırı Yüklenmesi" bahsinde ele aldığımız `COUNTPP3` örneğini hatırlıyor musunuz? Bu program genel amaçlı bir sayaç değişkeni olarak bir `Counter` sınıfı kullanmıştı. Sayacın gösterdiği sayıya başlangıç değeri olarak 0 değeri veya kurucu fonksiyon yardımıyla belirli bir değer atanabilir. Daha sonra bu sayı `++` operatörü ile artırılır ve `get_count()` operatörü ile okunur.

Diyelim ki, `Counter` sınıfının tam istediğimiz gibi çalışmasını sağlamak için uzun ve zorlu bir çalışmanın içine girdik ve tek bir şey haricinde elde ettiğimiz sonuçtan memnunuz. Sayacın değerini azaltacak bir yönteme gerçekten ihtiyacımız var. Belki bir bankaya giren insanları sayıyoruz. İnsanlar bankaya girdikçe sayacı artırıyoruz ve bankadan çıktıkça azaltıyoruz. Böylece, sayaç herhangi bir anda bankanın içindeki insan sayısını göstermiş olur.

`Counter` sınıfının kaynak kodunun içine doğrudan bir eksiltme rutini ekleyebiliriz. Yine de, bunu yapmak istemeyebiliriz. Bunun ayrı ayrı sebepleri vardır. Birincisi, `Counter` sınıfı çok iyi çalışıyor; üstelik, saatler süren test ve hata ayıklama aşamalarından geçmiş. (Elbette bu kez biraz abartıyoruz; fakat, daha büyük ve daha karmaşık sınıflar için bu doğru olabilir.) Eğer `Counter` kaynak koduyla zaman kaybetmeye başlarsak, test işlemi sil baştan tekrarlanmak zorunda kalır. Ayrıca, bazı şeyleri kaçınılmaz biçimde berbat edebiliriz ve değişiklik yapmadan önce düzgün çalışan bir kod üzerinde hata ayıklamak için saatler harcayabiliriz.

Bazı durumlarda `Counter` sınıfını değiştirmemenin başka bir sebebi daha olabilir: Özellikle bir sınıf kütüphanesinin bir parçası olarak dağıtılmış ise `Counter` sınıfının koduna erişim hakkımız olmayabilir. (Bu konuyu daha ayrıntılı Bölüm 13'te "Birden Fazla Dosya Kullanan Programlar" başlığı altında ele alacağız.)

Bu problemleri önlemek amacıyla `Counter`'ı temel alan ve `Counter`'ın kendisini değiştirmeyen yeni bir sınıf tanımlamak için kalıtım kullanabiliriz. `COUNTEN` programının listesi aşağıda verilmiştir. Bu program, `CountDn` isminde yeni bir sınıf içerir ve `Counter` sınıfına bir eksiltme operatörü ekler:

```
// counten.cpp
// Counter sinifi ile kalitim
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter                           //temel sinif
   {
   protected:                           //DIKKAT: private degil
      unsigned int count;               //count: sayactaki sayi
   public:
      Counter() : count(0)              //argumansiz kurucu fonksiyon
         { }
      Counter(int c) : count(c)         //1-argumanli kurucu fonksiyon
```

```
        { }
      unsigned int get_count() const   //count'u dondur
        { return count; }
      Counter operator ++ ()           //count'u bir artir (önek)
        { return Counter(++count); }
   };
//////////////////////////////////////////////////////////////
class CountDn : public Counter         //turetilmis sinif
   {
   public:
      Counter operator -- ()           //count'u bir azalt (sonek)
        { return Counter(--count); }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   CountDn c1;                         //CountDn sinifina ait c1

   cout << "\nc1=" << c1.get_count();  //c1'i ekranda goster

   ++c1; ++c1; ++c1;                   //c1'i 3 kez artir
   cout << "\nc1=" << c1.get_count();  //ekranda goster

   --c1; --c1;                         //c1'i iki kez azalt
   cout << "\nc1=" << c1.get_count();  //ekranda goster
   cout << endl;
   return 0;
   }
```

Program listesi **Counter** sınıfıyla başlar. **Counter** sınıfının **COUNTPP3** programındaki görünütüsü değiştirilmez (yalnızca küçük bir istisna var, bunu daha sonra ele alacağız). Dikkat ederseniz, kolaylık olsun diye bu programı **POSTFIX** programı üzerinde modellemedik. **POSTFIX** programı, sonek notasyon sunmak amacıyla aşırı yüklenmiş ikinci bir ++ operatörü içeriyordu.

Türetilmiş Sınıfı Açıkça Belirtmek

Program listesinde **Counter** sınıfının peşinden **CountDn** isimli yeni bir sınıf tanımı gelir. Bu sınıf, sayaçtaki sayıyı eksilten **operator--()** isminde yeni bir fonksiyon içerir. Bununla beraber – işte kilit nokta burası – yeni **CountDn** sınıfı, **Counter** sınıfının tüm özelliklerini devralır. **CountDn**, bir kurucu fonksiyon, **get_count()** fonksiyonu veya **operator++()** fonksiyonu gerektirmez; çünkü bunlar zaten **Counter**'da mevcuttur.

CountDn'ın ilk satırı, **CountDn**'ın **Counter**'dan türetildiğini açıkça belirtir:

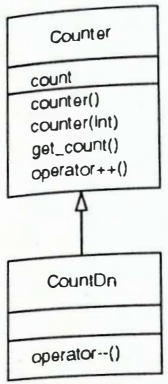
```
class CountDn : public Counter
```

Burada sadece bir tane iki nokta işareti (kapsam çözünürlük operatöründe olduğu gibi "iki" tane iki nokta işareti değil) kullanılır. Bu işaretin peşinden **public** anahtar kelimesi ve temel sınıf **Counter**'ın ismi gelir. Bu satır, sınıflar arası ilişki kurar ve şunu söyler: **CountDn**, **Counter** *temel sınıfından türetilmiştir.* (**public** anahtar kelimesinin etkisini daha sonra inceleyeceğiz.)

UML Sınıf Şemalarında Genelleştirme

UML'de kalıtıma *genelleştirme* denir; çünkü, temel sınıf çocuk sınıfın çok daha genelleştirilmiş bir şeklidir. Başka türlü ifade edersek; çocuk annenin daha spesifik bir versiyonudur. (UML'i

"Genel Görünüm" adlı Bölüm 1'de tanıtmıştık; sınıf şemalarıyla ise "Operatörlerin Aşırı Yüklenmesi" adlı Bölüm 8'de karşılaştık.) **COUNTEN** programının genelleştirilmiş hali Şekil 9.2'de gösteriliyor.



ŞEKİL 9.2: COUNTEN için UML sınıf şeması.

UML sınıf şemalarında genelleştirme, temel sınıf ile çocuk sınıfları birleştiren çizginin ucunda üçgen bir ok işareti ile belirtilir. Hatırlarsanız ok, *devralınmıştır, türetilmiştir* veya *daha spesifik bir versiyonudur* anlamlarına gelir. Okun yönü, türetilmiş sınıfın temel sınıfın verilerine ve fonksiyonlarına erişebileceğini, fakat temel sınıfın türetilmiş sınıflara erişim hakkı olmadığını vurgular.

Şemadaki sınıflara nitelikler (üye veriler) ve işlemler (üye fonksiyonlar) eklediğimize dikkat edin. En üstteki bölüm sınıfın ismini, ortadaki bölüm nitelikleri tutar; en aşağıdaki bölüm ise işlemler için ayrılır.

Temel Sınıfın Üyelerine Erişmek

Kalıtımdaki önemli bir konu, temel sınıfın bir üye fonksiyonunun, türetilmiş sınıfın nesneleri tarafından ne zaman kullanılacağını bilmektir. Buna *erişilebilirlik* (accessibility) denir. Şimdi, derleyicinin erişilebilirlik konusunu nasıl ele aldığını **COUNTEN** örneği üzerinde görelim.

Temel Sınıfın Kurucu Fonksiyonlarını Kullanmak

COUNTEN'in **main()** programında **CountDn** sınıfının bir nesnesini tanımlanır:

```
CountDn c1;
```

Bu ifade, **c1**'in **CountDn** sınıfının bir nesnesi olarak tanımlanmasını ve başlangıç değeri olarak 0 verilmesini sağlar. Fakat durun – bu nasıl mümkün olabilir? **CountDn** sınıf tanımında hiç kurucu fonksiyon yoktur. Öyleyse, ilk kullanıma hazırlama işini kim yapıyor? Görünen o ki, en azından belirli şartlar altında, eğer kurucu fonksiyonu açıkça belirtmemişseniz, türetilmiş sınıf

temel sınıftan uygun bir kurucu fonksiyon kullanacaktır. COUNTEN'de CountDn içinde tanımlanmış bir kurucu fonksiyon yoktur; bu nedenle, derleyici Count'un argümansız kurucu fonksiyonunu kullanır.

Derleyici tarafındaki bu esneklik, yani bir fonksiyon hazır olmadığında bir diğerini kullanmak, kalıtımla ilgili durumlarda düzenli olarak karşımıza çıkar. Genellikle bir fonksiyonu bir diğerinin yerine kullanma işlemi, istenilen bir şeydir; ancak kimi zaman cesaret kırıcı da olabilir.

Temel Sınıfın Üye Fonksiyonlarını Diğer Fonksiyonların Yerine Kullanmak

CountDn sınıfının c1 nesnesi de Counter sınıfının operator++() ve get_count() fonksiyonlarını kullanır. İlki c1'i artırmak için kullanılır:

```
++c1;
```

İkincisi c1'in içindeki sayıyı ekranda göstermek için kullanılır:

```
cout << "\nc1=" << c1.get_count();
```

Derleyici, bu fonksiyonları c1'in üyesi olduğu sınıfın içinde bulamaması üzerine yine temel sınıfın üye fonksiyonlarını kullanır.

COUNTEN'in Çıktısı

main() içinde c1 üç kez artırılır, elde edilen değer ekranda gösterilir, c1 iki kez azaltılır ve değerin tekrar ekranda gösterilir. Programın çıktısı şöyle olur:

```
c1=0        <==== ilk değer atandıktan sonra
c1=3        <==== ++c1, ++c1, ++c1 işlemlerinden sonra
c1=1        <==== --c1, --c1 işlemlerinden sonra
```

++ operatörü, kurucu fonksiyonlar, Counter sınıfı içindeki get_count() fonksiyonu ve CountDn sınıfı içindeki -- operatörü, CountDn tipindeki nesnelerle çalışırlar.

protected Erişim Belirteci

Bir fonksiyonun işlevselliğini kendisini değiştirmeden artırmış olduk. Yani, neredeyse değiştirmeden demek istiyoruz. Şimdi Counter sınıfın üzerinde yaptığımız tek değişikliğe göz atalım.

Şimdiye kadar incelediğimiz sınıfların verileri, önceki COUNTPP3 programındaki Counter sınıfının count'u da buna dahil olmak üzere, private erişim belirtecini kullanıyorlardı.

COUNTEN'de Counter sınıfının içinde count'a yeni bir erişim belirteci verilir: protected. Bu ne işe yarar?

Öncelikle, private ve public erişim belirteçleri hakkında neler bildiğimizi bir özetleyelim. public veya private olsun, bir sınıfın bir üye fonksiyonu, sınıf üyelerine her zaman erişebilir. Ancak, dışarıda tanımlanmış bir nesne, sınıfın yalnızca public üyelerine (nokta operatörünü kullanarak, örneğin) erişebilir. Bu nesnenin private üyeleri kullanmasına izin verilmez. Söz gelişi, objA'nın, A sınıfının bir örnek kopyası olduğunu ve funcA() fonksiyonunun da A'nın bir

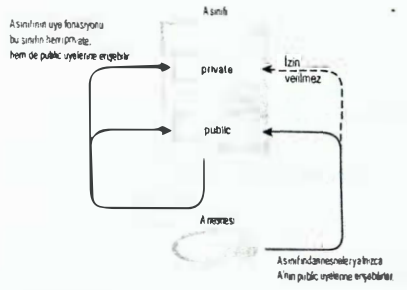
üye fonksiyonu olduğunu varsayalım. Sonra main()'de (ya da A'nın üyesi olmayan herhangi bir fonksiyonda) şu ifade, ancak funcA() public ise kurallara uygun bir ifade olur:

```
objA.funcA();
```

objA, A sınıfının public üyelerini çağıramaz. private üyeler özeldir. Bu, Şekil 9.3'te gösteriliyor.

Eğer kalıtım kullanmıyorsak, bilmemiz gerekenin tümü bundan ibarettir. Kalıtım kullandığımızda ise buna ilave çok sayıda ihtimal daha ortaya çıkar. Bu aşamada bizi ilgilendiren soru şudur: Türetilmiş sınıfın üye fonksiyonları temel sınıfın üyelerine erişebilirler mi? Başka bir deyişle, CountDn'ın içindeki operator--() fonksiyonu Counter'ın içindeki count'a erişebilir mi? Yanıt: Üye fonksiyonlar temel sınıfın üyelerine ancak, üyeler public veya protected (korumalı) olarak tanımlanmışsa erişebilirler. private üyelere erişemezler.

count'u public olarak tanımlamak istemiyoruz; çünkü, bu tür bir tanımlama, programın herhangi bir yerindeki herhangi bir fonksiyonun count'a erişmesine imkan verecektir, veri gizliliği ilkesinin sağladığı avantajı da ortadan kaldıracaktır. Bir protected üye, diğer yandan, kendi sınıfının veya – işte kilit nokta burası – kendi sınıfından türetilmiş herhangi bir sınıfın üye fonksiyonları tarafından erişilebilir. Bu sınıfların dışında yer alan fonksiyonlar, örneğin main() tarafından erişilemez. Bu tam bizim istediğimiz gibidir. Şekil 9.4'te bu durum açıklanıyor.



ŞEKİL 9.3: Kalıtım olmadan erişim belirteçleri.

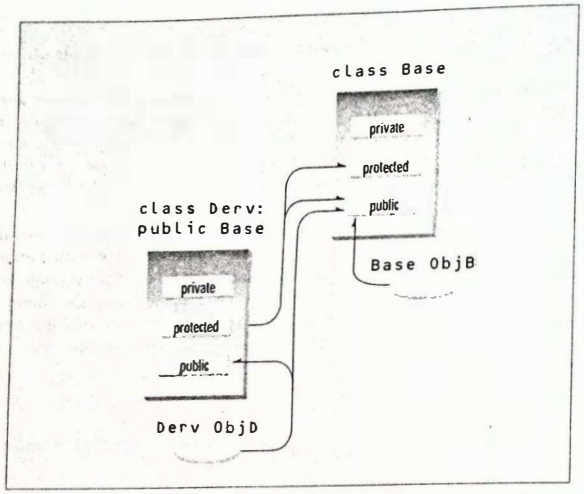
Tablo 9.1 bu durumu farklı bir açıdan özetliyor.

TABLO 9.1: Kalıtım ve Erişilebilirlik

Erişim	Kendi Sınıfından Erişilebilirlik	Türetilmiş Sınıftan Erişilebilirlik	Sınıf Dışındaki Nesnelerden Erişilebilirlik
public	evet	evet	evet
protected	evet	evet	hayır
private	evet	hayır	hayır

İşin özü şu: Diğer sınıflar için gelecekte bir gün temel sınıf olarak kullanılabileceğini umduğunuz bir sınıf yazıyorsanız, türetilmiş sınıfların erişme ihtimali olan üye verilerinin her

biri **private** yerine **protected** olarak tanımlanmalıdır. Bu yaklaşım sınıfın "kalıtıma hazır" olduğunu garanti ediyor.



ŞEKİL 9.4: Kalıtımla birlikte erişim belirteçleri.

protected Belirtecinin Tehlikeleri

Sınıf üyelerini **protected** (korumalı) yapmanın dezavantajları olduğunu bilmelisiniz. Diyelim ki, etrafa dağıtacağınız bir sınıf kütüphanesi yazdınız. Bu kütüphaneyi alan herhangi bir programcı, sizin sınıflarınızın protected üyelerine, sadece bu sınıflardan başka sınıflar türeterek erişebilir. Bu durum protected üyeleri private üyelerden çok daha az güvenilir kılar. Verilerin bozulmasını önlemek için türetilmiş sınıfları, temel sınıfın sadece public fonksiyonlarını kullanarak temel sınıftaki verilere erişmeye zorlamak daha güvencelidir. Tıpkı sıradan **main()** programlarının yapmak zorunda olduğu gibi. **protected** belirtecini kullanmak daha basit programlamaya yol açar. Bu nedenle, bu kitaptaki örneklerde **protected** belirtecine güveniyoruz - belki biraz da fazla güveniyoruz. Programlarınızda, **protected**'in avantajlarını dezavantajlarına karşı tartmanız gerekecektir.

Temel Sınıf Değişmez

Unutmayınız ki, kendisinden başka sınıflar türetiliyor olsa da temel sınıf değişmeden kalır. **COUNTEN** programının **main()** bölümünde, **Counter** tipinde nesneler tanımlayabiliriz:

```
Counter c2;  <==== temel sınıfın bir nesnesi
```

Bu tür nesneler, **CountDn** mevcut olmadığında nasıl davranırlarsa aynı biçimde davranacaklardır.

Kalıtımın ters yönde ele alınamayacağına dikkat edin. Temel sınıf ve temel sınıfın nesneleri, temel sınıftan türetilen sınıfların hiçbiri hakkında bilgi sahibi değildir. Bu örnekte bunun anlamı, **Counter** sınıfının nesneleri, mesela **c2**, **CountDn** içindeki **operator--()** fonksiyonunu kullanamaz demektir. Değerini azaltabileceğiniz bir sayaç istiyorsanız, bu sayacın sınıfı **CountDn** olmalıdır, **Counter** değil.

Diğer Terimler

Bazı dillerde temel sınıf, *üst sınıf*; türetilmiş sınıf ise *alt sınıf* olarak adlandırılır. Bazı yazarlar ise temel sınıftan *ebeveyn* ve türetilmiş sınıftan da *çocuk* diye bahsederler.

Türetilmiş Sınıf Kurucu Fonksiyonları

COUNTEN programında potansiyel bir problem söz konusudur. Bir **CountDn** nesnesine bir değer atamak istersek ne olur? **Counter** sınıfındaki tek argümanlı kurucu fonksiyon kullanılabilir mi? Hayır. **COUNTEN**'de gördüğümüz gibi, derleyici temel sınıftan argümansız bir kurucu fonksiyonu buraya yerleştirecektir ama, böylelikle daha karmaşık kurucu fonksiyonlar ortaya çıkacaktır. Bu tür bir tanımın düzgün çalışması amacıyla türetilmiş sınıf için yeni bir grup kurucu fonksiyon yazmamız gerekir. **COUNTEN2**'de bu durum gösteriliyor.

```
// counten2.cpp
// turetilmis sinif icinde kurucu fonksiyonlar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Counter
   {
   protected:                          //DIKKAT:ozel degil
      unsigned int count;              //count
   public:
      Counter() : count()              //argumansiz kurucu fonksiyon
         { }
      Counter(int c) : count(c)        //1 argumanli kurucu fonksiyon
         { }
      unsigned int get_count() const   //count'u dondur
         { return count; }
      Counter operator ++ ()           //count'u bir kez artir (onek)
         { return Counter(++count); }
   };
////////////////////////////////////////////////////////////
class CountDn : public Counter
   {
   public:
      CountDn() : Counter()            //kurucu fonksiyon, argumansiz
         { }
      CountDn(int c) : Counter(c)      //kurucu fonksiyon,1 argumanli
         { }
      CountDn operator -- ()           //count'u bir azalt (sonek)
         { return CountDn(--count); }
   };
////////////////////////////////////////////////////////////
int main()
   {
   CountDn c1;                         //CountDn sinifi
   CountDn c2(100);
```

```
    cout << "\nc1=" << c1.get_count();  //ekranda goster
    cout << "\nc2=" << c2.get_count();  //ekranda goster

    ++c1; ++c1; ++c1;                  //c1'i artir
    cout << "\nc1=" << c1.get_count();  //c1'i ekranda goster

    --c2; --c2;                        //c2'yi azalt
    cout << "\nc2=" << c2.get_count();  //c2'yi ekranda goster

    CountDn c3 = --c2;                 //c3'u c2'den uret
    cout << "\nc3=" << c3.get_count();  //c3'u ekranda goster
    cout << endl;
    return 0;
    }
```

Bu program **CountDn** sınıfına ait iki yeni kurucu fonksiyon kullanır. Argümanı olmayan kurucu fonksiyon şöyledir:

```
CountDn() : Counter()
   { }
```

Bu kurucu fonksiyon alışılmadık bir özelliğe sahiptir: İki nokta işaretinden sonra fonksiyon ismi gelir. Bu ifade, **CountDn()** kurucu fonksiyonunun temel sınıftaki **Counter()** kurucu fonksiyonunu çağırmasını sağlar. **main()**'de şu tür bir ifade kullandığımızda derleyici **CountDn** tipinde bir nesne oluşturur ve bu nesneye bir değer vermesi için **CountDn** kurucu fonksiyonunu çağırır:

```
CountDn c1;
```

Bu kurucu fonksiyon, sırası geldiğinde, işi yürütecek olan **Counter** kurucu fonksiyonunu çağırır. **CountDn()** kurucu fonksiyonu kendisi ilave ifadeler ekleyebilir; fakat, bu örnekte buna gerek yoktur. Bu nedenle, küme parantezleri içindeki fonksiyon gövdesi boştur.

Başlangıç değerlerinin atandığı listeden bir kurucu fonksiyonu çağırmak biraz tuhaf görünebilir, ama mantıklıdır. Türetilmiş veya temel sınıfın kurucu fonksiyonları çalışmaya başlamadan önce, türetilmiş veya temel sınıfın içinde yer alan herhangi bir değişkenin başlangıç değerinin atanmış olmasını istersiniz. Türetilmiş sınıfın kurucu fonksiyonu çalışmaya başlamadan önce temel sınıfın kurucu fonksiyonunu çağırarak bu işi gerçekleştiriyoruz.

main() içindeki şu ifade **CountDn** içinde yer alan tek argümanlı kurucu fonksiyonu kullanır:

```
CountDn c2(100);
```

Bir kurucu fonksiyon da temel sınıfta karşılık gelen tek argümanlı kurucu fonksiyonu çağırır:

```
CountDn(int c) : Counter(c)          <==== c argümanı, Counter'a aktarılır
   { }
```

Bu ifade, **c** argümanının **CountDn**'den **Counter**'a aktarılmasını sağlar. Bu argüman **Counter** içinde, nesneyi ilk kullanıma hazırlamak amacıyla kullanılacaktır.

main()'de **c1** ve **c2** nesnelerini ilk kullanıma hazırladıktan sonra biri bir kez artırılır, diğeri bir kez azaltılır ve sonuçlar ekrana yazdırılır. Değer atama ifadesinde tek argümanlı kurucu fonksiyon da kullanılabilir.

```
CountDn c3 = --c2;
```


Üye Fonksiyonları Dikkate Almamak

Bir türetilmiş sınıf içinde temel sınıftakilerle aynı isme sahip üye fonksiyonlar kullanabilirsiniz. Böyle bir şey gerçekleştirdiğinizde, programınızdaki çağrılar hem temel sınıf hem de türetilmiş sınıflar için aynı biçimde çalışır.

Şimdi, Bölüm 7'de "Diziler ve Karakter Dizileri" bahsinde incelediğimiz **STAKARAY** programını temel alan bir örneğe bakalım. Bu program, basit bir veri yedekleme aygıtı olan bir yığını modelliyordu. Program, tamsayıları yığının en üstüne yerleştirmenize ve yığının en üstünden almanıza imkan veriyordu. Bununla beraber, **STAKARAY**'in potansiyel bir kusuru vardı. Eğer yığına çok fazla öğe yerleştirmeye kalkarsanız program patlayabilir; çünkü, veriler bellekte **st[]** dizisinin sınırından daha ileriye yerleştirilecektir. Ya da çok fazla öğeyi yığından almaya çalışırsanız, elde edilen sonuçlar manasız olur; çünkü, bellekte dizinin dışında kalan yerlerden okuma yapıyor olacaksınız.

Bu kusurları düzeltmek için **Stack** sınıfından türetilen yeni bir sınıf tanımlıyoruz: **Stack2**. **Stack2**'in nesneleri, **Stack**'in nesneleri ile aynı davranışı gösterir; bir farkla, yığına çok fazla öğe eklemeye çalıştığınızda veya boş bir yığından öğe almaya kalktığınızda uyarı mesajı ile karşılaşırsınız. **STAKEN** için program listesi şöyledir:

```
// staken.cpp
// temel ve turetilmis siniflarda fonksiyonlarin asiri yuklenmesi
#include <iostream>
using namespace std;
#include <process.h>                    //exit() icin
////////////////////////////////////////////////////////////////
class Stack
   {
   protected:                           //DIKKAT: private olamaz
      enum { MAX = 3 };                 //yigin dizisinin buyuklugu
      int st[MAX];                      //yigin: tamsayi dizisi
      int top;                          //yiginin en ustunun indeksi
   public:
      Stack()                           //kurucu fonksiyon
         { top = -1; }
      void push(int var)                //sayiyi yigina koy
         { st[++top] = var; }
      int pop()                         //sayiyi yigindan al
         { return st[top--]; }
   };
////////////////////////////////////////////////////////////////
class Stack2 : public Stack
   {
   public:
      void push(int var)                //sayiyi yigina koy
         {
         if(top >= MAX-1)               //yigin doluysa, hata
            { cout << "\nError: stack is full"; exit(1); }
         Stack::push(var);              //Stack sinifindan push()'u cagir
         }
      int pop()                         //sayiyi yigindan al
         {
         if(top < 0)                    //yigin bossa, hata
            { cout << "\nError: stack is empty\n"; exit(1);}
         return Stack::pop();           //Stack sinifindan pop()'u cagir
         }
   };
```

```
//////////////////////////////////////////////////////////
int main()
   {
   Stack2 s1;

   s1.push(11);                          //yigina bazi degerler yerlestir
   s1.push(22);
   s1.push(33);

   cout << endl << s1.pop();             //yigindan bazi degerler al
   cout << endl << s1.pop();
   cout << endl << s1.pop();
   cout << endl << s1.pop();          //hooop, cok fazla aldin...
   cout << endl;
   return 0;
   }
```

Bu programda **Stack** sınıfı **STAKARAY** programındaki ile aynı; yalnızca, bu programda üye verileri **protected** yapıldı.

Hangi Fonksiyon Kullanılır?

Stack2 sınıfı iki fonksiyon içerir: **push()** ve **pop()**. Bu fonksiyonlar **Stack**'taki fonksiyonlarla aynı fonksiyon ve argüman ismine ve dönüş tipine sahiptir. Bu fonksiyonları aşağıdaki gibi bir ifade kullanarak **main()**'den çağırdığımız zaman derleyici bu iki **push()** fonksiyonundan hangisini kullanacağını nereden biliyor?

```
s1.push(11);
```

Kural şudur: Aynı fonksiyon hem temel sınıfta hem de türetilmiş sınıfta mevcutsa türetilmiş sınıftaki fonksiyon çalıştırılır. (Bu, türetilmiş sınıfın nesneleri için de geçerlidir. Temel sınıfın nesneleri türetilmiş sınıf hakkında hiçbir bilgiye sahip değildir; bu nedenle, her zaman temel sınıf fonksiyonlarını kullanacaklardır.) Bunu şöyle ifade ediyoruz; türetilmiş sınıfa ait fonksiyon temel sınıftaki fonksiyonu *dikkate almıyor*. Bu nedenle, bir önceki ifadede **s1**, **Stack2**'nin bir nesnesi olduğu için **Stack2**'nin içindeki **push()** fonksiyonu çalıştırılır; **Stack**'ın içindeki değil...

Stack2'deki **push()** fonksiyonu yığının dolu olup olmadığını kontrol eder. Eğer doluysa, bir hata mesajı verir ve programın çalışmasını durdurup, çıkmasına neden olur. Eğer dolu değilse, **Stack**'ın içindeki **push()** fonksiyonunu çağırır. Aynı şekilde, **Stack2** içindeki **pop()** fonksiyonu da yığının boş olup olmadığını kontrol ediyor. Eğer boşsa, bir hata mesajı verir ve çıkar; değilse, **Stack**'ın içindeki **pop()** fonksiyonunu çağırır.

main() içinde yığına üç öğe eklenir, fakat yığından dört öğe çıkarılır. Yığından öğe alma işleminin sonuncusu bir hata mesajının ortaya çıkmasına neden olur ve programı sonlandırır.

```
33
22
11
Error:stack is empty
```

Dikkate Alınmayan Fonksiyonlar ile Kapsam Çözünürlüğü

Stack2'nin içindeki **push()** ve **pop()**, **Stack**'ın içindeki **push()** ve **pop()**'a nasıl erişir? Bu fonksiyonlar şu ifadede


```
Stack::push(var);
```

ve şu ifadede

```
return Stack::pop();
```

kapsam çözünürlük operatörünü (::) kullanırlar. Bu ifadeler, **Stack**'ın içindeki **push()** ve **pop()** fonksiyonlarının çağrılacağını belirtir. Kapsam çözünürlük operatörü kullanılmasaydı, derleyici **Stack2**'nin içindeki **push()** ve **pop()** fonksiyonlarının kendilerini çağırdıklarını düşünecekti – ki bu, bu örnekte programın hatalı çalışmasına yol açacaktı. Kapsam çözünürlük operatörü, fonksiyonun tam olarak hangi sınıfın üyesi olduğunu açıkça belirtmeye imkan verir.

İngiliz Ölçü Sistemine Dayanan Distance Sınıfında Kalıtım

Şimdi, kalıtımın nispeten biraz daha karmaşık bir örneğine bir bakalım. Bu kitapta şu ana kadar yer alan çeşitli programlarda İngiliz ölçü sistemine göre tanımlanan **Distance** sınıfındaki uzaklıkların her zaman pozitif olarak simgeleneceği varsayıldı. Mimari çizimlerde gerçekten de böyledir. Ancak, diyelim ki değişen gelgitleri göz önüne alarak Pasifik okyanusundaki su seviyesini ölçüyorsak, negatif ayak ve inç değerlerini simgelemek isteyebiliriz. (En alçak ve alçak su seviyelerinin ortalamasının altında kalan gelgitler *eksi gelgit* olarak adlandırılır. Bu ifade, deniz kabuğu koleksiyoncularının açığa çıkan daha geniş kumsal alanından yararlanabilecekleri ne işaret eder.)

Distance sınıfından yeni bir sınıf türetelim. Bu sınıf, ayak ve inç ölçülerimize bir tek veri öğesi daha ekleyecektir: Pozitif veya negatif olabilen bir işaret. İşareti sınıfa eklediğimizde üye fonksiyonları da değiştirmemiz gerekir; böylece, bu fonksiyonlar işaretli uzaklıklar da çalışabilirler. **ENGLEN** için program listesi şöyledir:

```
// englen.cpp
// Ingiliz olcu sistemine dayanan Distance sinifi kullanilarak kalitim.
#include <iostream>
using namespace std;
enum posneg {pos,neg };    //DistSign icindeki isaret icin
//////////////////////////////////////////////////////////
class Distance       //Ingiliz olcu sistemine dayanan Distance sinifi
   {
   protected:                        //DIKKAT: private olamaz
      int feet;
      float inches;
   public:                           //argumansiz kurucu fonksiyon
      Distance():feet(0),inches(0.0)
         { }                         //2-argumanli kurucu fonksiyon
      Distance(int ft, float in) : feet(ft), inches(in)
         { }
      void getdist()                 //kullanicidan uzakligi al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }
      void showdist() const          //uzakligi ekranda goster
         { cout << feet << "\'-" << inches << '\"';}
   };
```

```
////////////////////////////////////////////////////////////
class DistSign : public Distance    //Distance'a isaret ekler
   {
   private:
      posneg sign;                  //isaret pos veya neg
   public:
                                    //argumansiz kurucu fonksiyon
      DistSign(): Distance()        //temel sinifin kurucu fonksiyonunu cagir
         { sign = pos;  }           //isarete + degeri ver
                                    //2- veya 3-argumanli kurucu fonksiyon
      DistSign(int ft, float in, posneg sg=pos) :
             Distance(ft, in)       //temel sinifin kurucu fonksiyonunu cagir
         { sign = sg; }             //isareti ayarla
      void getdist()                //kullanicidan uzakligi al
         {
         Distance::getdist();       //temel siniftan getdist()'i cagir
         char ch;                   //kullanicidan isareti al
         cout << "Enter sign (+ or -): "; cin >> ch;
         sign = (ch=='+') ? pos : neg;
         }
      void showdist() const         //uzakligi ekranda goster
         {
         cout << ( (sign==pos) ? "(+)" : "(-)" );  //isareti goster
         Distance::showdist();                     //ft ve in
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   DistSign alpha;                  //argumansiz kurucu fonksiyon
   alpha.getdist();                 //kullanicidan alpha'yi al

   DistSign beta(11, 6.25);         //2-argumanli kurucu fonksiyon

   DistSign gamma(100, 5.5, neg);   //3-argumanli kurucu fonksiyon

                                    //uzakliklarin tumunu ekranda goster
   cout << "\nalpha = "; alpha.showdist();
   cout << "\nbeta = "; beta.showdist();
   cout << "\ngamma = "; gamma.showdist();
   cout << endl;
   return 0;
   }
```

Burada **DistSign** sınıfı programa, işaretli sayıları ele almayı sağlayan bir işlevsellik katar. Bu programdaki **Distance** sınıfı önceki programlardakinin tamamen aynısıdır; sadece bu programdaki **Distance** verileri protected'tır. Aslında bu sefer veriler private da olabilirdi; çünkü, türetilmiş sınıfların hiç biri bu verilere erişmez. Yine de, bunları protected yapmak daha emniyetlidir; çünkü bu sayede, gerektiğinde bir türetilmiş sınıf bu verilere erişebilir.

ENGLEN'in Çalışması

main() program üç değişik, işaretli uzaklık deklare eder. Program **alpha**'nın değerini kullanıcıdan alır, **beta**'ya (+)11'-6.25" ve **gamma**'ya da (-)100'-5.5" ilk değerlerini atar. Program çıktısında ayak ve inç arasındaki tire ile karışmasın diye işaretin etrafında parantez kullanılır. Programın örnek bir çıktısı şöyledir:


```
Enter feet: 6
Enter inches: 2.5
Enter sign (+or -): -

alpha=(-)6 '-2.5 "
beta=(+)11 '-6.25 "
gamma=(-)100 '-5.5 "
```

DistSign sınıfı **Distance** sınıfından türetilir. **DistSign** ek olarak, kendisine **posneg** tipinde **sign** isimli bir tek değişken daha ekler. **sign** değişkeni uzaklığın işaretini tutar. **posneg** tipi, iki olası değer içermesinden ötürü bir **enum** ifadesi içinde tanımlanır. **posneg**, **pos** veya **neg** değerlerinden birine sahip olabilir.

DistSign İçindeki Kurucu Fonksiyonlar

DistSign, **Distance**'dakileri yansıtan iki kurucu fonksiyon içerir. Birincisi argüman almaz, ikincisi iki veya üç argüman alır. İkinci kurucu fonksiyondaki üçüncü argüman (bu, isteğe bağlıdır) **pos** veya **neg** değeri alabilen bir işarettir. Bu argümanın varsayılan değeri **pos**'tur. Bu kurucu fonksiyonlar, **DistSign** tipinde değişkenleri (nesneleri) birkaç yoldan tanımlamaya imkan verir.

DistSign içindeki her iki kurucu fonksiyon, ayak ve inç değerlerini ayarlamak için **Distance**'ta karşılık gelen kurucu fonksiyonları çağırırlar. Daha sonra **sign** değişkenini ayarlar. Argümansız kurucu fonksiyon, bu değişkene her zaman **pos** değerini verir. İkinci kurucu fonksiyon ise, eğer üçüncü argüman değeri sağlanmamışsa **sign** değişkenine **pos** değerini verir; eğer üçüncü argüman açıkça belirtilmişse **sign** değişkenine **pos** veya **neg** değerini atar.

DistSign içindeki ikinci kurucu fonksiyona **main()**'den aktarılan **ft** ve **in** argümanları doğrudan **Distance**'taki kurucu fonksiyonlara gönderilirler.

DistSign İçindeki Üye Fonksiyonlar

Distance'a bir işaret eklemek her iki üye fonksiyonunu etkileyecek sonuçlar doğuruyor. **DistSign** sınıfı içindeki **getdist()** fonksiyonunun kullanıcıya ayak ve inç değerlerinin yanı sıra işareti de sorması gerekir. **showdist()** fonksiyonu da ayak ve inç değerlerinin yanı sıra işareti de ekranda görüntülemesi gerekir. Bu fonksiyonlar şu şekilde;

```
Distance::getdist();
```

ve şu şekildedir:

```
Distance::showdist();
```

Distance içindeki karşılık gelen fonksiyonları çağırırlar. Bu çağrılar ayak ve inç değerlerini alarak ekranda gösterirler. **DistSign** içindeki **getdist()** ve **showdist()**'in gövdeleri daha sonra işareti ele almak üzere işlevlerine devam ederler.

Kalıtıma Yardım Etmek

C++, türetilmiş sınıfları tanımlamayı verimli kılmak için tasarlanmıştır. Bir temel sınıfın parçalarını kullanmak istediğimizde, bu parçalar ister veri, ister kurucu fonksiyon veya üye fonksiyonlar olsun, bunu gerçekleştirmek çok kolaydır. Daha sonra, yeni geliştirilmiş sınıfı tanımlamak için gerekli olan işlevselliği de ekleriz. Dikkat ederseniz, **ENGLEN** içinde kodun hiç-

bir bölümünün kopyasını kullanmak zorunda kalmadık; bunun yerine, temel sınıfın içindeki uygun fonksiyonlardan yararlandık.

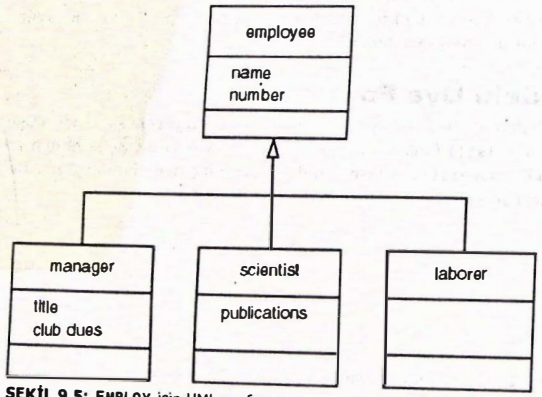
Sınıf Hiyerarşileri

Bu bölümde şimdiye kadar gördüğümüz örneklerde kalıtım, mevcut bir sınıfa işlevsellik kalmak amacıyla kullanıldı. Şimdi kalıtımın farklı bir amaç için kullanıldığı bir örneğe göz atalım: Bir programın orijinal tasarımının bir parçası olarak.

Örneğimiz bir yedek parça şirketinin çalışanlarının veri tabanını modeller. Olayı basitleştirdik; böylece, programda sadece üç tip çalışan simgelenecektir. Yöneticiler idare ederler, bilim adamları daha iyi yedek parçalar geliştirmek için araştırma yaparlar ve işçiler tehlikeli parça baskı preslerini çalıştırırlar.

Çalışanların kategorileri ne olursa olsun, veri tabanı tüm çalışanlar için bir isim (name) ve bir de kimlik numarası (number) saklar. Fakat, yöneticiler için bu bilgilere ek olarak yöneticilerin unvanları (title) ve golf kulübü aidat ödeme tarihleri (club dues) de saklanır. Bilim adamları için yayınladıkları bilimsel makalelerin sayısı (publications) saklanır. İşçiler için isimleri ve kimlik numaraları haricinde ilave bilgiye gerek kalmaz.

Örneğimiz **employee** isimli temel sınıf ile başlar. Bu sınıf, çalışanın soyadını ve kimlik numarasını ele alır. Bu sınıftan diğer üç sınıf daha türetilir: **manager**, **scientist** ve **laborer**. **manager** ve **scientist** sınıfları bu kategorideki çalışanlarla ilgili ek bilgi ve bu bilgiyi ele almak için gerekli üye fonksiyonlar içerirler. Şekil 9.5'te bu durum özetleniyor.



ŞEKİL 9.5: EMPLOY için UML sınıf şeması.

EMPLOY'un program listesi şöyledir:

```
// employ.cpp
// kalitim kullanarak calisanlarin veri tabanini modeller
#include <iostream>
using namespace std;
const int LEN=80;                          //isimlerin maksimum uzunlugu
```



```
//////////////////////////////////////////////////////////////
class employee                             //calisanlar sinifi
   {
   private:
      char name[LEN];                      //calisanin ismi
      unsigned long number;                //calisanin numarasi
   public:
      void getdata()
         {
         cout << "\n Enter last name: "; cin >> name;
         cout << " Enter number: "; cin >> number;
         }
      void putdata() const
         {
         cout << "\ nName: " << name;
         cout << "\n Number: " << number;
         }
   };
//////////////////////////////////////////////////////////////
class manager : public employee            //yonetici sinifi
   {
   private:
      char title[LEN];                     //"baskan yardimcisi " vs.
      double dues;                         //golf kulubu aidat odemeleri
   public:
      void getdata()
         {
         employee::getdata();
         cout << " Enter title: "; cin >> title;
         cout << " Enter golf club dues: "; cin >> dues;
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n Title: " << title;
         cout << "\n Golf club dues: " << dues;
         }
   };
//////////////////////////////////////////////////////////////
class scientist : public employee          //bilim adami sinifi
   {
   private:
      int pubs;                            //yayinlarin sayisi
   public:
      void getdata()
         {
         employee::getdata();
         cout << " Enter number of pubs: "; cin >> pubs;
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n Number of publications: " << pubs;
         }
   };
//////////////////////////////////////////////////////////////
class laborer : public employee            //isci sinifi
   {
   };
```

```
//////////////////////////////////////////////////////////////
int main()
   {
   manager m1, m2;
   scientist s1;
   laborer l1;

   cout << endl;                        //birkac calisanla ilgili verileri al
   cout << "\nEnter data for manager 1";
   m1.getdata();

   cout << "\nEnter data for manager 2";
   m2.getdata();

   cout << "\nEnter data for scientist 1";
   s1.getdata();

   cout << "\nEnter data for laborer 1";
   l1.getdata();
                                        //birkac calisanin verilerini ekranda goster
   cout << "\nData on manager 1";
   m1.putdata();

   cout << "\nData on manager 2";
   m2.putdata();

   cout << "\nData on scientist 1";
   s1.putdata();

   cout << "\nData on laborer 1";
   l1.putdata();
   cout << endl;
   return 0;
   }
```

Programın `main()` bölümü değişik sınıflara ait dört nesne tanımlar: İki yönetici, bir bilim adamı bir de işçi. (Elbette, her tipten çok daha fazla çalışan tanımlanabilir, fakat böyle bir durumda program çıktısı oldukça büyük olur.) Program daha sonra `getdata()` üye fonksiyonunu çağırarak çalışanların her birinin bilgilerini elde eder ve `putdata()` fonksiyonunu çağırarak da bu bilgileri ekranda gösterir. `EMPLOY` ile örnek bir etkileşim şu şekilde olabilir: (Önce kullanıcı veriyi sağlıyor.)

```
Enter data for manager 1
   Enter last name: Wainsworth
   Enter number: 10
   Enter title: President
   Enter golf club dues: 1000000
Enter data on manager 2
   Enter last name: Bradley
   Enter number: 124
   Enter title: Vice-President
   Enter golf club dues: 500000
Enter data for scientist 1
   Enter last name: Hauptman-Frenglish
   Enter number: 234234
   Enter number of pubs: 999
Enter data for laborer 1
```



```
   Enter last name: Jones
   Enter number: 6546544
```

Program daha sonra bunları tekrar görüntüler:

```
Data on manager 1
   Name: Wainsworth
   Number: 10
   Title:President
   Golf club dues:1000000
Data on manager 2
   Name:Bradley
   Number:124
   Title:Vice-President
   Golf club dues:500000
Data on scientist 1
   Name:Hauptman-Frenglish
   Number:234234
   Number of publications:999
Data on laborer 1
   Name:Jones
   Number:6546544
```

Daha sofistike bir program, verileri düzenlemek amacıyla bir dizi veya başka bir veri saklama yapısı kullanırdı; böylece, çok fazla sayıda "çalışan" nesnesine yer sağlanabilirdi.

"Soyut" Temel Sınıf

Dikkat ederseniz, `employee` temel sınıfına ait hiçbir nesne tanımlamıyoruz. Bu sınıfı genel bir sınıf olarak kullanıyoruz; öyle ki, bu sınıfın tek amacı, diğer sınıfların türetilebilecekleri bir temel gibi davranmak.

`laborer` sınıfı, ilave veri veya fonksiyon içermediğinden `employee` sınıfı ile aynı şekilde işlev görür. `laborer` sınıfı gereksizmiş gibi görünebilir; fakat, bunu ayrı bir sınıf yaparak tüm sınıfların aynı kaynaktan, `employee`'den geldiğini vurguluyoruz. Ayrıca, ileride `laborer` sınıfını değiştirmeye karar verirsek `employee` sınıfının deklarasyonunu değiştirmemize gerek kalmayacaktır.

Diğer sınıfları türetmek için kullanılan sınıflar, `EMPLOY`'daki `employee` gibi, kimi zaman kabaca *soyut* (abstract) *sınıflar* olarak adlandırılır. Bu sınıfların soyut olarak adlandırılmasının sebebi, bu sınıflara ait gerçek örnek kopyalar (nesneler) tanımlanmıyor olmasıdır. Bununla beraber, *soyut* terimi çok daha kesin bir tanıma sahiptir. Bunu "Sanal Fonksiyonlar" adlı Bölüm 11'de ele alacağız.

Kurucu ve Üye Fonksiyonlar

Temel sınıfta veya türetilmiş sınıfların hiç birinde kurucu fonksiyon yok; bu nedenle, derleyici aşağıdaki gibi bir tanım ile karşılaşınca `manager`'ın varsayılan kurucu fonksiyonunu kullanarak `employee` için varsayılan kurucu fonksiyonu çağırmak yoluyla çeşitli sınıfların nesnelerini otomatik olarak tanımlar.

```
manager m1,m2;
```

`employee` içindeki `getdata()` ve `putdata()` fonksiyonları kullanıcıdan bir isim ve sayı alır ve bir isim ve sayıyı ekranda gösterir. `manager` ve `scientist` sınıflarındaki yine `getdata()` ve `putdata()` olarak adlandırılan fonksiyonlar, hem `employee` içindeki fonksiyonları kullanırlar hem de kendi işlerini yaparlar. `manager` sınıfında `getdata()` fonksiyonu kullanıcıya unvanı ve golf kulübü aidat ödemesinin miktarını sorar; `putdata()` bu değerleri ekranda görüntüler. `scientist` sınıfında bu fonksiyonlar yayınların sayısını ele alırlar.

Kalıtım ve Grafik Şekiller

"Nesneler ve Sınıflar" adlı Bölüm 6'da `CIRCLES` programını görmüştük. Bu program, ekranda görüntülenebilen grafik daireleri simgeleyen bir sınıf içeriyordu. Elbette dairelerin yanı sıra başka şekil çeşitleri de mevcuttur: Kare, üçgen vs. "Başka şekil çeşitleri" sözcük öbeğinin kendisi bile "şekil" denilen bir şey ile belirli bir şekil türü (daire, kare gibi) arasında bir kalıtım ilişkisini ima eder. Değişik şekillerin birbiriyle alakasız olarak ele alındığı bir programa kıyasla daha dengeli ve anlaşılması daha kolay bir program geliştirmek için bu ilişkiyi kullanabiliriz.

Özel olarak, üç türetilmiş sınıfın temel sınıfı (ebeveyn) olan bir `shape` sınıfı geliştireceğiz. Türetilmiş sınıflar şöyledir: `circle` sınıfı, `rect` sınıfı (dikdörtgen için) ve `tria` sınıfı (üçgen için). Console Graphics Lite fonksiyonlarını kullanan diğer programlarda olduğu gibi, "Console Graphics Lite" adlı Ek E bölümünü okumanız gerekebilir. Ayrıca, belirli bir derleyici kullanarak grafik dosyalarını programlarınıza nasıl kuracağınızı öğrenmek için "Microsoft Visual C++" adlı Ek C bölümüne veya "Borland C++ Builder" adlı Ek D bölümüne bakmanız gerekebilir. `MULTSHAP` programının listesi şöyledir:

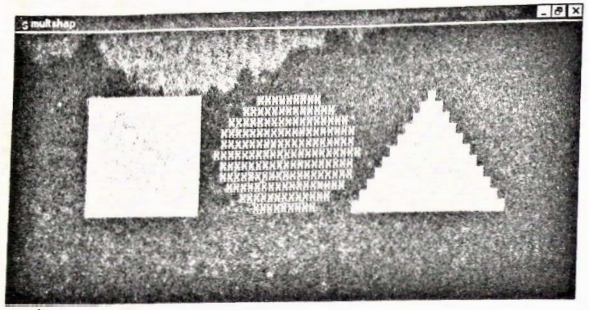
```
// multshap.cpp
// toplar, dikdortgenler ve cokgenler
#include "msoftcon.h"                    //grafik fonksiyonlari icin
////////////////////////////////////////////////////////////////
class shape                              //temel sinif
   {
   protected:
      int xCo, yCo;                      //seklin koordinatlari
      color fillcolor;                   //renk
      fstyle fillstyle;                  //dolgu deseni
   public:                               //argumansiz kurucu fonksiyon
      shape() : xCo(0), yCo(0), fillcolor(cWHITE),
                                fillstyle(SOLID_FILL)
         { }                             //4-argumanli kurucu fonksiyon
      shape(int x, int y, color fc, fstyle fs) :
                  xCo(x), yCo(y), fillcolor(fc), fillstyle(fs)
         { }
      void draw()const                   //renk ve dolgu desenini ayarla
         {
         set_color(fillcolor);
         set_fill_style(fillstyle);
         }
   };
////////////////////////////////////////////////////////////////
class circle : public shape
   {
   private:
      int radius;
   public:                               //((xCo, yCo): dairenin merkezidir
      circle() : shape()                 //argumansiz kurucu fonksiyon
```



```
         { }
      circle(int x, int y, int r, color fc, fstyle fs)
                                 //5-argumanli kurucu fonksiyon
                              : shape(x, y, fc, fs), radius(r)
         { }
      void draw() const                  //daireyi ciz
         {
         shape::draw();
         draw_circle(xCo, yCo, radius);
         }
   };
////////////////////////////////////////////////////////////////
class rect : public shape
   {
   private:
      int width, height;                 //(xCo, yCo): sol ust kosedir
   public:
      rect() : shape(), height(0), width(0)      //argumansiz kurucu fonksiyon
         { }                             //6-argumanli kurucu fonksiyon
      rect(int x, int y, int h, int w, color fc, fstyle fs) :
                           shape(x, y, fc, fs), height(h), width(w)
         { }
      void draw() const                  //dikdortgeni ciz
         {
         shape::draw();
         draw_rectangle(xCo, yCo, xCo+width, yCo+height);
         set_color(cWHITE);              //kosegeni ciz
         draw_line(xCo, yCo, xCo+width, yCo+height);
         }
   };
////////////////////////////////////////////////////////////////
class tria : public shape
   {
   private:
      int height;                        //(xCo, yCo): piramidin tepesidir
   public:
      tria() : shape(), height(0)        //argumansiz kurucu fonksiyon
         { }                             //5-argumanli kurucu fonksiyon
      tria(int x, int y, int h, color fc, fstyle fs) :
                                    shape(x, y, fc, fs), height(h)
         { }
      void draw() const                  //ucgeni ciz
         {
         shape::draw();
         draw_pyramid(xCo, yCo, height);
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   init_graphics();                      //grafik sistemini ilk kullanima hazirla

   circle cir(40, 12, 5, cBLUE, X_FILL);          //daireyi olustur
   rect rec(12, 7, 10, 15, cRED, SOLID_FILL);     //dikdortgeni olustur
   tria tri(60, 7, 11, cGREEN, MEDIUM_FILL);      //ucgeni olustur

   cir.draw();                                    //tum sekilleri ciz
   rec.draw();
   tri.draw();
```

```
                                        //imleci sol alt koseye tasi
    set_cursor_pos(1, 25);
    return 0;
    }
```

Bu program çalıştırıldığı zaman üç değişik şekil üretir: Mavi bir daire, kırmızı bir dikdörtgen ve yeşil bir üçgen. Programın çıktısı Şekil 9.6'da gösteriliyor.



ŞEKİL 9.6: MULTSHAP programının çıktısı.

Tüm şekillerde ortak olan özellikler, örneğin konum, renk ve dolgu deseni, **shape** sınıfının içine yerleştirilir. Şekiller ayrı ayrı daha spesifik nitelikler de içerirler. Örneğin, bir dairenin bir yarıçapı; bir dikdörtgenin ise eni ve boyu vardır. **shape** içinde yer alan bir **draw()** rutini şekillerin tümüne özgü görevleri idare eder: Renklerini ve dolgu desenlerini ayarlar. **circle**, **rect** ve **tria** sınıfları içindeki aşırı yüklenmiş **draw()** fonksiyonları bir kez renk ve desen belirlendikten sonra spesifik şekilleri çizme işlemini yürütürler.

Son örnekte olduğu gibi, burada da **shape** temel sınıfı, soyut sınıfa bir örnektir; çünkü, bu sınıfa ait bir nesnenin örnek kopyasını oluşturmanın bir manası yoktur. Bir **shape** nesnesi ekranda nasıl bir şekil gösterir? Bu soru çok anlamsızdır. Sadece belirli bir şekil kendisini görüntüleyebilir. **shape** sınıfı sadece, tüm sınıflarda ortak olan niteliklerin ve etkinliklerin bir deposu olarak kullanılmak üzere mevcuttur.

Public ve Private Kalıtım

C++, sınıf üyelerine erişimi ustaca ayarlamak için çok sayıda yöntem sunar. Bu tür erişim kontrol mekanizmalarından biri, türetilmiş sınıfların deklarasyon şekillerine dayanır. Şu ana kadarki örneklerimizde public olarak türetilebilen sınıflar kullandık. **EMPLOY** programındaki şu deklarasyon da buna bir örnektir:

```
class manager : public employee
```

Bu ifadedeki **public** anahtar kelimesinin etkisi nedir ve alternatifleri nelerdir? Dikkatle okuyun: **public** anahtar kelimesi, türetilmiş sınıfın nesnelerinin temel sınıfın public üye fonksiyonlarına erişmelerinin mümkün olduğunu belirtir. Bunun alternatifi **private** anahtar kelimesidir. Bu anahtar kelime kullanıldığında türetilmiş sınıfın nesneleri, temel sınıfın public

üye fonksiyonlarına erişemezler. Nesneler bir sınıfın **private** veya **protected** üyelerine asla erişemeyeceklerine göre sonuç, temel sınıfın hiçbir elemanı türetilmiş sınıfın nesneleri tarafından erişilemez, şeklindedir.

Erişim Kombinasyonları

Erişim için o kadar çok ihtimal söz konusudur ki, neyin çalışıp neyin çalışmadığını gösteren bir örnek programa bir göz atmak çok daha öğretici olacaktır. İşte **PUBPRIV** programının listesi:

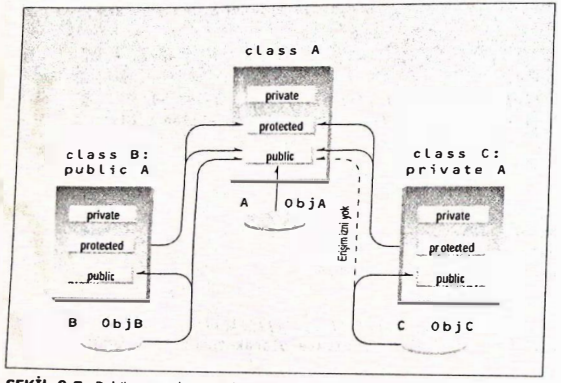
```
// pubpriv.cpp
// public ve private olarak turetilen siniflari test eder
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class A                                 //temel sinif
   {
   private:
      int privdataA;                    // (fonksiyonlar burada gosterilen
   protected:                           //verilerle ayni erisim kurallarina sahiptir)
      int protdataA;
   public:
      int pubdataA;
   };
////////////////////////////////////////////////////////////////
class B : public A                      //public olarak turetilen sinif
   {
   public:
      void funct()
         {
         int a;
         a = privdataA;                 //hata: erisilemez
         a = protdataA;                 //tamam
         a = pubdataA;                  //tamam
         }
   };
////////////////////////////////////////////////////////////////
class C : private A                     //private olarak turetilen sinif
   {
   public:
      void funct()
         {
         int a;
         a = privdataA;                 //hata: erisilemez
         a = protdataA;                 //tamam
         a = pubdataA;                  //tamam
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   int a;

   B objB;
   a = objB.privdataA;                  //hata: erisilemez
   a = objB.protdataA;                  //hata: erisilemez
   a = objB.pubdataA;                   //tamam (A, B'icin public)
```

```
C objC;
a = objC.privdataA;          //hata: erisilemez
a = objC.protdataA;          //hata: erisilemez
a = objC.pubdataA;           //hata: erisilemez (A, C icin private)
return 0;
}
```

Bu program, bir `A` temel sınıfı ve bu sınıfın içinde private, protected ve public veri öğeleri tanımlar. `B` ve `C`, adlı iki sınıf, `A` sınıfından türetilir. `B` public olarak, `C` ise private olarak türetilir.

Önceden gördüğümüz gibi, türetilmiş sınıfın içindeki fonksiyonlar temel sınıfın protected ve public verilerine erişebilirler. Türetilmiş sınıfın nesneleri temel sınıfın private veya protected üyelerine erişmezler.

Burada yeni olan, public ve private olarak türetilen sınıflar arasındaki farktır. Public olarak türetilen `B` sınıfının nesneleri, `A` temel sınıfının public üyelerine erişebilirken, private olarak türetilen `C` sınıfının nesneleri bu üyelere erişemezler. `C` sınıfının nesneleri sadece kendilerinin ait olduğu türetilmiş sınıfın public üyelerine erişebilirler. Şekil 9.7'de bu durum açıklanıyor.



ŞEKİL 9.7: Public ve private erişimlerin türetilmesi.

Bir sınıf tanımlarken erişim türünü açıkça belirtmezseniz, erişimin **private** olduğu varsayılır.

Erişim Belirteçleri: Ne Zaman Ne Kullanmalı?

Public kalıtıma karşı private kalıtımı ne zaman kullanacağınıza nasıl karar verirsiniz? Temel sınıfın geliştirilmiş – veya daha da özelleştirilmiş- bir versiyonunu sunan bir türetilmiş sınıf bir çok durumda mevcuttur. Bu tür türetilmiş sınıf örneklerini görmüştük (söz gelişi, **Counter** sınıfına eksiltme operatörünü ekleyen **CountDn** sınıfı ve **employee** sınıfının daha özelleştirilmiş versiyonu olan **manager** sınıfı). Bu tür durumlarda, türetilmiş sınıfın nesnelerinin temel işlemleri gerçekleştirmek amacıyla temel sınıfın public fonksiyonlarına; türetilmiş sınıfın sunduğu daha özel işlemleri gerçekleştirmek için de türetilmiş sınıfın fonksiyonlarına erişmeleri mantıklıdır. Bu tür durumlarda erişimin public olarak türetilmesi uygundur.

Bununla beraber, kimi durumlarda türetilmiş sınıf, temel sınıfın orijinal arayüzünü saklayarak veya farklılaştırarak temel sınıfın işleyişini tamamen değiştirecek bir yöntem olarak tanımlanır. Örneğin, `Array` isminde gerçekten hoş bir sınıf tanımladığınızı hayal edin. `Array` sınıfı bir dizi gibi davranır, ama dizi indekslerinin dizi dışına taşmasına karşı bir de koruma sağlar. Sonra varsayalım, `Stack` sınıfının temeli olarak, temel bir dizi tipi kullanmaktansa, bu `Array` sınıfını kullanmak istiyorsunuz. `Stack`'i, `Array`'den türetebilirsiniz; fakat, `Stack` nesnesini kullananların, bu nesneleri birer dizi gibi ele almalarını istemeyebilirsiniz. Örneğin, [] operatörünü kullanarak veri öğelerine erişmelerini istemeyebilirsiniz. `Stack`'in nesneleri, `Push()` ve `pop()` fonksiyonları kullanılarak, her zaman birer yığın gibi ele alınmalıdır. Yani, aslında `Array` sınıfını bir `Stack` sınıfı gibi göstermek istiyorsunuz. Bu durumda, erişimin özel olarak türetilmesi, `Array` sınıfının fonksiyonlarının tümünü, türetilmiş `Stack` sınıfının nesnelerinden gizlemenize imkan verir.

Kalıtımın Seviyeleri

Sınıflar, kendileri de türetilmiş olan sınıflardan türetilebilir. Bu görüşü sunan işte mini bir program:

```
class A
   { };
class B : public A
   { };
class C : public B
   { };
```

Burada `B`, `A`'dan ve `C` de `B`'den türetiliyor. Bu işlem istenilen seviyeye kadar genişletilebilir – `D` de `C`'den türetilebilir vs.

Biraz daha somut bir örnek olarak, `EMPLOY` programına *foreman* adında özel bir tür işçi eklemeye karar verdiğimizi farz edelim. `foreman`'ın nesnelerinin dahil edildiği `EMPLOY2` adında yeni bir program geliştirelim.

Bir ustabaşı (foreman) da bir tür işçi (laborer) olduğuna göre, `foreman` sınıfı Şekil 9.8'de gösterildiği gibi `laborer` sınıfından türetilir.

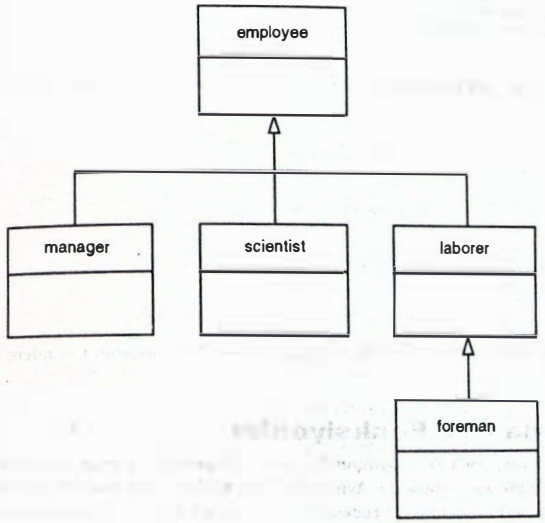
Ustabaşılar, bir grup işçiyi denetleyerek yedek parça baskı işlemine göz kulak olurlar. Ustabaşılar, kendi gruplarında yedek parça üretim kotasından sorumludurlar. Bir ustabaşının becerisi, başarıyla tamamlanan üretim kotalarının yüzdesi ile ölçülür. `foreman` sınıfındaki `quotas` veri öğesi bu yüzdeyi simgeler. `EMPLOY2`'nin program listesi şöyledir:

```
// employ2.cpp
// kalitimin birkac seviyesi
#include <iostream>
using namespace std;
const int LEN = 80;                    //isimlerin maksimum uzunlugu
////////////////////////////////////////////////////////////////
class employee
   {
   private:
      char name[LEN];                  //calisanin ismi
      unsigned long number;            //calisanin numarasi
   public:
      void getdata()
         {
```

```
            cout << "\n Enter last name: "; cin >> name;
            cout << " Enter number: "; cin >> number;
            }
         void putdata() const
            {
            cout << "\n   Name: " << name;
            cout << "\n   Number: " << number;
            }
      };
   ////////////////////////////////////////////////////////////
   class manager : public employee        //yonetici sinifi
      {
      private:
         char title[LEN];                 //"baskan yardimcisi" vs.
         double dues;                     //golf kulubu aidat odemeleri
      public:
         void getdata()
            {
            employee::getdata();
            cout << "   Enter title: "; cin >> title;
            cout << "   Enter golf club dues: "; cin >> dues;
            }
         void putdata() const
            {
            employee::putdata();
            cout << "\n   Title: " << title;
            cout << "\n   Golf club dues: " << dues;
            }
   };
   ////////////////////////////////////////////////////////////
   class scientist : public employee      //bilim adami sinifi
      {
      private:
         int pubs;                        //yayinlarin sayisi
      public:
         void getdata()
            {
            employee::getdata();
            cout << "   Enter number of pubs: "; cin >> pubs;
            }
         void putdata()const
            {
            employee::putdata();
            cout << "\n   Number of publications: " << pubs;
            }
   };
   ////////////////////////////////////////////////////////////
   class laborer : public employee        //isci sinifi
      {
      };
   ////////////////////////////////////////////////////////////
   class foreman : public laborer         //ustabasi sinifi
      {
      private:
         float quotas;                    //kota yuzdesine basariyla ulasildi
      public:
         void getdata()
            {
            laborer::getdata();
```



```
            cout << "   Enter quotas: "; cin >> quotas;
            }
         void putdata() const
            {
            laborer::putdata();
            cout << "\n   Quotas: " << quotas;
            }
      };
   ////////////////////////////////////////////////////////////
   int main()
      {
      laborer l1;
      foreman f1;

      cout << endl;
      cout << "\nEnter data for laborer 1";
      l1.getdata();
      cout << "\nEnter data for foreman 1";
      f1.getdata();

      cout << endl;
      cout << "\nData on laborer 1";
      l1.putdata();
      cout << "\nData on foreman 1";
      f1.putdata();
      cout << endl;
      return 0;
      }
```

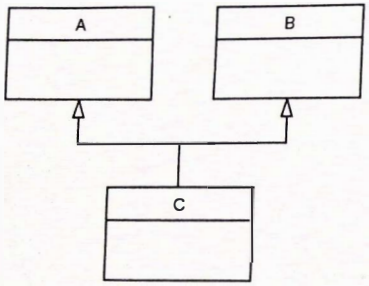


ŞEKİL 9.8: EMPLOY2 için UML sınıf şeması.

Bir sınıf hiyerarşisinin bir organizasyon şeması ile aynı olmadığına dikkat edin. Organizasyon şeması komuta zincirini gösterir. Sınıf hiyerarşisi ise ortak özelliklerin genelleştirilmesiyle elde edilir. Sınıf ne kadar genelse, şemada o kadar yukarıda yer alır. Yani, bir işçi bir ustabaşından daha geneldir. Ustabaşı özel bir tür işçidir. Bu nedenle, her ne kadar bir ustabaşı bir işçiden daha fazla kazanıyor olsa da, sınıf hiyerarşisinde `laborer`, `foreman`'ın üstünde gösterilmiştir.

Çoklu Kalıtım

Bir sınıf birden fazla temel sınıftan türetilebilir. Buna *çoklu kalıtım* (multiple inheritance) denir. Şekil 9.9, bir `C` sınıfı `A` ve `B` sınıflarından türetildiğinde bunun nasıl göründüğünü açıklıyor.



ŞEKİL 9.9: Çoklu kalıtım için UML sınıf şeması.

Çoklu kalıtımın söz dizimi, tek kalıtımın söz dizimiyle aynıdır. Şekil 9.9'da gösterilen ilişki şu şekilde ifade edilir:

```
class A                              //base class A
   {
   };
class B                              //base class B
   {
   };
class C : public A, public B         //C , A ve B'den turetilir
   {
   };
```

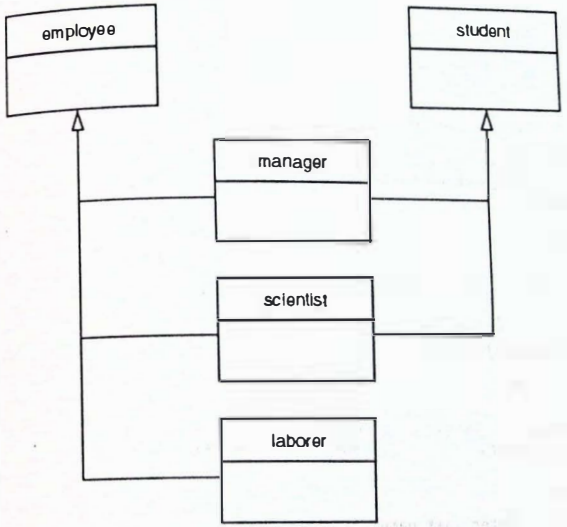
`C`'nin türetildiği temel sınıflar, `C`'nin spesifikasyonunda iki nokta işaretinin peşinden, virgül ile ayrılarak listelenir.

Çoklu Kalıtımda Üye Fonksiyonlar

Çoklu kalıtıma örnek olarak, `EMPLOY` programında, bazı çalışanların eğitim deneyimlerini kaydetmek zorunda olduğumuzu varsayalım. Ayrıca, belki farklı bir proje için, değişik eğitim geçmişlerine sahip öğrencileri modelleyen `student` isminde bir sınıfı zaten geliştirmiş olduğumuzu varsayalım. Eğitim verilerini dahil etmek için `employee` sınıfını değiştirmek yerine, bu verileri çoklu kalıtım yoluyla `student` sınıfından eklemeye karar veriyoruz.

`student` sınıfı, en son mezun olunan okul veya üniversite ismini ve alınan en yüksek dereceyi saklar. Bu bilgilerin her ikisi de karakter katarı olarak saklanır. `getedu()` ve `putedu()` isimli iki üye fonksiyon, kullanıcıdan bu bilgileri alır ve ekranda gösterir.

Eğitim bilgileri çalışan sınıflarının hepsini ilgilendirmez. Şimdi, biraz demokratik olmayan bir şekilde, işçilerin eğitim deneyimlerini kaydetmek zorunda olmadığımızı varsayalım. Bu bilginin sadece yöneticiler ve bilim adamlarıyla ilgili olduğunu düşünelim. Bu nedenle, `manager` ve `scientist` sınıflarını Şekil 9.10'da gösterildiği gibi değiştiriyoruz. Böylece, bu sınıflar hem `employee` hem de `student` sınıflarından devralabilirler.



ŞEKİL 9.10: `EMPMULT` için UML sınıf şeması.

İşte, bu ilişkileri gösteren (fakat, bunun haricinde her şeyi dışarıda bırakan) küçük bir program:

```
class student
   { };
class employee
   { };
class manager :private employee,private student
   { };
class scientist :private employee,private student
   { };
class laborer :public employee
   { };
```

Burada da, hatırı sayılır ölçüde ayrıntılı olmasıyla göze çarpan **EMPMULT** programının listesi görülüyor:

```
// empmult.cpp
// calisanlar ve derecelerle coklu kalitim
#include <iostream>
using namespace std;
const int LEN = 80;                         //isimlerin maksimum uzunlugu
//////////////////////////////////////////////////////////////
class student                               //egitim gecmisi
   {
   private:
      char school[LEN];                     //okul veya universitenin ismi
      char degree[LEN];                     //kazanilan en yuksek derece
   public:
      void getedu()
         {
         cout << "   Enter name of school or university: ";
         cin >> school;
         cout << "   Enter highest degree earned \n";
         cout << "   (Highschool, Bachelor 's, Master 's, PhD): ";
         cin >> degree;
         }
      void putedu() const
         {
         cout << "\n   School or university: " << school;
         cout << "\n   Highest degree earned: " << degree;
         }
   };
//////////////////////////////////////////////////////////////
class employee
   {
   private:
      char name[LEN];                       //calisanin ismi
      unsigned long number;                 //calisanin numarasi
   public:
      void getdata()
         {
         cout << "\n   Enter last name: "; cin >> name;
         cout << "   Enter number: "; cin >> number;
         }
      void putdata() const
         {
         cout << "\n   Name: " << name;
         cout << "\n   Number: " << number;
         }
   };
//////////////////////////////////////////////////////////////
class manager : private employee, private student        //yonetim
   {
   private:
      char title[LEN];
      double dues;                          //"baskan yardimcisi" vs.
   public:                                  //golf kulubu aidat odemeleri
      void getdata()
         {
         employee::getdata();
         cout << "   Enter title: "; cin >> title;
```



```
         cout << "   Enter golf club dues: "; cin >> dues;
         student::getedu();
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n   Title: " << title;
         cout << "\n   Golf club dues: " << dues;
         student::putedu();
         }
   };
//////////////////////////////////////////////////////////////
class scientist : private employee, private student      //bilim adami
   {
   private:
      int pubs;                             //yayinlarin sayisi
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter number of pubs: "; cin >> pubs;
         student::getedu();
         }
      void putdata() const
         {
         employee::putdata();
         cout <<"\n Number of publications:" <<pubs;
         student::putedu();
         }
   };
//////////////////////////////////////////////////////////////
class laborer : public employee          //isci
   {
   };
//////////////////////////////////////////////////////////////
int main()
   {
   manager m1;
   scientist s1, s2;
   laborer l1;

   cout << endl;
   cout << "\nEnter data for manager 1 ";  //birkac calisan icin
   m1.getdata();                           //verileri al

   cout << "\nEnter data for scientist 1 ";
   s1.getdata();
   cout << "\nEnter data for scientist 2 ";

   s2.getdata();
   cout << "\nEnter data for laborer 1 ";
   l1.getdata();

   cout << "\nData on manager 1 ";         //birkac calisanla ilgili
   m1.putdata();                           //verileri ekranda goster

   cout << "\nData on scientist 1 ";
   s1.putdata();
```

```
    cout << "\nData on scientist 2 ";
    s2.putdata();

    cout << "\nData on laborer 1 ";
    l1.putdata();
    cout << endl;
    return 0;
    }
```

`manager` ve `scientist` sınıflarındaki `getdata()` ve `putdata()` fonksiyonları, `student` sınıfındaki fonksiyonlara çağrılar içerir. Örneğin,

```
student::getedu();
```

ve

```
student::putedu();
```

Bu rutinlere `manager` ve `scientist` sınıflarının içinden erişilebilir, çünkü bu sınıflar `student` sınıfından gelir.

`EMPMULT` programı ile bazı etkileşim örnekleri şu şekildedir:

```
Enter data for manager 1
   Enter last name: Bradley
   Enter number: 12
   Enter title: Vice-President
   Enter golf club dues: 100000
   Enter name of school or university: Yale
   Enter highest degree earned
   (Highschool,Bachelor's,Master's,PhD): Bachelor's
Enter data for scientist 1
   Enter last name: Twilling
   Enter number: 764
   Enter number of pubs: 99
   Enter name of school or university: MIT
   Enter highest degree earned
   (Highschool,Bachelor's,Master's,PhD): PhD
Enter data for scientist 2
   Enter last name: Yang
   Enter number: 845
   Enter number of pubs: 101
   Enter name of school or university: Stanford
   Enter highest degree earned
   (Highschool,Bachelor's,Master's,PhD): Master's
Enter data for laborer 1
   Enter last name: Jones
   Enter number: 48323
```

`EMPLOY` ve `EMPLOY2` örneklerinde gördüğümüz gibi, program bu bilgileri aşağı yukarı aynı biçimde ekranda gösterir.

EMPMULT Programında private Erişim Türetmek

`EMPMULT` programındaki `manager` ve `scientist` sınıfları, `employee` ve `student` sınıflarından private olarak türetilir. Public olarak türetmeye gerek yoktur; çünkü, `manager` ve `scientist` nesneleri, `employee` ve `student` temel sınıflarındaki rutinleri hiçbir zaman çağırmazlar. Bununla birlikte, `laborer` sınıfının kendisine ait hiç üye fonksiyonu olmadığı ve `employee` sınıfındakileri kullandığı için `laborer` sınıfının `employee` sınıfından public olarak türetilmesi gerekir.

Çoklu Kalıtımda Kurucu Fonksiyonlar

`EMPMULT` programında hiç kurucu fonksiyon yoktur. Şimdi, gerçekten kurucu fonksiyon kullanan bir örneğe bakalım ve kurucu fonksiyonların çoklu kalıtımda nasıl ele alındığını görelim.

Müteahhitler için bir program yazdığımızı ve bu programın kereste stok parçalarını modellediğini hayal edelim. Program, belirli bir tipteki kerestenin miktarını simgeleyen bir sınıf kullanır. Örneğin, 100 tane 8 ayak uzunluğunda 2x4'lük yapı sınıfı.

Sınıf, bu tür kereste parçalarının her biri için çeşitli türde veri saklamalıdır. Bu tür kereste parçalarının uzunluğunu bilmemiz gerekir (örneğin, 3'-6"). Ayrıca, kereste parçalarının sayısını ve birim maliyetini de saklamamız gerekir.

Üstelik, söz konusu kerestenin tanımını da saklamalıyız. Bu iki parçadan oluşur: Birinci parça, kereste kesitinin nominal boyutlarıdır. Bu inç cinsinden verilir. Örneğin, 2 inçe 4 inçlik bir kereste (siz metrik arkadaşlar için bu yaklaşık 5 cm'e 10 cm demektir) *ikiye-dört* olarak adlandırılır. Bu da genellikle 2x4 şeklinde yazılır. Ayrıca, kerestenin cinsini de bilmeye ihtiyacımız var – ham kesim, yapı tipi, dört yüzü kaplanmış vs. Bu veriyi tutmak için bir `Type` adlı sınıfı tanımlamayı uygun bulduk. Bu sınıf, nominal boyutlar ve kerestenin cinsi için üye veriler içerir. Bu üye verilerin her ikisi de karakter katarı olarak ifade edilir; örneğin, 2x6 ve *construction* (yapı tipi kereste) gibi. Üye fonksiyonlar bu bilgileri kullanıcıdan alırlar ve ekranda görüntülerler.

Uzunluğu saklamak için önceki örneklerden tanıdığımız `Distance` sınıfını kullanacağız. En son olarak, hem `Type` sınıfından hem de `Distance` sınıfından devralan bir `Lumber` sınıfı tanımlanır. `ENGLMULT` programının listesi işte şöyledir:

```
// englmult.cpp
// Ingiliz olcu sistemine dayanan Distance sinifi ile coklu kalitim
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class Type                              //kerestenin tipi
   {
   private:
      string dimensions;
      string grade;
   public:                              //argumansiz kurucu fonksiyon
      Type() : dimensions("N/A"), grade("N/A")
         { }
                                        //2-argumanli kurucu fonksiyon
      Type(string di, string gr) : dimensions(di), grade(gr)
         { }
      void gettype()                    //tipi kullanicidan al
         {
         cout << "   Enter nominal dimensions (2x4 etc.): ";
```

```
            cin >> dimensions;
            cout << "   Enter grade (rough,const,etc.): ";
            cin >> grade;
            }
         void showtype() const              //tipi ekranda goster
            {
            cout << "\n   Dimensions: " << dimensions;
            cout << "\n   Grade: " << grade;
            }
      };
   ////////////////////////////////////////////////////////////
   class Distance          //Ingiliz uzaklik olculerine dayanan Distance sinifi
      {
      private:
         int feet;
         float inches;
      public:                               //argumansiz kurucu fonksiyon
         Distance() : feet(0), inches(0.0)
            { }                             //kurucu fonksiyon (iki argumanli)
         Distance(int ft, float in) : feet(ft), inches(in)
            { }
         void getdist()                     //uzunlugu kullanicidan al
            {
            cout << "   Enter feet: "; cin >> feet;
            cout << "   Enter inches: "; cin >> inches;
            }
         void showdist() const              //uzakligi ekranda goster
            { cout << feet << "\'-" << inches << '\"'; }
      };
   ////////////////////////////////////////////////////////////
   class Lumber : public Type, public Distance
      {
      private:
         int quantity;                      //parcalarin sayisi
         double price;                      //her parcanin fiyati
      public:                               //kurucu fonksiyon (argumansiz)
         Lumber() : Type(), Distance(), quantity(0), price(0.0)
            { }
                                            //kurucu fonksiyon (6 argumanli)
         Lumber( string di, string gr,      //Type icin argumanlar
                 int ft, float in,          //Distance icin argumarlar
                 int qu, float prc ) :      //kendi verilerimiz icin argumanlar
                 Type(di, gr),              //Type kurucu fonksiyonunu cagir
                 Distance(ft, in),          //Distance kurucu fonksiyonunu cagir
                 quantity(qu), price(prc) //kendi verilerimize ilk degeri ata
            { }
         void getlumber()
            {
            Type::gettype();
            Distance::getdist();
            cout << "   Enter quantity: "; cin >> quantity;
            cout << "   Enter price per piece: "; cin >> price;
            }
         void showlumber() const
            {
            Type::showtype();
            cout << "\n   Length: ";
            Distance::showdist();
            cout << "\n   Price for " << quantity
```



```
                 << " pieces: $" << price * quantity;
            }
      };
   ////////////////////////////////////////////////////////////
   int main()
      {
      Lumber siding;                        //kurucu fonksiyon (argumansiz)

      cout << "\nSiding data:\n";
      siding.getlumber();                   //siding verilerini kullanicidan al

                                            //kurucu fonksiyon (6 argumanli)
      Lumber studs( "2x4", "const", 8, 0.0 ,200 ,4.45F );

                                            //kereste verilerini ekranda goster
      cout << "\nSiding";  siding.showlumber();
      cout << "\nStuds";   studs.showlumber();
      cout << endl;
      return 0;
      }
```

Bu programdaki başlıca yeni özellik, türetilmiş sınıf **Lumber**'ın içinde kurucu fonksiyonların kullanılmış olmasıdır. Bu kurucu fonksiyonlar **Type** ve **Distance** içinde bulunan uygun kurucu fonksiyonları çağırırlar.

Argümansız Kurucu Fonksiyon

Type içindeki argümansız kurucu fonksiyon şöyle görünür:

```
Type()
   { strcpy(dimensions, "N/A"); strcpy(grade, "N/A"); }
```

Bu kurucu fonksiyon **dimensions** ve **grade** değişkenlerini "N/A" (kullanılamaz) değeri ile doldurur. Böylece, ilk değeri verilmemiş bir kereste nesnesinin verileri görüntülenmeye çalışıldığında kullanıcı bu durumdan haberdar edilmiş olur.

Argümansız kurucu fonksiyonlara zaten **Distance** sınıfından aşinasınız:

```
Distance() : feet(0), inches(0.0)
   { }
```

Lumber'ın içindeki argümansız kurucu fonksiyon, bu kurucu fonksiyonların her ikisini de çağırır.

```
Lumber() : Type(), Distance(), quantity(0), price(0.0)
   { }
```

Temel sınıfın kurucu fonksiyonlarının isimleri, iki nokta işaretinin peşinden gelir ve birbirinden virgülle ayrılır. **Lumber()** kurucu fonksiyonu çağrıldığı zaman, temel sınıfın bu kurucu fonksiyonları – **Type()** ve **Distance()** – çalıştırılacaktır. Bu sayede, **quantity** ve **price** nitelikleri de başlangıç değerlerine atanmış olur.

Birden Fazla Argüman İçeren Kurucu Fonksiyonlar

Type sınıfı için iki argümanlı kurucu fonksiyon şöyledir:

```
Type(string di, string gr) : dimensions(di), grade(gr)
   { }
```

Bu kurucu fonksiyon, karakter katarı argümanları, boyut ve kereste cinsini gösteren üye veri öğelerine kopyalar.

Yine önceki programlardan tanıdık olan **Distance** için kurucu fonksiyon şöyledir:

```
Distance(int ft, float in) : feet(ft), inches(in)
   { }
```

Lumber'ın kurucu fonksiyonu, bu kurucu fonksiyonların her ikisini de çağırır; bu nedenle, bu fonksiyonların argümanları için değer temin etmesi gerekir. İlaveten, kendisinin de iki argümanı vardır: Kerestenin miktarı ve birim fiyatı. Böylece, bu kurucu fonksiyon altı tane argüman içerir. **Lumber**'ın kurucu fonksiyonu, iki kurucu fonksiyona iki çağrı yapar, her bir çağrı iki argüman alır ve sonra kendi veri öğelerine ilk değerleri atar. Bu kurucu fonksiyon şöyle görünür:

```
Lumber( string di, string gr,    //Type icin argumanlar
        int ft, float in,        //Distance icin argumanlar
        int qu, float prc ) :    //kendi verilerimiz icin argumanlar
        Type(di, gr),            //Type kurucu fonksiyonunu cagir
        Distance(ft, in),        //Distance kurucu fonksiyonunu cagir
        quantity(qu), price(prc) //kendi verilerimize ilk deger ata
    { }
```

Çoklu Kalıtımda Belirsizlik

Çoklu kalıtım içeren belli durumlarda garip problemler ortaya çıkabilir. Çok yaygın bir tanesi şöyledir: Diyelim ki, iki temel sınıfın fonksiyonlarının isimleri aynı; bu temel sınıfların her ikisinden türetilen bir türetilmiş sınıfın bu isimlerde hiç fonksiyonu yok. Türetilmiş sınıfın nesneleri, doğru temel sınıfın fonksiyonlarına nasıl ulaşabilecekler? Derleyici iki fonksiyondan hangisinin kast edildiğini anlayamayacağı için fonksiyon ismi tek başına yeterli olmaz.

Bu durumu ortaya koyan **AMBIGU** programının listesi şöyledir:

```
// ambigu.cpp
// coklu kalitimda belirsizlik
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class A
   {
   public:
      void show()    { cout << "Class A\n"; }
   };
class B
   {
   public:
      void show()    { cout << "Class B\n"; }
   };
class C : public A, public B
   {
   };
////////////////////////////////////////////////////////////
int main()
   {
```



```
   C objC;                  //C sinifinin nesnesi
// objC.show();             //belirsiz -- derlenmeyecek
   objC.A::show();          //tamam
   objC.B::show();          //tamam
   return 0;
   }
```

Fonksiyonun içinde bulunduğu sınıfı açıkça belirtmek için kapsam çözünürlük operatörü kullanılmasıyla problem çözülür. Böylece aşağıdaki ifade, **show()** fonksiyonunun **A** sınıfı içinde bulunan versiyonunu kast eder:

```
objC.A::show();
```

Şu ifade ise **B** sınıfı içinde bulunan versiyonunu kullanır:

```
objC.B::show();
```

Stroustrup bu durumu, *belirsizliği ortadan kaldırmak* (*disambiguation*) olarak adlandırır.

Bir başka tür belirsizlik, her biri aynı sınıftan türetilmiş iki ayrı sınıftan, bir sınıf türetmek istediğinizde ortaya çıkar. Bu durum baklava biçimde bir kalıtım ağacına neden olur. **DIAMOND** programı bunun nasıl göründüğünü ortaya koyuyor.

```
// diamond.cpp
// baklava seklinde coklu kalitim
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class A
   {
   public:
      void func();
   };
class B : public A
   { };
class C : public A
   { };
class D : public B, public C
   { };
////////////////////////////////////////////////////////////
int main()
   {
   D objD;
   objD.func();       //belirsiz:derlenmeyecek
   return 0;
   }
```

B ve **C** sınıflarının her ikisi de **A** sınıfından türetilir; **D** sınıfı ise çoklu kalıtım yoluyla hem **B** hem **C**'den türetilir. Eğer **D** sınıfının bir nesnesinden **A** sınıfının bir üye fonksiyonuna erişmeye çalışırsanız problem başlar. Bu örnekte, **objD func()** fonksiyonuna erişmeye çalışır. Bununla birlikte, **B** ve **C**'nin her ikisi de **A**'dan kalıtım yoluyla aldıkları **func()**'ın birer kopyasını içerir. Bu durumda, derleyici hangi kopyanın kullanılacağına karar veremez ve hatalı bir durum olduğuna işaret eder.

Bu problemle başa çıkmanın geliştirilmiş çeşitli yöntemleri mevcuttur. Ancak, bu tür belirsizliklerin ortaya çıkabileceği gerçeği bir çok uzmanın, çoklu kalıtımın bütünüyle önlenmesini tavsiye etmelerine neden olur. Büyük ölçüde deneyiminiz yoksa önemli programlarda bunu kesinlikle kullanmamalısınız.

Birlik (Aggregation): Sınıf İçinde Sınıf

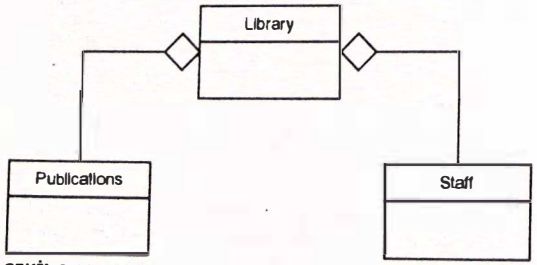
Birliği (aggregation) burada ele alacağız; çünkü, birlik doğrudan kalıtımla ilgili olmasa da, hem birlik hem de kalıtım, bağlantılardan (associations) çok daha özelleştirilmiş sınıf ilişkileridir. Bu ikisini karşılaştırmak ve farklılıklarını görmek öğretici olacaktır.

Eğer bir `B` sınıfı kalıtım yoluyla bir `A` sınıfından türetilmişse, "`B`, `A`'nın bir *türüdür*" diyebiliriz. Çünkü `B`, `A`'nın bütün özelliklerine sahiptir ve bunlara ek olarak birkaç da kendine ait özellik içerir. Bu, sığırcık bir kuş türüdür, demeye benzer: Bir sığırcığın tüm kuşlar tarafından paylaşılan özellikleri vardır (kanatlar, tüyler vs.); fakat, kendisine has bazı ayırt edici özellikleri de vardır (mesela, koyu parıldayan tüyler). Bu nedenle, kalıtım çoğu kez "tür" ("kind of") ilişkisi olarak adlandırılır.

Birlik ise "sahiplik" ("has a") ilişkisi olarak adlandırılır. Bir kütüphanenin kitaplara sahip olduğundan bahsederiz; ya da bir fatura, çeşitli kalemlerin alt alta sıralandığı bir listeye sahiptir. Birlik, ayrıca "parça-bütün" ("part-whole") ilişkisi olarak da adlandırılır: Kitap, kütüphanenin bir parçasıdır.

Nesne yönelimli programlamada, bir nesne diğer bir nesnenin bir niteliği ise bir birlikten söz edilebilir. İşte bir örnek: Bu örnekte, `A` sınıfının bir nesnesi, `B` sınıfının bir niteliği konumunda:

```
class A
   {
   };
class B
   {
   A objA;     //objA'yi A sinifinin bir nesnesi olarak tanimla
   };
```



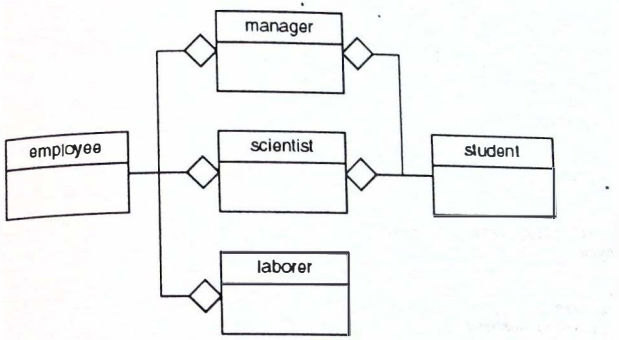
ŞEKİL 9.11: Birliği (aggregation) gösteren bir UML sınıf şeması.

UML dilinde birlik, bağlantının özel bir türü olarak ele alınır. Bazen bir bağlantının ayrıca bir birlik olup olmadığını söylemek zordur. Bir ilişkiyi bir bağlantı olarak adlandırmak her zaman emniyetlidir. Ancak, bir `A` sınıfı, `B` sınıfının nesnelerini içeriyorsa ve organizasyonel olarak `B` sınıfının üstünde yer alıyorsa, bu ilişki bir birlik için iyi adaydır. Bir şirket, çalışanlarının bütününe sahip olabilir veya bir pul koleksiyoncusu, pulların bütününe sahip olabilir.

Birlik, UML sınıf şemalarında bir bağlantı ile aynı biçimde gösterilir; yalnızca, bağlantı çizgisinin "bütün" ucunda (bütünü gösteren ucunda) açık (içi boş) baklava şeklinde ok ucu bulunur. Şekil 9.11 bunu gösterir.

EMPCONT Programında Birlik

`EMPMULT` programını kalıtım yerine birlik kullanarak yeniden düzenleyelim. `EMPMULT` programında `manager` ve `scientist` sınıfları, kalıtım ilişkisi kullanılarak `employee` ve `student` sınıflarından türetilir. `EMPCONT` isimli yeni programımızda `manager` ve `scientist` sınıfları, `employee` ve `student` sınıflarının örneklerini birer nitelik olarak içeriyorlar. Bu birlik ilişkisi Şekil 9.12'de gösteriliyor.



ŞEKİL 9.12: `EMPCONT` programının UML sınıf şeması.

Aşağıdaki küçük program bu ilişkileri farklı bir açıdan gösteriyor:

```
class student
   { };
class employee
   { };
class manager
   {
   student stu;   //stu, student sinifinin bir nesnesi
   employee emp;  //emp, employee sinifinin bir nesnesi
   };
class scientist
   {
   student stu;   //stu, student sinifinin bir nesnesi
   employee emp;  //emp, employee sinifinin bir nesnesi
   };
class laborer
   {
   employee emp;  //emp, employee sinifinin bir nesnesi
   };
```

EMPCONT'un program listesinin tamamı şöyledir:

```
// empcont.cpp
// calisanlar ve derecelerle aidiyet iliskisi
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class student                           //egitim gecmisi
   {
   private:
      string school;                    //okul veya universitenin ismi
      string degree;                    //kazanilan en yuksek derece
   public:
      void getedu()
         {
         cout << "   Enter name of school or university: ";
         cin >> school;
         cout << "   Enter highest degree earned \n";
         cout << "   (Highschool, Bachelor's, Master's, PhD): ";
         cin >> degree;
         }
      void putedu() const
         {
         cout << "\n   School or university: " << school;
         cout << "\n   Highest degree earned: " << degree;
         }
   };
////////////////////////////////////////////////////////////////
class employee
   {
   private:
      string name;                      //calisanin ismi
      unsigned long number;             //calisanin numarasi
   public:
      void getdata()
         {
         cout << "\n   Enter last name: "; cin >> name;
         cout << "   Enter number: "; cin >> number;
         }
      void putdata() const
         {
         cout << "\n   Name: " << name;
         cout << "\n   Number: " << number;
         }
   };
////////////////////////////////////////////////////////////////
class manager                           //yonetim
   {
   private:
      string title;                     //"baskan yardimcisi " etc..
      double dues;                      //golf kulubu aidat odemeleri
      employee emp;                     //employee sinifinin nesnesi
      student stu;                      //student sinifinin nesnesi
   public:
      void getdata()
         {
         emp.getdata();
```



```
         cout << "   Enter title: "; cin >> title;
         cout << "   Enter golf club dues: "; cin >> dues;
         stu.getedu();
         }
      void putdata() const
         {
         emp.putdata();
         cout << "\n   Title: " << title;
         cout << "\n   Golf club dues: " << dues;
         stu.putedu();
         }
   };
////////////////////////////////////////////////////////////////
class scientist                         //bilim adami
   {
   private:
      int pubs;                         //yayinlarin sayisi
      employee emp;                     //employee sinifinin bir nesnesi
      student stu;                      //student sinifinin bir nesnesi
   public:
      void getdata()
         {
         emp.getdata();
         cout << "   Enter number of pubs: "; cin >> pubs;
         stu.getedu();
         }
      void putdata() const
         {
         emp.putdata();
         cout << "\n   Number of publications: " << pubs;
         stu.putedu();
         }
   };
////////////////////////////////////////////////////////////////
class laborer                           //isci
   {
   private:
      employee emp;                     //employee sinifinin bir nesnesi
   public:
      void getdata()
         { emp.getdata(); }
      void putdata() const
         { emp.putdata(); }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   manager m1;
   scientist s1, s2;
   laborer l1;

   cout << endl;
   cout << "\nEnter data for manager 1";   //birkac calisanla ilgili
   m1.getdata();                           //verileri al

   cout << "\nEnter data for scientist 1";
   s1.getdata();

   cout << "\nEnter data for scientist 2";
```

```
s2.getdata();

cout << "\nEnter data for laborer 1";
l1.getdata();

cout << "\nData on manager 1";          //birkac calisanla ilgili
m1.putdata();                           //verileri ekranda goster

cout << "\nData on scientist 1";
s1.putdata();

cout << "\nData on scientist 2";
s2.putdata();

cout << "\nData on laborer 1";
l1.putdata();
cout << endl;
return 0;
}
```

EMPCONT içinde yer alan **student** ve **employee** sınıfları **EMPMULT**'takilerle aynıdır; fakat, bu sınıflar **manager** ve **scientist** sınıfları ile farklı bir biçimde ilişkilidir.

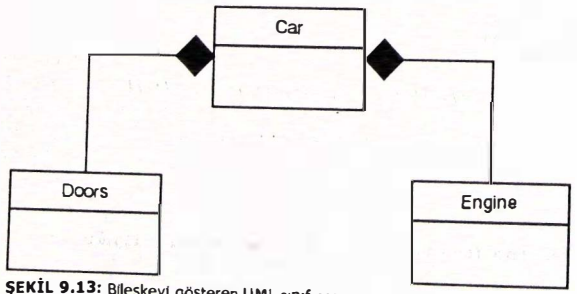
Bileşke: Daha Kuvvetli Bir Birlik

Bileşke (composition), birliğin kuvvetlendirilmiş bir şeklidir. Bileşke, birliğin tüm özelliklerine sahiptir, ilaveten iki özellik daha içerir:

- Parça sadece bir bütüne ait olabilir.
- Parçanın ömrü bütünün ömrü ile aynıdır.

Bir araba, (diğer parçaların yanı sıra) kapılardan meydana gelir. Kapılar bir başka arabaya ait olamazlar; arabayla doğar ve arabayla ölürler. Bir oda, taban, tavan ve duvarlardan oluşur. Birlik, bir "sahiplik" ilişkisi iken bileşke, bir "oluşma" ("consists of") ilişkisidir.

UML şemalarında bir bileşke, bir birlik ile aynı biçimde gösterilir; tek farkla, baklava şeklindeki ok ucu açık değil, içi doludur. Bu, Şekil 9.13'te gösteriliyor.



ŞEKİL 9.13: Bileşkeyi gösteren UML sınıf şeması.

Hatta tek bir nesne bile bileşke sayesinde bir sınıf ile ilişkilendirilebilir. Bir arabada yalnızca tek bir motor vardır.

Kalıtım ve Program Geliştirmek

On yıllar boyunca her yerdeki programcılar tarafından uygulanan program geliştirme işlemi, nesne yönelimli programlama sayesinde esas itibariyle değişikliğe uğrar. Bu sadece nesne yönelimli programlamaya ait sınıfların kullanımından kaynaklamaz. Kalıtım da ayrıca bunun bir sebebidir. Şimdi bunun nasıl gerçekleştiğine bir göz atalım.

A programcısı bir sınıf tanımlar. Belki de bu, **Distance** sınıfı gibi bir şeydir. Bu sınıf, kullanıcı tarafından tanımlanan veri tipleri üzerinde aritmetik işlemler yapan komple bir üye fonksiyonlar kümesi de içerebilir.

B programcısı **Distance** sınıfını beğenir; fakat, negatif mesafeler kullanılarak bu sınıfın geliştirileceğini düşünür. Çözüm, **ENGLEN** örneğindeki **DistSign** gibi, **Distance**'dan türetilen fakat negatif mesafeleri de gerçeklemek amacıyla bazı ilaveler içeren yeni bir sınıf tanımlamaktır.

Daha sonra C ve D programcıları **DistSign** sınıfını kullanan uygulamalar yazarlar.

B programcısının **Distance** üye fonksiyonlarının kaynak koduna erişim hakkı olmayabilir. Ayrıca, C ve D programcıları da **DistSign**'ın kaynak koduna erişim hakkına sahip olmayabilirler. Yine de, C++'ın yazılımların yeniden kullanılabilirliği özeliğinden ötürü B, A'nın çalışmalarını değiştirebilir ve geliştirebilir; C ve D ise B'nin (ve A'nın) çalışmalarından yararlanabilirler.

Dikkat ederseniz, yazılım aracı geliştirenler ile uygulama yazarları arasındaki fark giderek netliğini kaybediyor. A programcısı genel amaçlı bir programlama aracı (**Distance** sınıfı) geliştiriyor. B programcısı bunun özelleştirilmiş bir versiyonunu geliştiriyor (**DistSign** sınıfı). C ve D programcıları uygulama geliştiriyorlar. A, bir araç geliştiricidir; C ve D ise birer uygulama geliştiricisi. B, ikisinin arasında bir yerdedir. Ne de olsa, nesne yönelimli programlama, programlama şartlarını çok daha esnek, aynı zamanda çok daha karmaşık hale getirmektedir.

Bölüm 13'te bir sınıfın nasıl iki parçaya ayrılabileceğini göreceğiz: Alıcı tarafından erişilebilen bir parça ve sadece bir nesne formunda dağıtılabilen bir parça. Bu sayede bir sınıf, kaynak kodu dağıtılmadan, diğer programcılar tarafından da kullanılabilir.

Özet

Türetilmiş sınıf olarak adlandırılan bir sınıf, *temel sınıf* olarak adlandırılan bir başka sınıfın özelliklerini aynen kendisine aktarabilir. Türetilmiş sınıf kendisine ait başka özellikler de ekleyebilir; böylece, temel sınıfın özelleştirilmiş bir versiyonu halini alır. Kalıtım, mevcut sınıfların becerilerini genişletmek ve hiyerarşik ilişkiler kullanarak programlar tasarlamak için güçlü bir çözüm sağlamaktadır.

Temel sınıf üyelerinin, türetilmiş sınıflardan ve türetilmiş sınıfların nesnelerinden erişilebilir olmaları önemli bir hususdur. İsimlerinin önünde **protected** anahtar kelimesi kullanılan temel sınıfın veri ve fonksiyonları, türetilmiş sınıflardan erişilebilirler; fakat, türetilmiş sınıfların nesneleri de dahil olmak üzere başka hiçbir nesne tarafından erişilemezler. Sınıflar, temel sınıftan public veya private olarak türetilebilirler. Public olarak türetilmiş bir sınıfın nesneleri, temel sınıfın public üyelerine erişilebilirler; private olarak türetilen sınıfın nesneleri erişemezler.

Bir sınıf birden fazla temel sınıftan türetilebilir. Buna *çoklu kalıtım* denir. Bir sınıf, bir başka sınıfın içinde de yer alabilir.

UML dilinde kalıtım, genelleştirme olarak adlandırılır. Bu ilişki, sınıf şemalarında temel (ebeveyn) sınıfı işaret eden açık (içi boş) bir üçgenle simgelenir.

Birlik (aggregation), bir "sahiplik" veya "parça-bütün" ilişkisidir: Bir sınıf diğer bir sınıfın nesnelerini içerir. Birlik, UML sınıf şemalarında, parça-bütün çiftinin "bütün" kısmını işaret eden açık (içi boş) baklava şekliyle simgelenir. Bileşke (composition) ise birliğin kuvvetlendirilmiş bir şeklidir. Bileşkede kullanılan okun ucu içi boş değil, doludur.

Kalıtım, yazılımların yeniden kullanılabilirliğine imkan verir: Türetilmiş sınıflar, temel sınıfı değiştirmeden – hatta temel sınıfın kaynak koduna erişmelerine bile gerek kalmadan – temel sınıfın becerilerini genişletebilirler. Bu, yazılım geliştirme sürecinde yeni esnekliklere öncülük eder ve yazılım geliştiriciler için daha geniş görev çeşitliliği sağlar.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Kalıtım için aşağıdakilerden hangisi (hangileri) söylenebilir?
   a. Kalıtım, genel sınıfları daha spesifik sınıflara dönüştürme yöntemlerinden biridir.
   b. Kalıtım, sınıfların nesnelerine argüman aktarma yöntemlerinden biridir.
   c. Kalıtım, mevcut sınıflara, sınıfları yeniden yazmaya gerek kalmadan yeni özellikler ekleme yöntemlerinden biridir.
   d. Kalıtım, veri gizliliği ve benzer göreve sahip verilerin paketlenmesi (encapsulation) işlevlerini geliştiren yöntemlerden biridir.
2. Bir "çocuk" sınıf, bir temel sınıftan _____.
3. Kalıtımın sağladığı avantajlar arasında aşağıdakilerden hangisi (hangileri) yer alır?
   a. Doğal seleksiyon yoluyla sınıfın büyümesini sağlamak.
   b. Sınıf kütüphanelerinin kullanımını kolaylaştırmak.
   c. Kodun yeniden yazılmasını önlemek.
   d. Kullanışlı bir kavramsal iskelet çıkarmak.
4. `Alphonso` sınıfından açık olarak türetilen bir `Bosworth` sınıfı için bir belirtecin ilk satırını yazın.
5. Doğru/Yanlış: Bir temel sınıfa bir türetilmiş sınıf eklemek temel sınıf üzerinde esaslı değişiklikler gerektirir.
6. Türetilmiş bir sınıfın üye fonksiyonları tarafından erişilebilmeleri için temel sınıftaki veri veya fonksiyonlar public veya _____ olmalıdır.
7. Eğer bir temel sınıf `basefunc()` isminde bir üye fonksiyon içeriyorsa ve bir türetilmiş sınıf bu isimde bir fonksiyona sahip değilse, türetilmiş sınıfın bir nesnesi `basefunc()` fonksiyonuna erişebilir mi?
8. 4. soruda bahsedilen sınıfları düşünün. `Alphonso` isimli sınıfın `alfunc()` isminde bir üye fonksiyon içerdiğini varsayın. `Bosworth` sınıfının `BosworthObj` isimli nesnesinin, `alfunc()` fonksiyonuna erişmesini mümkün kılan bir ifade yazın.
9. Doğru/Yanlış: Bir türetilmiş sınıf içinde eğer hiç kurucu fonksiyon bulunmuyorsa, türetilmiş sınıfın nesneleri temel sınıfın kurucu fonksiyonlarını kullanırlar.
10. Eğer bir temel sınıf ile bir türetilmiş sınıfın her biri, aynı isimde bir üye fonksiyon içeriyorlarsa, kapsam çözünürlük operatörünün kullanılmadığını varsayarak, bu üye fonksiyonlardan hangisi türetilmiş sınıfın bir nesnesi tarafından çağrılacaktır?

11. 4. sorudaki `Bosworth` türetilmiş sınıfının argümansız kurucu fonksiyonu için bir deklarasyon yazın. Bu kurucu fonksiyon, `Alphonso` temel sınıfının argümansız kurucu fonksiyonunu çağırmalı.
12. Kapsam çözünürlük operatörü genellikle
    a. değişkenlerin belirli bir fonksiyona erişimlerini kısıtlar.
    b. bir sınıfın hangi temel sınıftan türetildiğini bildirir.
    c. belirli bir sınıfı açıkça belirtir.
    d. belirsizlikleri ortadan kaldırır.
13. Doğru/Yanlış: Kendisinden hiçbir zaman bir nesne oluşturulmayacak bir sınıf tanımlamak kimi zaman kullanışlı olur.
14. Varsayalım, `Derv` isimli bir sınıf `Base` isimli temel sınıftan türetilmiş olsun. Türetilmiş sınıfın kurucu fonksiyonu için bir deklarasyon yazın. Bu fonksiyon tek argüman almalı ve bu argümanı temel sınıfın kurucu fonksiyonuna aktarmalı.
15. `Derv` isimli bir sınıfın `Base` sınıfından private olarak türetildiğini varsayalım. `main()` içinde bulunan `Derv`'in bir nesnesi aşağıdakilerden hangisine (hangilerine) erişebilir?
    a. `Derv`'in public üyelerine.
    b. `Derv`'in protected üyelerine.
    c. `Derv`'i private üyelerine.
    d. `Base`'in public üyelerine.
    e. `Base`'in protected üyelerine.
    f. `Base`'in private üyelerine.
16. Doğru/Yanlış: `A` sınıfından bir `B` sınıfı türetilmiş ve `B` sınıfından da bir `C` sınıfı türetilmişse; `C` sınıfından da bir `D` sınıfı türetilebilir.
17. Sınıf hiyerarşisi
    a. bir organizasyon şemasının gösterdiği ilişkilerin aynısını gösterir.
    b. bir "sahiplik" ("has a") ilişkisi tanımlar.
    c. bir "tür" ("is kind of") ilişkisi tanımlar.
    d. bir aile ağacının gösterdiği ilişkilerin aynısını gösterir.
18. `Wheel` ve `Rubber` sınıflarından türetilmiş olan `Tire` sınıfının tanımının ilk satırını yazın.
19. Varsayalım, `Derv` sınıfı `Base` sınıfından türetilmiş olsun. Her iki sınıf da argüman almayan `func()` isimli bir fonksiyon içeriyorlar. Temel sınıftaki `func()` fonksiyonunu çağıran `Derv` sınıfının bir üye fonksiyonuna gitmek için bir ifade yazın.
20. Doğru/Yanlış: Bir sınıfın nesnelerini bir başka sınıfın nesnesi yapmak kurallara aykırıdır.
21. UML dilinde kalıtıma _____ denir.
22. Birlik (aggregation)
    a. örneklemenin daha kuvvetli bir şeklidir.
    b. Genelleştirmenin daha kuvvetli bir şeklidir.
    c. Bileşkenin daha kuvvetli bir şeklidir.
    d. Bir "sahiplik" ilişkisidir.
23. Doğru/Yanlış: Genelleştirmeyi simgeleyen bir ok, daha spesifik olan sınıfı işaret eder.
24. Bileşke _____ _____ bir şeklidir.

Alıştırmalar

Yıldızla işaretli soruların cevaplarını Ek G'de bulabilirsiniz.

1. Ürünlerini hem kitap hem de kaset olarak pazarlayan bir yayınevi düşünün. Ürünün adını (bir karakter katarı) ve fiyatını (`float` tipinde) tutacak `publication` isminde bir sınıf oluşturun. Bu sınıftan iki sınıf türetin: Sayfa sayısını (`int` tipinde) ilave eden `book` isminde bir sınıf ve kasetin (dakika cinsinden) çalma süresini (`float` tipinde) ilave eden `tape` isminde bir sınıf. Bu üç sınıfın her biri, kullanıcıdan verileri almak için `getdata()` fonksiyonunu ve verileri ekranda göstermek için `putdata()` fonksiyonunu içermeleri gerekiyor.*
   `book` ve `tape` sınıflarını, birer örneklerini oluşturarak test eden bir `main()` programı yazın. Program, `getdata()` yardımıyla kullanıcıdan verileri girmesini istemeli ve `putdata()` ile bu verileri ekranda göstermeli.
2. Bölüm 8'deki `STRCONV` örneğini hatırlayın. Bu örnekteki `String` sınıfının bir kusuru vardır: Nesnelerinin başlangıç değeri olarak çok fazla karakter içermesi durumunda kendisini koruyamıyor (`SZ` sabitinin değeri 80). Örneğin, şu tanım*

```
String s = "This string will surely exceed the width of the "
                "screen,which is what the SZ constant represents.";
```

   `s`'in içindeki `str` dizisinin kestirilemeyen sonuçlar doğuracak şekilde (mesela, sistem çökebilir) taşmasına neden olacak.
   `String`'i temel sınıf alarak bir `Pstring` ("protected string" anlamında) sınıfı türetin. `Pstring`, tanımda fazlasıyla uzun bir karakter katarı sabiti kullanıldığında tamponun taşmasını önlüyor. Türetilmiş sınıftaki yeni bir kurucu fonksiyon eğer sabit, `SZ`'den daha uzun ise sadece `SZ-1` karakteri `str`'a kopyalamalı; eğer `SZ`'den kısaysa sabitin tümünü kopyalamalı. Değişik uzunluktaki karakter katarlarını test eden bir `main()` programı yazın.
3. Birinci alıştırmadaki `publication`, `book` ve `tape` sınıfları ile başlayın. Üç `float`'tan ibaret bir diziyi tutan `sales` adında bir temel sınıf ilave edin. Böylece bu sınıf, belirli bir yayının son üç aydaki dolar üzerinden satışlarını kaydedebilir. Kullanıcıdan üç satış miktarı almak için bir `getdata()` fonksiyonunu ve satış rakamlarını görüntülemek için de bir `putdata()` fonksiyonunu programa dahil edin. `book` ve `tape` sınıflarını hem `publication`'dan hem de `sales`'dan türetilecek şekilde değiştirin. `book` ve `tape`'in bir nesnesi, diğer verilerin yanı sıra satış verilerini de kullanıcıdan alıp görüntülemeli. Bir `book`, bir de `tape` nesnesi tanımlayan bir `main()` fonksiyonu yazın ve bu nesnelerin giriş/çıkış becerilerini sınayın.*
4. Birinci ve üçüncü alıştırmalardaki yayıncının kitaplarını dağıtmak için üçüncü bir yöntem dahil etmeye karar verdiğini varsayalım: Bilgisayar diskleri ile dağıtım (okumalarını diz üstü bilgisayarlarında yapmayı sevenler için). `book` ve `tape` gibi `publication`'dan türetilen bir `disk` sınıfı ilave edin. `disk` sınıfı, diğer sınıfların içinde yer alan fonksiyonların aynısını içermeli. Sadece bu sınıfa ait olan veri öğesi ise disk tipi olmalı: CD veya DVD. Bu öğeyi tutmak için bir `enum` tipi kullanabilirsiniz. Kullanıcı `c` veya `d` tuşlayarak uygun olan tipi seçebilir.
5. Bu bölümdeki `EMPLOY` programındaki `employee` sınıfından `employee2` isminde bir sınıf türetin. Bu yeni sınıf, `compensation` isminde `double` tipinde bir veri öğesi ilave etmeli; ayrıca, `period` isminde, çalışanın saat bazında mı, haftalık mı, aylık mı ücretlendirileceğine işaret eden `enum` tipinde bir veri öğesi daha içermelidir. Kolaylık olması açısından, `manager`, `scientist` ve `laborer` sınıflarını `employee` yerine `employee2`'den türetilecek şekilde değiştirebilirsiniz. Buna rağmen, dikkat ederseniz, `compensation` adında ayrı bir sınıf oluşturup; `manager2`, `scientist2` ve `laborer2` diye üç yeni sınıf tanımlamak ve bu üç yeni sınıfı orijinal `manager`, `scientist` ve `laborer` sınıflarından ve `compensation` sınıfından türetmek için çoklu kalıtım kullanmak bir çok koşulda nesne yönelimli programlamanın ruhuna daha uygun düşebilir. Bu şekilde orijinal sınıfların hiç biri değiştirilmek zorunda kalmaz.
6. Bölüm 8'deki `AROVER3` programı ile başlayın. `safearay` sınıfını o programdaki gibi bırakın. Kullanıcının bir kurucu fonksiyon içinde, dizinin hem üst hem de alt sınırlarını belirlediği, bir sınıf türetin. Bu, Bölüm 8'deki 9. alıştırmanın aynısı; yalnızca, burada orijinal sınıfı değiştirmek yerine, yeni bir sınıf türetmek için kalıtım kullanılıyor. (Bu yeni sınıfa `safehilo` ismini verebilirsiniz.)
7. Bu bölümdeki `COUNTEN2` programı ile başlayın. Bu program sadece önek notasyonu kullanarak bir sayaçtaki sayıyı artırabilir veya azaltabilir. Kalıtım kullanarak, artırma ve eksiltme işlemlerinin her ikisinde sonek notasyonunu kullanmaya imkan veren bir beceri ilave edin. (Sonek notasyonunun tanımı için Bölüm 8'e bakın.)
8. Visual Basic gibi bazı bilgisayar dillerindeki operatörler, mevcut bir karakter katarının parçalarını seçip, bunları başka karakter katarlarına atamanıza imkan verirler. (Standart C++ `string` sınıfı farklı bir yaklaşım sunuyor.) Kalıtım kullanarak, 2. alıştırmadaki `Pstring` sınıfına böyle bir beceri ilave edin. Türetilmiş sınıf olan `Pstring2` üç yeni fonksiyon içermeli: `left()`, `mid()` ve `right()`.

```
s2.left(s1, n)   //s2'ye, s1'in en soldaki n karakteri ataniyor
s2.mid(s1, s, n) //karakter numarasi s olan konumdan baslayarak,
                 //s2'ye, s1'in ortasindaki n karakter ataniyor
                 //(en soldaki karakter 0 numarali)
s2.right(s1, n)  // s2'ye, s1'in en sagdaki n karakteri ataniyor
```

   `s1`'in ilgili parçalarını geçici bir `Pstring2` nesnesine karakter karakter kopyalamak için `for` döngüleri kullanabilirsiniz. Sonra bu `Pstring2` nesnesi döndürülecektir. Ekstra puan için, bu fonksiyonların referans yoluyla dönmesini sağlayın. Böylece, mevcut bir karakter katarının parçalarını değiştirmek için bu fonksiyonların çağrılarını eşittir işaretinin sol tarafında kullanabilirsiniz.
9. Birinci alıştırmadaki `publication`, `book` ve `tape` sınıfları ile başlayın. Kitaplar ve kasetler için yayın tarihini de ilave etmek istediğinizi varsayalım. `publication` sınıfından, bu üye veriyi içeren `publication2` adında yeni bir sınıf türetin. Sonra, `book` ve `tape`'i `publication` yerine `publication2`'den türetilecek biçimde değiştirin. Üye fonksiyonlarda gerekli tüm değişiklikleri yapın; böylece, kullanıcı diğer verilerle birlikte tarihleri de programa verip alabilsin. Tarihler için Bölüm 6'daki 5. alıştırmada anlatılan `date` sınıfını kullanabilirsiniz. `date` sınıfı, tarihi (ay, gün ve yıl için) üç tamsayı halinde saklıyor.
10. Bu bölümdeki `EMPMULT` programında sadece tek tip yönetici mevcuttur. Herhangi bir ciddi şirkette yöneticilerin yanı sıra yetki sahibi kişiler de vardır. `manager` sınıfından `executive` adında bir sınıf türetin. (Yetki sahibi birinin en üst düzey yönetici olduğunu varsayacağız.) `executive` sınıfının içindeki ilave veriler, çalışanın yıllık primi ve portföyünde mevcut olan şirket hisselerinin sayısı olmalı. Bu verilerin kullanıcıdan alınmasını ve diğer `manager` verileriyle birlikte ekranda gösterilmesini sağlayacak ilgili üye fonksiyonları ekleyin.
11. Çeşitli durumlarda, bir çift sayının tek bir birim olarak ele alınması gerekebilir. Örneğin, ekrandaki her koordinat x (yatay) bileşeni ve y (düşey) bileşenine sahiptir. Bu tür sayı çiftlerini, `pair` adında ve iki tane `int` tipinde üye değişken içeren bir yapı olarak

simgeleyin. Şimdi, `pair` değişkenlerini bir yığında saklamayı mümkün kılmak istediğinizi varsayın. Yani, bir `push()` fonksiyonuna tek çağrı yaparak ve `pair` tipindeki yapıyı fonksiyona argüman olarak aktararak bir sayı çiftini yığına yerleştirmeyi mümkün kılmak istiyorsunuz. Aynı şekilde, `pair` tipinde bir yapı döndüren bir `pop()` fonksiyonuna tek çağrı yaparak bir çift sayıyı yığından almak istiyorsunuz. Bu bölümdeki `STAKEN` programında yer alan `Stack2` sınıfı ile başlayın. Bu sınıftan `pairStack` adında yeni bir sınıf türetin. Bu yeni sınıf, sadece iki üye içermek zorunda: Aşırı yüklenmiş `push()` ve `pop()` fonksiyonları. `PairStack::push()` fonksiyonunun, çift olarak iki tamsayıyı saklamak için `Stack2::push()` fonksiyonuna iki çağrı yapması gerekecek; `pairStack::pop()` fonksiyonu da `Stack2::pop()` fonksiyonuna iki çağrı yapmak zorunda olacak (fonksiyonların aynı sıra ile çağrılması gerekmiyor).

12. Şaşırtıcı ama, eski Britanya pound-shilling-pence para notasyonu (£9.19.11 – "Yapılar" adlı Bölüm 4'teki 10. alıştırmaya bakın) aslında hikayenin tamamı değil. Bir penny de halfpenny'lere ve farthing'lere bölünüyordu. Bir farthing, 1/4 penny değerindeydi. Bozuk para olarak halfpenny, farthing ve halffarthing mevcuttu. Neyse ki, bir penny'nin sekizde biri cinsinden sadece bu kadarı nümerik olarak ifade edilebiliyordu:

1/8 penny = 1 halffarthing
1/4 penny = 1 farthing
3/8 penny = 1.5 farthing
1/2 penny = 1 halfpenny (ha'penny olarak telaffuz edilir)
5/8 penny = 1 halfpenny + 1 halffarthing
3/4 penny = 1 halfpenny + 1 farthing
7/8 penny = 1 halfpenny + 1.5 farthing

Varsayalım, `sterling` sınıfına, bu tür ondalık penny'leri de ele alabilecek bir beceri katmak istiyoruz. I/O formatı şöyle, £1.1.1-1/4 veya şöyle, £9.19.11-7/8 olabilir. Burada tire ondalık kısmı penni'lerden ayırıyor.

`sterling` sınıfından `sterfrac` adında yeni bir sınıf türetin. Bu yeni sınıf, bir penny'nin sekizde bir oranlarını da içeren `sterling` miktarları üzerinde dört aritmetik işlemi yapabilmelidir. Bu sınıfın tek üye verisi, "sekizde bir"lerden kaç tane olduğuna işaret eden bir tamsayıdır; bunu `eighths` olarak adlandırabilirsiniz. Sekizde birlik değerleri ele alabilmek için `sterling` içindeki birçok fonksiyonunun işlevini artırmak zorunda kalacaksınız. Kullanıcı kesirli sayıları sadeleştirilmiş olarak girmeli; ekranda gösterilen kesirler de sadeleştirilerek gösterilmeli. `fraction` sınıfını (Bölüm 6'daki 11. alıştırmaya bakın) tüm ayrıntılarıyla kullanmanıza gerek yok; fakat, bunu ekstra puan olarak deneyebilirsiniz.

ONUNCU BÖLÜM

10

İŞARETÇİLER

İşaretçiler, C++ programlamada yaşanan sebepsiz korkulardandır: Nadirdir ki, bu kadar basit bir fikir bu kadar çok insanı şaşırtsın. Fakat bu sizi korkutmasın. Bu bölümde işaretçiler üzerindeki esrar perdesini kaldırmaya çalışacağız ve C++ programlamada işaretçilerin pratik kullanımlarını göstereceğiz.

İşaretçiler ne işe yarar? İşte işaretçilerin yaygın kullanımlarından bazıları:

- Dizi elemanlarına erişmek için.
- Bir fonksiyonun, orijinal argümanı değiştirmesi gerekiyorsa, bu tür fonksiyonlara argüman aktarmak için.
- Fonksiyonlara dizi ve karakter katarı aktarmak için.
- Sistemden bellek almak için.
- Veri yapıları tanımlamak için (örneğin, bağlı listeler).

Diğer birçok dilde (örneğin, Visual Basic ve Java) hiç işaretçi olmamasına rağmen işaretçiler C++'ın (ve C'nin) önemli bir özelliğidir. (Java'da referanslar vardır; bunlara bir tür sulandırılmış işaretçi diyebiliriz.) İşaretçilerin üzerinde bu denli durmak gerçekten gerekli mi? İşaretçiler olmadan da pek çok şey yapabilirsiniz; işaretçilerden yoksun olan önceki bölümler bunun bir göstergesidir. İşaretçileri kullanan bazı işlemler C++'ta diğer yollardan da gerçekleştirilebilir. Örneğin, dizi elemanları, işaretçi notasyonundan ziyade dizi notasyonu ile erişilebilir (ikisi arasındaki farkı birazdan göreceğiz); bir fonksiyon işaretçi olarak aktarılan argümanların yanı sıra referans olarak aktarılan argümanları da değiştirebilir.

Bununla birlikte, işaretçiler bazı durumlarda C++'ın gücünü artırmayı sağlayan başlıca yöntemlerden biridir. İşaretçiler kullanılarak veri yapılarının oluşturulması, mesela bağlı listeler veya ikili ağaçlar, göze çarpan örneklerdendir. Aslında, C++'ın birkaç temel özelliği, örneğin, sanal fonksiyonlar, `new` operatörü ve `this` işaretçisi ("Sanal Fonksiyonlar" adlı Bölüm 11'de anlatılıyor) işaretçilerin kullanımını gerektirir. Bu nedenle, C++'ta işaretçi kullanmadan bol bol programlama yapabiliyor olmanıza rağmen, dilden daha fazlasını elde etmek için işaretçilerin şart olduğunu fark edeceksiniz.

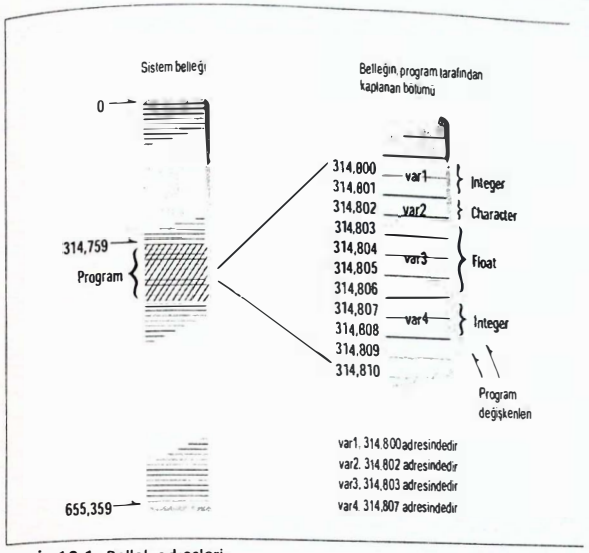
Bu bölümde, işaretçileri yavaş yavaş tanıtıyoruz; en temel kavramlardan başlayarak karmaşık işaretçi uygulamalarına doğru ilerleyeceğiz.

Eğer önceden C'yi biliyorsanız, muhtemelen bölümün ilk yarısını atlayabilirsiniz. Bununla birlikte, ikinci yarıdaki `new` ve `delete` operatörleri, işaretçi kullanarak dizi elemanlarına erişim, nesneleri işaret eden işaretçi dizileri ve nesnelerden oluşan bağlı listelerle ilgili bölümleri okumalısınız.

Adresler ve İşaretçiler

İşaretçilerin ardında yatan fikirler karmaşık değildir. İşte birinci temel kavram: Bilgisayarın belleğindeki her byte'ın bir *adresi* vardır. Adresler, tıpkı bir sokaktaki evlerin numaraları gibi birer sayıdan ibarettir. Bu sayılar 0 ile başlar ve 1, 2, 3 vs diye artar. Eğer 1 MB belleğiniz varsa, en yüksek adres 1,048,575'tir. (Elbette siz çok daha fazlasına sahipsinizdir.)

Programınız belleğe yüklendiği zaman bu adresler üzerinde belirli bir aralığı kaplar. Bu, programınızdaki her değişken ve her fonksiyon belirli bir adresten başlıyor demektir. Şekil 10.1 bunun nasıl olduğunu gösteriyor.



ŞEKİL 10.1: Bellek adresleri.

Adres Operatörü (&)

Bir değişkenin işgal ettiği adresi *adres* operatörü (&) kullanarak bulabilirsiniz. Bunun nasıl yapılacağını gösteren işte kısa bir program, **VARADRR**:

```
// varaddr.cpp
// degiskenlerin adresleri
#include <iostream>
using namespace std;
int main()
   {
   int var1 = 11;     //uc degisken tanimla ve
   int var2 = 22;     //deger ata
   int var3 = 33;

   cout << &var1 << endl      //bu degiskenlerin
        << &var2 << endl      //adreslerini yazdir
        << &var3 << endl;
   return 0;
   }
```

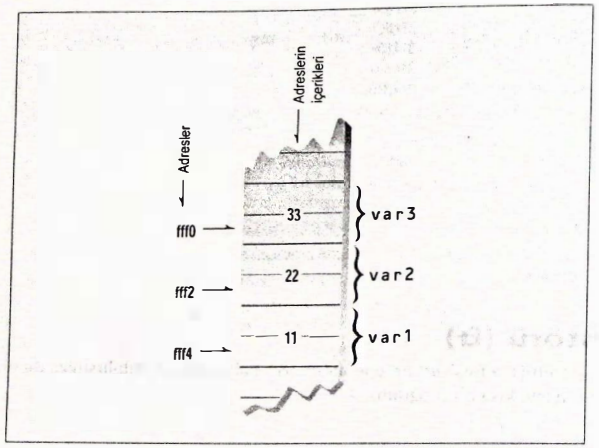
Bu basit program üç tamsayı değişken tanımlar ve bunlara başlangıçta 11, 22 ve 33 değerlerini atar. Program daha sonra bu değişkenlerin adreslerin dökümünü verir.

Bir programdaki değişkenlerin kapladığı gerçek adresler pek çok faktöre bağlıdır; söz gelişi, programın çalıştığı bilgisayar, işletim sisteminin boyutu ve diğer programlardan herhangi biri-

nin bellekte olup olmaması gibi. Bu nedenlerden dolayı, bu programı siz çalıştırdığınızda muhtemelen bizim elde ettiğimiz adresleri elde edemeyeceksiniz. (Hatta bir satırda bile aynı sonuçları iki kez elde edemeyebilirsiniz.) İşte bizim makinemizdeki çıktı şöyle:

```
0x8f4ffff4    <==== var1'in adresi
0x8f4ffff2    <==== var2'nin adresi
0x8f4ffff0    <==== var3'un adresi
```

Bir değişkenin *adresinin*, değişkenin *içeriğiyle* asla aynı anlama gelmediğini hatırınızdan çıkarmayın. Bu üç değişkenin içeriği 11, 22 ve 33. Şekil 10.2 bu üç değişkenin bellekte nasıl yer kapladığını gösteriyor.



ŞEKİL 10.2: Değişkenlerin adresleri ve içerikleri.

<< operatörü, adresleri onaltılık aritmetik kullanarak yorumlar. Her sayının önündeki 0x öneki buna işaret eder. Bu, bellek adreslerini göstermenin olağan bir yoludur. Onaltılık sayı sistemine aşina değilseniz, telaşlanmayın. Sizin gerçekten bilmeniz gereken her değişkenin ayrı bir adreste başladığıdır. Bununla birlikte, çıktıdan fark etmiş olabileceğiniz gibi, her adres bir sonrakinden tam olarak 2 byte farklıdır. Bu, tamsayıların (16 bit sistemde) bellekte 2 byte yer kaplamasından kaynaklanır. Eğer **char** tipinde değişkenler kullanmış olsaydık, bir **char** 1 byte yer kapladığı için bu değişkenlerin adresleri peş peşe olmuş olacaktı. Eğer **double** tipi kullanmış olsaydık, adresler arasındaki fark 8 byte olacaktı.

Adresler azalan sırada görünüyor, çünkü yerel değişkenler, belleğin aşağı yönüne doğru büyüyen yığında saklanırlar. Eğer global değişkenler kullanmış olsaydık, global değişkenler yukarı yönde büyüyen katmanda (heap) saklandıkları için artan sırada adrese sahip olacaklardı. Yine bu hususları dert etmenize gerek yok; çünkü derleyici sizin adınıza tüm ayrıntıların takibini yapar.

Bir değişken deklarasyonunda değişken isminin önünde yer alan adres operatörünü (&), fonksiyon deklarasyonunda veya tanımında tip isminin peşinden gelen referans operatörü (&) ile karıştırmayın. (Referanslar "Fonksiyonlar" adlı Bölüm 5'te anlatıldı.)

İşaretçi Değişkenleri

Adreslerin kendileri oldukça kısıtlayıcıdır. Aradığımız şeylerin bellekte nerede olduklarını bilmek hoştur, VARADDR programında olduğu gibi... Ancak, adres değerlerinin dökümünü almak o kadar kullanışlı değildir. Yine de, programlama gücümüzü artırma fırsatına sahibiz. Ancak bunun için ek bir kavram gerekiyor: *Adres değerlerini tutan değişkenler*. Karakterleri, tam sayıları, kayan noktalı sayıları vs saklayan değişken tiplerini görmüştük. Adresler de benzer şekilde saklanırlar. Bir adres değerini tutan değişkene *işaretçi değişkeni* veya kısaca *işaretçi* denir.

İşaretçi değişkenlerinin veri tipi nedir? Bir işaretçi değişkeninin tipi, adresini tutmakta olduğunuz değişken ile aynı değildir; bir **int**'e işaret eden bir işaretçinin tipi **int** değildir. Bir işaretçi veri tipinin **pointer** veya **ptr** gibi bir şey olarak adlandırılacağını düşünüyor olabilirsiniz. Ancak, işler bir parça daha karmaşık. Sıradaki örnek, PTRVAR, işaretçi değişkenlerinin söz dizimini gösteriyor.

```
// ptrvar.cpp
// isaretciler (adres degiskenleri)
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11;              //iki tamsayi degisken
   int var2 = 22;

   cout << &var1 << endl       //degiskenlerin adreslerini yazdir
        << &var2 << endl << endl;

   int* ptr;                   //tamsayilara isaret eden isaretci

   ptr = &var1;                //isaretci var1'e isaret ediyor
   cout << ptr << endl;        //isaretcinin degerini yazdir

   ptr = &var2;                //isaretci var2'ye isaret ediyor
   cout << ptr << endl;        //isaretci degerini yazdir
   return 0;
   }
```

Bu program iki tamsayı değişken tanımlar: **var1** ve **var2**. Bu değişkenlere 11 ve 22 başlangıç değerlerini atar; sonra da adreslerinin dökümünü verir.

Program daha sonra, aşağıdaki satırda gösterildiği gibi, bir *işaretçi değişkeni* tanımlar:

```
int* ptr;
```

Konuya yabancı olanlar için bu oldukça acayip bir sözdizimi olarak görünebilir. Asteriks, *işaret eder* anlamına geliyor. Böylece, bu ifade **ptr** değişkenini, **int**'e *işaret edecek* şekilde tanımlar. Bu ifade; bu değişken, tamsayı değişkenlerin adreslerini tutabilir, demenin bir başka yoludur.

Herhangi bir veri tipine işaret eden genel amaçlı bir işaretçi tipi tanımlama fikrinin neresi yanlış? Eğer bu tipi `pointer` olarak adlandırırsak, şu tür deklarasyonlar yazabiliriz:

```
pointer ptr;
```

Problem, *işaretçinin işaret ettiği* değişkenin *ne tür bir değişken* olduğunu derleyicinin bilmek zorunda olmasından kaynaklanır. (Bunun neden böyle olduğunu işaretçiler ve diziler hakkında konuşacağımız zaman öğreneceğiz.) C++'ta kullanılan söz dizimi, her türlü tip için bu tipe işaret eden işaretçi tanımlamaya imkan verir.

```
char* cptr;          //char'a isaret eder
int* iptr;           //int'e isaret eder
float* fptr;         //float'a isaret eder
Distance* distptr;   //kullanicinin tanimladigi Distance sinifina
                     //isaret eder
```

Söz Dizimi ile İlgili Önemsiz Tartışmalar

İşaretçi tanımlarında asteriksi tip ismine değil de, değişken ismine yakın yazmanın daha yaygın yaklaşım olduğuna dikkat çekmeliyiz.

```
char *charptr;
```

Derleyici açısından bunun bir önemi yoktur; fakat, asteriksi tipin yanına yerleştirmek asteriksin değişken isminin değil de, değişken tipinin (`char`'ı işaret eden işaretçi) bir parçası olduğunu vurgulamaya yardımcı olur.

Aynı tipte birden fazla işaretçiyi tek satırda tanımlıyorsanız, işaret edilen tipi bir kez yazmanız gerekir, fakat değişkenlerin her birinin önüne bir asteriks yerleştirmelisiniz.

```
char* ptr1, * ptr2, * ptr3;   //char* tipinde uc degisken
```

Yada, "asteriks ismin yanında yer almalı" yaklaşımını izleyebilirsiniz.

```
char *ptr1, *ptr2, *ptr3;     //char* tipinde uc degisken
```

İşaretçilerin Bir Değeri Olmalı

`0x8f4ffff4` gibi bir adres *işaretçi sabiti* olarak düşünülebilir. `ptr` gibi işaretçi ise bir *işaretçi değişkeni* olarak düşünülebilir. Tıpkı tamsayı değişkeni `var1`'e 11 sabit değeri atanabilmesi gibi, işaretçi değişkeni `ptr`'a da `0x8f4ffff4` sabit değeri atanabilir.

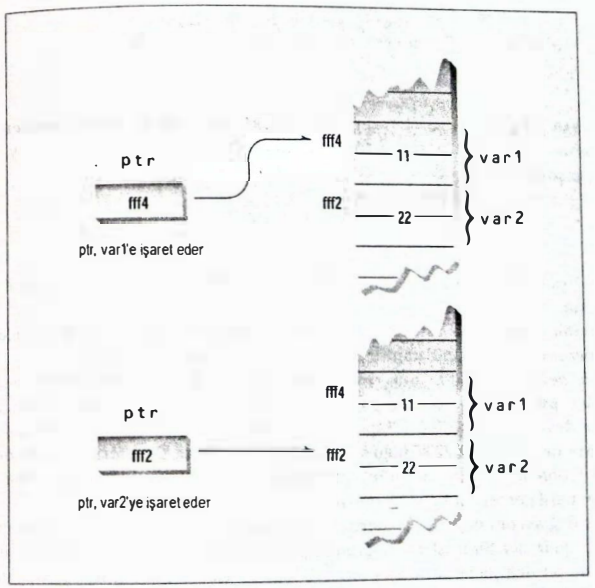
Bir değişkeni ilk kez tanımladığımızda (eğer aynı zamanda ilk değer de atamamışsak), değişken hiçbir değer içermez. İçinde alakasız bir değer olabilir ama, bu bir anlam taşımaz. İşaretçiler söz konusu olduğunda ise, alakasız bir değer bellekte bir şeyin adresidir; fakat, muhtemelen istemediğimiz bir şeyin adresidir. Bu nedenle, bir işaretçi kullanılmadan önce spesifik bir adres işaretçinin içine yerleştirilmelidir. `PTRVAR` programında, `ptr`'a önce `var1`'in adresi atanıyor:

```
ptr=&var1     <==== var1'in adresini ptr'in icine koy
```


Bunun peşinden program, `ptr`'ın içindeki değeri ekrana yazar. Bu değer, `&var1` için ekrana yazılan adres ile aynı olmalıdır. Aynı işaretçi değişkenine, yani `ptr`'a daha sonra `var2`'nin adresi atanır ve bu değer ekrana yazdırılır. Şekil 10.3 `PTRVAR` programının işleyişini gösteriyor. `PTRVAR`'ın çıktısı şöyledir:

```
0x8f51fff4   <==== var1'in adresi
0x8f51fff2   <==== var2'nin adresi
0x8f51fff4   <==== ptr, var1'in adresini gosterecek sekilde ayarlanir
0x8f51fff2   <==== ptr, var2'in adresini gosterecek sekilde ayarlanir
```

Özetlemek gerekirse; bir işaretçi, doğru tipteki herhangi bir değişkenin adresini tutabilir. Bir işaretçi adres bekleyen bir haznedir. Bununla birlikte, bir işaretçiye belli bir değer verilmesi gerekli değildir. Aksi halde, işaretçi, istemediğimiz bir yerlere işaret edecektir; program kodumuz veya işletim sistemi işaretçinin işaret ettiği yerler olabilir. Yanlış işaretçi değerleri sistemin çökmesine yol açabilir; ayrıca, derleyici uyarıda bulunmadığı için hataların giderilmesi de zordur. Ana fikir: Her işaretçi değişkenine, değişkeni kullanmadan önce geçerli bir adres verdiğinizden emin olun.



ŞEKİL 10.3: `ptr`'ın içindeki değerleri değiştirmek.

İşaret Edilen Değişkene Erişmek

Diyelim ki, bir değişkeninin ismini bilmiyoruz, ama adresini biliyoruz. Değişkenin içeriğine erişebilir miyiz? (Değişken isimlerinin izini kaybetmek kötü bir yönetim şekli olarak görünebilir; fakat, isimlerini bilmediğimiz pek çok değişken olduğunu yakında öğreneceğiz.)

Bir değişkenin ismi yerine adresini kullanarak değişkenin değerine erişmek için özel bir söz dizimi mevcuttur. İşte bunun nasıl gerçeklendiğini gösteren bir örnek program olan **PTRACC**:

```
// ptracc.cpp
// isaretcinin isaret ettigi degiskene erismek
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11;              //iki tamsayi degisken
   int var2 = 22;
   int* ptr;                   //tamsayilara isaret eden isaretci

   ptr = &var1;                //isaretci var1'e isaret ediyor
   cout << *ptr << endl;       //isaretcinin icerigini yazdir(11)

   ptr = &var2;                //isaretci var2'e isaret ediyor
   cout << *ptr << endl;       //isaretcinin icerigini yazdir (22)
   return 0;
   }
```

Bu program **PTRVAR** programına çok benzer; yalnızca, **ptr**'ın içindeki adres değerlerini ekrana yazdırmak yerine, **ptr**'ın gösterdiği adreste tutulan tamsayı değerini ekrana yazdırır. Programın çıktısı işte şöyledir:

```
11
22
```

var1 ve **var2** değişkenlerine erişimi sağlayan deyim, ***ptr**'dır ve bu deyim iki **cout** ifadesinin her birinde yer alır.

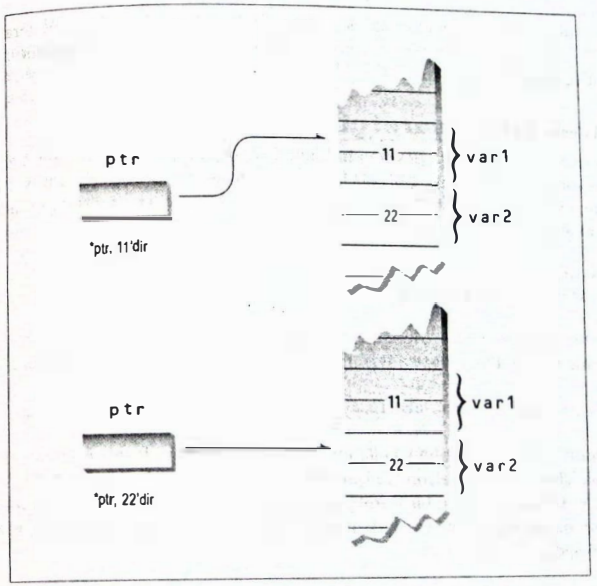
Bir değişken isminin önünde bir asteriks kullanıldığı zaman, ***ptr** deyiminde olduğu gibi, buna *dereferans operatörü* (veya kimi zaman *indirection operatörü*) denir. Bunun anlamı, *işaret edilen değişkenin değeri* demektir. Yani, ***ptr** deyimi, **ptr** tarafından işaret edilen değişkenin değerini simgeler. **ptr**, **var1**'in adresini gösterecek şekilde ayarlandığında, **var1**'in değeri 11 olduğu için ***ptr** deyimi 11 değerine sahip olur. **ptr**, **var2**'nin adresini gösterecek şekilde değiştirildiğinde, ***ptr** deyimi, **var2** 22 olduğu için 22 değerine ulaşır. Dereferans operatörünün bir diğer ismi *içerik* (*contents of*) operatörüdür; bu, aynı şeyi bir başka yoldan ifade etme şeklidir. Şekil 10.4 bunun nasıl gerçeklendiğini gösteriyor.

Bir işaretçiyi, bir değişkenin değerini göstermek için kullanabileceğiniz gibi, değişken üzerinde gerçekleştireceğiniz her türlü işlemi doğrudan gerçekleştirmek için de kullanabilirsiniz. **PTRTO** isimli program, bir değişkene bir değer atamak için bir işaretçi kullanır; sonra, yine işaretçi yardımıyla bu değeri bir başka değişkene atar:

```
// ptrto.cpp
// isaretciler kullanilarak diger erisimler
#include <iostream>
```



```
using namespace std;

int main()
   {
   int var1, var2;          //iki tamsayi degisken
   int* ptr;                //tamsayilara isaret eden isaretci

   ptr = &var1;             //isaretciye var1'in adresini ata
   *ptr = 37;               //var1=37 ile ayni
   var2 = *ptr;             //var2=var1 ile ayni

   cout << var2 << endl;    //var2'nin 37 oldugunu dogrula
   return 0;
   }
```



ŞEKİL 10.4: İşaretçi üzerinden erişim.

Hatırlarsanız, dereferans operatörü olarak kullanılan asteriks ile işaretçi değişkenlerini tanımlamak için kullanılan asteriks farklı anlamlar içeriyorlar. Dereferans operatörü değişkenin *önünde* yer alıyor ve *işaret edilen değişkenin değeri* anlamını taşıyor. Tanım sırasında kullanılan asteriks ise *işaret eder* anlamını taşıyor.

```
int* ptr;       //deklarasyon: int'e isaret eden isaretci
*ptr = 37;      //dolayli gosterim: ptr'in isaret ettigi degiskenin degeri
```

Bir adreste tutulan bir değere erişmek için dereferans operatörü kullanmak *dolaylı adresleme* (kimi zaman da *dereferencing*) olarak adlandırılır.

İşte, şimdiye kadar öğrendiklerimizin derli toplu bir özeti:

```
int v;      //int tipinde v degiskeni tanimlar
int* p;     //p'yi, int'e isaret eden bir isaretci olarak tanimlar
p = &v;     //v degiskeninin adresini p isaretcisine atar
v = 3;      //v'ye 3 degerini verir
*p = 3;     //bu da v'ye 3 degerini verir
```

Son iki ifade normal (veya direkt) adresleme ile işaretçi yoluyla (veya dolaylı) adresleme arasındaki farkı gösteriyor. Normal adresleme yaparken değişkene ismi ile ulaşıyoruz. İşaretçi yoluyla adresleme yaparken aynı değişkene adresini kullanarak ulaşıyoruz.

Bu bölümde şimdiye kadar gösterdiğimiz örnek programlarda, değişkenlere erişmek için işaretçi deyimleri kullanmanın gerçekten hiçbir avantajı yoktur, çünkü bu değişkenlere zaten doğrudan erişebiliriz. İşaretçilerin değeri, değişkenlere doğrudan erişemediğimiz durumlarda öne çıkar. Bu konuyu daha sonra ele alacağız.

void'a İşaret Eden İşaretçi

İşaretçilerin nasıl çalıştığını anlamaya devam etmeden önce, işaretçi veri tipinin bir özelliğine dikkat çekmek istiyoruz. Her zamanki gibi, bir işaretçinin içine yerleştirdiğiniz adres, işaretçi ile aynı tipte olmalıdır. `int`'e işaret eden bir işaretçiye bir `float` değişkenin adresini atayamazsınız. Örneğin:

```
float flovar = 98.6;
int* ptrint = &flovar;  //HATA:float* degerini int* e atayamaz
```

Ancak, bunun bir istisnası vardır. Herhangi bir veri tipine işaret edebilen, bir tür genel amaçlı bir işaretçi mevcuttur. Buna, `void`'a işaret eden işaretçi denir ve şu şekilde tanımlanır:

```
void* ptr;        //ptr herhangi bir tipe isaret edebilir
```

Bu tür işaretçilerin belli özelleştirilmiş kullanımları vardır. Örneğin, işaret edilen veri tipinden bağımsız olarak işlev gören fonksiyonlara işaretçileri aktarmak için kullanılabilirler.

Sıradaki örnek, `void`'a işaret eden bir işaretçi kullanır; ayrıca, eğer `void` kullanmıyorsanız, işaretçi ile aynı tipte bir adresi işaretçiye atarken dikkatli olmanız gerektiğini gösterir. `PTRVOID` programın listesi şöyledir:

```
// ptrvoid.cpp
// void tipine isaret eden isaretci
#include <iostream>
using namespace std;

int main()
   {
   int intvar;             //tamsayi degisken
   float flovar;           //float degisken

   int* ptrint;
   float* ptrflo;          //int'e isaret eden isaretci tanimla
                           //float'a isaret eden isaretci tanimla
```



```
   void* ptrvoid;          //void'a isaret eden isaretci tanimla

   ptrint = &intvar;       //tamam, int*'den int*'e
// ptrint = &flovar;       //hatali, float*'tan int*'e
// ptrflo = &intvar;       //hatali, int*'ten float*'a
   ptrflo = &flovar;       //tamam, float*'tan float*'a
   ptrvoid = &intvar;      //tamam, int*'ten void*'a
   ptrvoid = &flovar;      //tamam, float*'tan void*'a
   return 0;
   }
```

`intvar`'ın adresini `ptrint`'e atayabilirsiniz çünkü her ikisi de `int` tipindedir. Fakat, `flovar`'ın adresini `ptrint`'e atayamazsınız çünkü ilki `float`, ikincisi ise `int*` tipindedir. Bununla birlikte, `ptrvoid`'e herhangi bir işaretçi değeri verilebilir. örneğin `int*`, çünkü bu, `void`'a işaret eden bir işaretçidir.

Alışılmamış bazı sebeplerden dolayı, eğer bir tür işaretçi tipini gerçekten bir diğerine atamanız gerekiyorsa, `reinterpret_cast` tip atamasını (cast) kullanabilirsiniz. `PTRVOID`'da açıklama şeklinde işaretlenerek çıkarılmış satırlar yerine bu tip ataması kullanılmış olsaydı, şu şekilde görünürdü:

```
ptrint = reinterpret_cast<int*>(flovar);
ptrflo = reinterpret_cast<float*>(intvar);
```

`reinterpret_cast` tip atamasının bu şekilde kullanımı tavsiye edilmez, fakat bazen zor bir durumdan kurtulmanın tek yolu da budur. Eski stil C tip atamaları da kullanılabilir ama, C++'ta bunları kullanmak her zaman kötü bir fikirdir. `reinterpret_cast` tip atamasının örneklerini "Akışlar ve Dosyalar" adlı Bölüm 12'de göreceğiz. `reinterpret_cast` tip ataması Bölüm 12'de bir veri tamponunun kullanımını değiştirmek amacıyla kullanılıyor.

İşaretçiler ve Diziler

İşaretçiler ve diziler arasında yakın bir ilişki vardır. Bölüm 7'de "Diziler ve Karakter Katarları" başlığı altında dizi elemanlarına nasıl erişildiğini gördük. Aşağıdaki program, `ARRNOTE`, bunu tekrar gözden geçiriyor.

```
// arrnote.cpp
// dizi notasyonu ile erisilen dizi
#include <iostream>
using namespace std;

int main()
   {                                    //dizi
   int intarray[5] = { 31, 54, 77, 52, 93 };

   for(int j=0; j<5; j++)               //her eleman icin,
      cout << intarray[j] << endl;      //degeri yazdir
   return 0;
   }
```

`cout` ifadesi, dizi elemanlarının her birini sırayla ekrana yazar. Örneğin, `j`, 3 iken, `intarray[j]` deyimi `intarray[3]` değerini alır ve dizinin dördüncü elemanına, yani 52'ye erişir. `ARRNOTE` programının çıktısı şöyledir:

```
31
54
77
52
93
```

Şaşırtıcı ama, dizi elemanları dizi notasyonunun yanı sıra işaretçi notasyonu ile de erişilebilir. Sıradaki örnek, **PTRNOTE**, **ARRNOTE** ile aynı; yalnızca, **PTRNOTE**'da işaretçi notasyonu kullanılıyor.

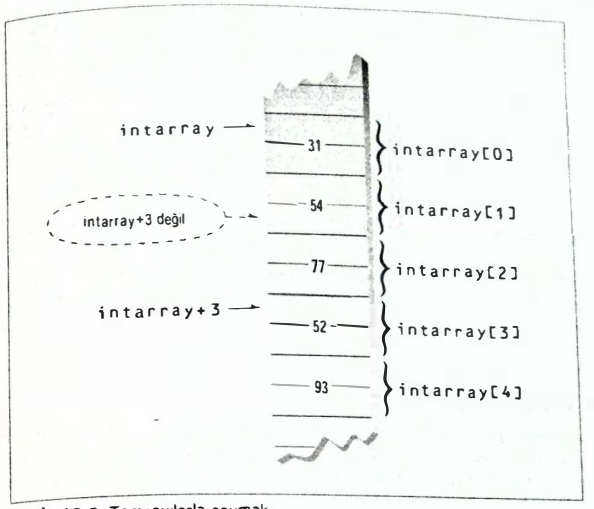
```
// ptrnote.cpp
// isaretci notasyonu ile erisilen dizi
#include <iostream>
using namespace std;

int main()
   {                                   //dizi
   int intarray[5] = { 31, 54, 77, 52, 93 };

   for(int j=0; j<5; j++)              //her eleman icin,
      cout << *(intarray+j) << endl;   //degeri yazdir
   return 0;
   }
```

PTRNOTE içindeki ***(intarray+j)** deyimi, **ARRNOTE**'taki **intarray[j]** ile tamamen aynı etkiye sahiptir. Programların çıktıları da aynıdır. Fakat ***(intarray+j)** deyimini nasıl yorumlarız? Varsayalım, j'nin değeri 3 olsun; böylece, bu deyim ***(intarray+3)** deyimi ile eşdeğer olur. Bu deyimin, dizinin dördüncü elemanının içeriğini (52) simgelemesini istiyoruz. Hatırlarsanız, dizinin ismi dizinin adresi demektir. Dolayısıyla, **intarray+j** deyimi, üzerine bir şey eklenmiş bir adrestir. **intarray+3** deyiminin **intarray**'e 3 byte eklenmesine neden olacağını bekleyebilirsiniz. Fakat bu, istediğimiz gibi bir sonuç üretmez: **intarray** bir tamsayı dizisidir ve bu dizinin içinde 3 byte ilerleyince ikinci elemanın ortasına gelinmiş olur; fakat, bu pek işe yaramaz. Biz Şekil 10.5'te gösterildiği gibi, dizideki dördüncü elemanı elde etmek istiyoruz, dördüncü byte'ı değil. (Bu şekil, tamsayıların 2 byte yer kapladığını varsayıyor.)

C++ derleyicisi, veri adresleri üzerinde aritmetik işlem yaparken verinin büyüklüğünü hesaba katacak kadar akıllıdır. Derleyici, **intarray**'in **int** tipinde bir dizi olduğunu bilir; çünkü, dizi bu şekilde tanımlanmıştır. Bu nedenle, **intarray+3** deyimini gördüğünde, bunu **intarray** içindeki dördüncü byte olarak değil de, dördüncü tamsayının adresi olarak yorumlar.

Fakat biz dördüncü dizi elemanının adresini değil, değerini istiyoruz. Değere ulaşmak için dereferans operatörünü (*) kullanıyoruz. Sonuçta elde edilen deyim j, 3 iken ***(intarray+3)** olur; bu da dördüncü dizi elemanının içeriğine, yani 52'ye karşılık gelir.

Şimdi, bir işaretçi tanımına, işaret edilen değişkenin tipinin neden dahil edilmesi gerektiğini anlıyoruz. Derleyicinin, işaretçinin bir **int**'e mi veya bir **double**'a mı işaret ettiğini bilmesi gerekir; derleyici bu sayede, dizinin elemanlarına erişmek için doğru bir aritmetik gerçekleştirebilir. Derleyici, **int** tipi söz konusu olduğunda indeks değerini 2 ile; fakat, **double** durumunda ise 8 ile çarpar.



ŞEKİL 10.5: Tamsayılarla saymak.

İşaretçi Sabitleri ve İşaretçi Değişkenleri

Varsayalım, dizi adresleri içinde adım adım gezmek için **intarray**'e **j** eklemektense artırım operatörünü kullanmak istiyorsunuz. Şöyle yazabilir misiniz: ***(intarray++)**?

Yanıt: Hayır. Bunun sebebi, bir sabitin değerini artıramamanızdan kaynaklanır (hatta hiçbir şekilde değiştiremezsiniz bile). **intarray** deyimi, sistemin dizinizi yerleştirmek üzere seçtiği adrestir ve program sona erene kadar dizi bu adreste kalacaktır. **intarray** bir işaretçi sabitidir. Nasıl ki **7++** diyemiyorsunuz, **intarray++** da diyemezsiniz. (Çok görevliliği (multitasking) destekleyen bir sistemde, değişken adresleri programın çalışması sırasında değişebilir. Aktif bir program diskle takas edilip, sonra belleğin bir başka konumuna yeniden yüklenebilir. Yine de, bu işlemler sizin programınız tarafından fark edilmez.)

Bir adresi artıramıyor olabilirsiniz ama, adres tutan bir işaretçiyi artırabilirsiniz. Sıradaki örnek olan **PTRINC**, bunun nasıl yapılabileceğini gösteriyor:

```
// ptrinc.cpp
// isaretci ile erisilen dizi
#include <iostream>
using namespace std;

int main()
   {
   int intarray[] = { 31, 54, 77, 52, 93 };       //dizi
   int* ptrint;                                   //int'e isaret eden isaretci
   ptrint = intarray;                        //intarray'e isaret eden isaretci

   for(int j=0; j<5; j++)                    //her eleman icin,
```

```
    cout << *(ptrint++) << endl;          //degeri yazdir
  return 0;
  }
```

Burada bir `int`'e işaret eden bir işaretçi tanımlıyoruz: `ptrint`. Bu işaretçiye `intarray` değerini; yani, dizinin adresini, veriyoruz. Artık dizi elemanlarının içeriklerine şu deyim ile erişebiliriz:

```
*(ptrint++)
```

`ptrint` değişkeni, `intarray` ile aynı adres değerinden başlar. Böylece, 31 değerini içeren birinci dizi elemanına, `intarray[0]`, önceki gibi erişilmesine imkan verir. Fakat, `ptrint` bir sabit olmayıp, bir değişken olduğu için artırılabilir. Artırıldıktan sonra ikinci dizi elemanına, `intarray[1]`, işaret eder. `*(ptrint++)` deyimi, bu kez, ikinci dizi elemanının içeriğini, yani 54'ü, simgeler. Döngü, deyimin sırayla dizi elemanlarının her birine erişmesini sağlar. `PTRINC` programının çıktısı, `PTRNOTE`'unki ile aynıdır.

İşaretçiler ve Fonksiyonlar

Bölüm 5'te, bir fonksiyona argüman aktarmanın üç yöntemi olduğuna dikkat çekmiştik: Değer olarak, referans yoluyla ve işaretçi üzerinden. Eğer fonksiyon kendisini çağıran programdaki değişkenleri değiştirme niyetindeyse, bu değişkenler değer olarak aktarılamaz, çünkü bu durumda fonksiyon, değişkenin yalnızca bir kopyasını elde edebilir. Bununla birlikte, bu durumda ya bir referans argümanı ya da bir işaretçi kullanılabilir.

Basit Değişkenleri Aktarmak

Öncelikle argümanların referans olarak nasıl aktarıldığını yeniden gözden geçireceğiz ve bunu, işaretçi argümanlarının aktarılması ile karşılaştıracağız. `PASSREF` programı referans olarak aktarmayı gösteriyor:

```
// passref.cpp
// referans olarak aktarilan argumanlar
#include <iostream>
using namespace std;

int main()
   {
   void centimize(double&);                  //prototip

   double var = 10.0;                        //var, 10 inc degerine sahip
   cout << "var = " << var << " inches " << endl;

   centimize(var);                           //var'i santimetreye cevir
   cout << "var = " << var << " centimeters " << endl;
   return 0;
   }
//-------------------------------------------------------------
void centimize(double& v)
   {
   v *= 2.54;                                //v, var ile ayni
   }
```


Burada, `main()` içindeki `var` değişkenini inçten santimetreye çevirmek istiyoruz. Değişken, `centimize()` fonksiyonuna referans olarak aktarılır. (Bu fonksiyonun prototipinde `double` veri tipinin peşinden gelen `&` simgesinin, argümanın referans olarak aktarıldığına işaret ettiğini hatırınızdan çıkarmayın.) `centimize()` fonksiyonu, orijinal değişkeni 2.54 ile çarpar. Fonksiyonun, değişkeni nasıl ele aldığına dikkat edin. Fonksiyon sadece `v` argüman ismini kullanır. `v` ve `var` aynı şeye verilen farklı isimlerdir.

`main()`, `var`'ı santimetreye çevirdikten hemen sonra sonucu ekranda gösterir. `PASSREF` programının çıktısı şöyledir:

```
var = 10 inches
var = 25.4 centimeters
```

Sıradaki örnek olan `PASSPTR`, işaretçilerin kullanıldığı, öncekine denk bir durumu gösterir:

```
// passptr.cpp
// isaretci uzerinden aktarilan argumanlar
#include <iostream>
using namespace std;

int main()
   {
   void centimize(double*);                  //prototip

   double var = 10.0;                        //var, 10 inc degerine sahip
   cout << "var = " << var << " inches " << endl;

   centimize(&var);                          //var'i santimetreye cevirir
   cout << "var = " << var << " centimeters " << endl;
   return 0;
   }
//-------------------------------------------------------------
void centimize(double* ptrd)
   {
   *ptrd *= 2.54;                            //*ptrd, var ile ayni
   }
```

`PASSPTR`'ın çıktısı `PASSREF`'inki ile aynıdır.

`centimize()` fonksiyonunun deklarasyonu aşağıdaki gibi yapılır. `centimize()` fonksiyonu, `double`'a işaret eden bir işaretçiyi argüman olarak alır.

```
void centimize(double*)       //arguman, double'a isaret eden bir isaretci
```

`main()` bu fonksiyonu çağırdığında, değişkenin adresini argüman olarak fonksiyona aktarır:

```
centimize(&var);
```

Bunun, referans ile aktarmadan farklı olarak, değişkenin kendisi olmadığını, fakat değişkenin adresi olduğunu hatırınızdan çıkarmayın. (Hatırlarsanız, referans ile aktarmada değişkenin kendisi argüman olarak aktarılıyordu.)

`centimize()` fonksiyonuna bir adres aktarıldığı için fonksiyon, bu adresteki değere erişebilmek için dereferans operatörünü kullanmalıdır: `*ptrd`.

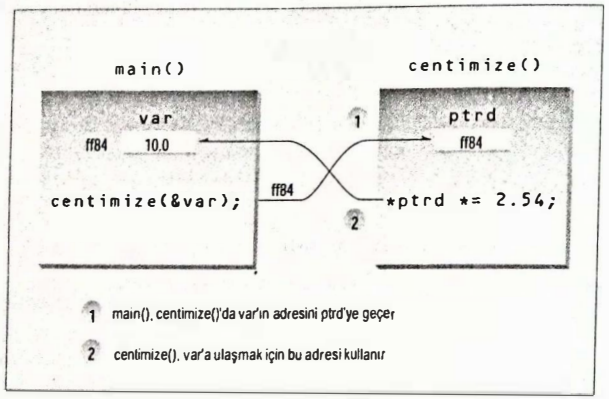
```
*ptrd *= 2.54;              //ptrd'in icerigini 2.54 ile carp
```

Elbette bu, aşağıdaki ifade ile aynıdır:

```
*ptrd = *ptrd * 2.54;   //ptrd'in icerigini 2.54 ile carp
```

Bu ifadede tek başına yer alan asteriks çarpma anlamına gelir. (Bu operatör gerçekten ortalıkta çok geziyor.)

ptrd, **var** değişkeninin adresini içerdiği için ***ptrd** üzerinde yapılan her türlü değişiklik aslında **var** üzerinde gerçekleştirilmiş olur. Şekil 10.6, fonksiyonun içinde ***ptrd**'ı değiştirmenin, fonksiyonu çağıran programda **var**'ı nasıl değiştirdiğini gösteriyor.



ŞEKİL 10.6: İşaretçinin fonksiyona aktarılması.

Bir fonksiyona argüman olarak bir işaretçi aktarmak bazı açılardan referans aktarmaya benzer. Her ikisi de, fonksiyonu çağıran programdaki değişkenin fonksiyon tarafından değiştirilmesine imkan verir. Yine de, kullanılan mekanizmalar farklıdır. Bir referans, orijinal değişken için bir takma isimde ibaret; ancak, bir işaretçi değişkenin adresini tutar.

Dizileri Aktarmak

Bölüm 7'den itibaren, dizilerin argüman olarak fonksiyonlara aktarılması ve dizi elemanlarının fonksiyon tarafından erişilmesi ile ilgili birçok örnek gördük. Bu bölüme kadar bu işlemler, işaretçileri öğrenmemiş olduğumuz için dizi notasyonu kullanılarak gerçekleştirildi. Ancak, fonksiyonlara dizi aktarıldığında, dizi notasyonu yerine işaretçi notasyonu kullanmak çok daha yaygın bir yaklaşımdır. **PASSARR** programı bunun nasıl gerçeklendiğini gösterir:

```
// passarr.cpp
// isaretci uzerinden aktarilan dizi
#include <iostream>
using namespace std;
const int MAX = 5;
                                    //dizinin eleman sayisi
```



```
int main()
   {
   void centimize(double*);              //prototip

   double varray[MAX] = { 10.0, 43.1, 95.9, 59.7, 87.3 };

   centimize(varray);                    //varray'in elemanlarini cm'ye cevir

   for(int j=0; j<MAX; j++)              //yeni dizi degerlerini goster
      cout << "varray [" << j << "]="
           << varray[j] << " centimeters" << endl;
   return 0;
   }
//-------------------------------------------------------------
void centimize(double* ptrd)
   {
   for(int j=0; j<MAX; j++)
      *ptrd++ *= 2.54;                   //ptrd, varray'in elemanlarina isaret eder
   }
```

Fonksiyonun prototipi **PASSPTR**'daki ile aynıdır; fonksiyonun tek argümanı **double**'a işaret eder. Bu dizi notasyonunda şu şekilde yazılır:

```
void centimize(double[]);
```

Yani, her ne kadar işaretçi söz dizimi daha yaygın olarak kullanılıyor olsa da, **double*** burada **double[]**'a eşdeğer olur.

Bir dizinin ismi, dizinin adresi olduğu için, fonksiyon çağrıldığında adres operatörünü (**&**) kullanmaya gerek yoktur:

```
centimize(varray);              //dizi adresini aktar
```

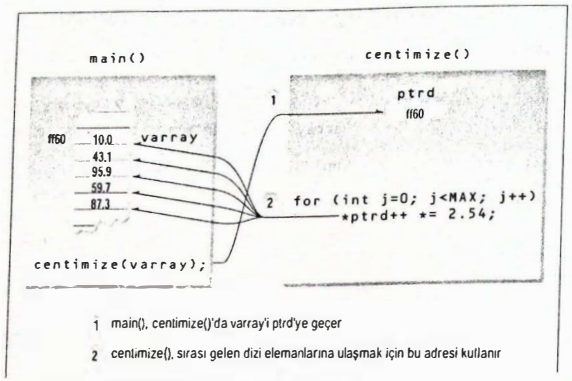
Bu dizi adresi, **centimize()**'da **ptrd** değişkeninin içine yerleştirilir. Dizinin her elemanına sırayla işaret etmesi için yalnızca **ptrd**'ı artırmamız yeter:

```
*ptrd++ *= 2.54;
```

Diziye nasıl erişildiği Şekil 10.7'de gösteriliyor. **PASSARR** programının çıktısı şöyledir:

```
varray[0] = 25.4 centimeters
varray[1] = 109.474 centimeters
varray[2] = 243.586 centimeters
varray[3] = 151.638 centimeters
varray[4] = 221.742 centimeters
```

Şimdi, sözdizimi ile ilgili bir meseleyi ele alalım: ***ptrd++** deyiminin işaretçiyi artırıp, işaretçi içeriğini artırmadığını nasıl bilebiliriz? Başka bir deyişle, derleyici bu deyimi ***(ptrd++)** olarak yani bizim istediğimiz gibi mi, yoksa **(*ptrd)++** olarak mı yorumlar? Görünen şu ki, * (dereferans operatörü olarak kullanıldığında) ve ++ aynı öncelik sırasına sahip. Ancak, aynı önceliğe sahip operatörler ikinci bir yöntem ile ayırt ediliyor: *birleşme özelliği* (*associativity*) ile. Birleşme özelliği, operatörlerin öncelik sırası ile ilgilidir. Derleyici, işlemlere sağdaki operatörle mi, yoksa soldaki operatörle mi başlayacağına bu özellik yardımıyla karar verir.



ŞEKİL 10.7: Bir fonksiyondan bir diziye erişmek.

Eğer bir grup operatör sağdan birleşme özelliğine (right associativity) sahipse, derleyici öncelikle deyimin sağ tarafında kalan işlemleri gerçekleştirir; sonra, sol taraftakilerle işleme devam eder. Tekli (unary) operatörler, mesela * veya ++, sağdan birleşme özelliğine sahiptir. Bu nedenle, bu deyim `*(ptrd++)` şeklinde yorumlanır ve işaretçinin işaret ettiğini değil de, işaretçiyi artırır. Yani, önce işaretçi bir kez artırılır ve elde edilen sonuca dereferans operatörü uygulanır.

Dizi Elemanlarını Sıralamak

Dizi elemanlarına erişmek için işaretçi kullanımına bir başka örnek olarak, şimdi bir dizinin elemanlarının nasıl sıralanacağını görelim. İki program örneği kullanacağız. İlkinde olayın temelini atacağız. İkinci program ise sıralama işlemini göstermek amacıyla ilkinin genişletilmiş bir versiyonundan ibaret olacak.

İşaretçilerle Sıralamak

Birinci program "Nesneler ve Sınıflar"adlı Bölüm 6'daki **REFORDER** programına benzer; yalnız, bu programda referans yerine işaretçi kullanılır. Burada kullanılan fonksiyon, kendisine argüman olarak aktarılan iki sayıyı, eğer ikinci sayı birinciden küçükse iki sayının yerlerini değiştirerek sıralar. **PTRORDER**'ın program listesi şöyledir:

```
// ptrorder.cpp
// isaretci kullanarak iki argumani siralar
#include <iostream>
using namespace std;

int main()
   {
   void order(int*, int*);                  //prototip

   int n1=99, n2=11;
   int n3=22, n4=88;                        //bir cift sirali, bir cift degil
```



```
   order(&n1, &n2);                         //her sayi ciftini sirala
   order(&n3, &n4);

   cout << "n1=" << n1 << endl;             //sayilarin tumunu yazdir
   cout << "n2=" << n2 << endl;
   cout << "n3=" << n3 << endl;
   cout << "n4=" << n4 << endl;
   return 0;
   }
//--------------------------------------------------------------
void order(int*numb1, int*numb2)            //iki sayiyi sirala
   {
   if(*numb1 > *numb2)                      //eger 1. Sayi 2.den buyukse,
      {
      int temp = *numb1;                    //ikisinin yerlerini degistir
      *numb1 = *numb2;
      *numb2 = temp;
      }
   }
```

`order()` fonksiyonu **REFORDER**'daki gibi çalışır. Yalnızca, sıralanacak sayıların adresleri fonksiyona aktarılır ve fonksiyon sayılara, işaretçileri kullanarak erişir. Yani, `*numb1`, `main()`'den aktarılan birinci argümana erişir, `*numb2` ise ikincisine erişir.

PTRORDER programının çıktısı şöyledir:

```
n1=11          <==== bunun ve
n2=99          <==== bunun yerleri değiştirildi, çünkü sıralı değillerdi
n3=22          <==== bunun ve
n4=88          <==== bunun yerleri değiştirilmedi, çünkü sıralıydılar
```

PTRORDER'daki `order()` fonksiyonunu, **PTRSORT** isimli sıradaki örneğimizde bir tamsayı dizisini sıralamak için kullanacağız.

```
// ptrsort.cpp
// isaretci kullanarak diziyi sirala
#include <iostream>
using namespace std;

int main()
   {
   void bsort(int*, int);                   //prototip
   const int N = 10;                        //dizinin buyuklugu
                                            //test dizisi
   int arr[N] = { 37, 84, 62, 91, 11, 65, 57, 28, 19, 49 };

   bsort(arr, N);                           //diziyi sirala

   for(int j=0; j<N; j++)                   //sirali diziyi yazdir
      cout << arr[j] << " ";
   cout << endl;
   return 0;
   }
//--------------------------------------------------------------
void bsort(int* ptr, int n)
   {
```

```
   void order(int*, int*);              //prototip
   int j, k;                            //dizi indeksleri

   for(j=0; j<n-1; j++)                 //distaki dongu
      for(k=j+1; k<n; k++)             //icteki dongu distaki kaldigi yerden baslar
         order(ptr+j, ptr+k);           //isaretci icerigini sirala
   }
//-------------------------------------------------------------
void order(int* numb1, int* numb2)      //iki sayiyi sirala
   {
   if(*numb1 > *numb2)                  //eger 1. Sayi 2.den buyukse,
      {
      int temp = *numb1;                //yerlerini degistir
      *numb1 = *numb2;
      *numb2 = temp;
      }
   }
```

`main()` içindeki, tamsayılardan oluşan `arr` dizisine başlangıçta sıralanmamış ilk değerler atanır. Dizinin adresi ve eleman sayısı `bsort()` fonksiyonuna aktarılır. Bu fonksiyon diziyi sıralar ve sıralı değerleri ekrana yazdırır. `PTRSORT` programının çıktısı şöyledir:

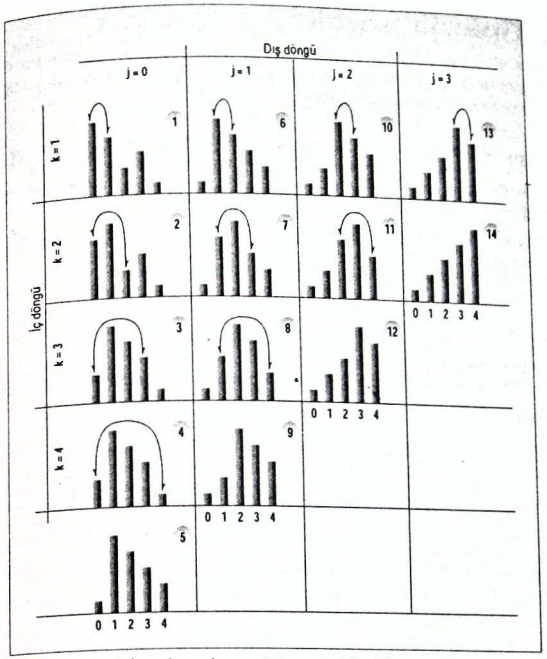
```
11 19 28 37 49 57 62 65 84 91
```

Kabarcık Sıralama (Bubble Sort)

`bsort()` fonksiyonu, kabarcık sıralama yöntemi kullanarak diziyi sıralar. Kabarcık sıralama (yavaşlığı ile kötü bir üne sahip olsa da), basit bir sıralama yöntemidir. Dizinin elemanlarını artan sırada sıralamak istediğimizi varsayarsak, bu yöntem şöyle çalışır: Önce, dizinin birinci elemanı (`arr[0]`) sırayla dizinin diğer elemanlarının her biri ile (ikinci elemandan başlayarak) karşılaştırılır. Eğer birinci eleman, diğerlerinin herhangi birinden büyükse, bu ikisinin yerleri değiştirilir. Bu işlem bittiğinde, en azından birinci elemanın sıraya girmiş olduğunu biliyoruz; birinci eleman artık dizinin en küçük elemanıdır. Sonra, ikinci eleman, üçüncü elemandan başlayarak sırayla dizinin diğer elemanları ile karşılaştırılır ve eğer daha büyükse yeri değiştirilir. Bu işlemi bitirdiğimizde, ikinci elemanın dizideki ikinci en küçük değer olduğunu biliriz. Bu işlem sondan bir önceki dizi elemanına kadar tüm elemanlar için sürdürülür. Sondan bir önceki elemana gelindiğinde dizinin sıralanmış olduğu varsayılır. Şekil 10.8'de kabarcık sıralamanın (`PTRSORT`'takinden birkaç tane daha az eleman ile) nasıl gerçekleştirildiği gösteriliyor.

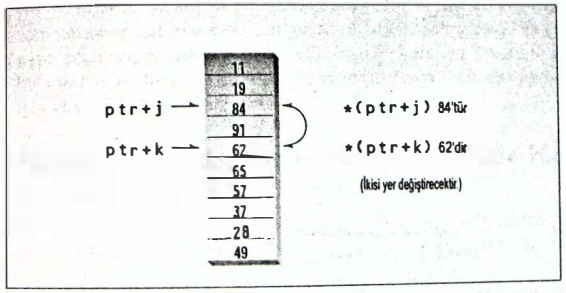
`PTRSORT`'ta, birinci pozisyondaki sayı olan 37, sırayla dizinin her elemanıyla karşılaştırılır ve 11 ile yeri değiştirilir. İkinci pozisyondaki sayı olan 84 değeri ile işleme başlanır ve her elemanla karşılaştırılır. 62 ile yeri değiştirilir (artık 62 ikinci pozisyonda yer alır); sonra 62, 37 ile yer değiştirir; 37, 28 ile; 28 de 19 ile yer değiştirir. Üçüncü pozisyondaki sayı (yine 84) 62 ile yer değiştirir; 62, 57 ile; 57, 37 ile; 37 de 28 ile yer değiştirir. Bu işlem dizi sıralanana kadar sürer.

`PTRSORT` programındaki `bsort()` fonksiyonu, her biri bir işaretçiyi kontrol eden iç içe iki döngü içerir. Dıştaki döngü, `j` döngü değişkenini; içteki döngü, `k` döngü değişkenini kullanır. `ptr+j` ve `ptr+k` deyimleri, döngü değişkenleri tarafından belirlenen dizinin çeşitli elemanlarına işaret ederler. `ptr+j` deyimi, birinci elemandan (en üstteki) başlayarak tamsayılar üzerinden adını adım ilerler ve sondan bir önceki elemana (en alttaki) ulaşır. Dıştaki döngüde `ptr+j` deyiminin her pozisyonu için, içteki döngüde `ptr+k` deyimi, `ptr+j`'in bir altındaki pozisyona işaret ederek başlar ve dizinin en altına doğru ilerler. İçteki döngünün her dönüşünde `ptr+j` ve `ptr+k` tarafından işaret edilen elemanlar `order()` fonksiyonu kullanılarak karşılaştırılır. Eğer ilki ikincisinden büyükse, her ikisinin yerleri değiştirilir. Bu işlem Şekil 10.9'da açıklanmıştır.



ŞEKİL 10.8: Kabarcık sıralama yönteminin işleyişi.

`PTRSORT` örneği ile işaretçilerin gücü açığa çıkmaya başlar. İşaretçiler, dizi elemanları ve isimleri belirli bir fonksiyon tarafından bilinmeyen diğer değişkenler üzerinde işlem yapmak için tutarlı ve etkili bir yöntem sunarlar.



ŞEKİL 10.9: `PTRSORT`'un işleyişi.

İşaretçiler ve C Tipinde Karakter Katarları

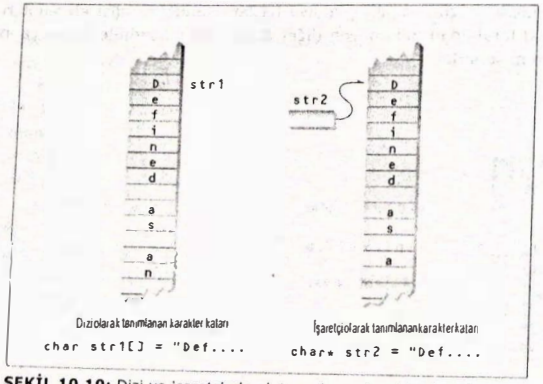
Bölüm 7'de vurguladığımız gibi, bir C tipinde karakter katarı sadece `char` tipinde bir diziden ibarettir. Bu nedenle, işaretçi notasyonu, tıpkı herhangi bir dizi elemanına uygulanabildiği gibi, karakter katarlarındaki karakterlere de uygulanabilir.

Karakter Katarı Sabitlerine İşaret Eden İşaretçiler

İşte bir örnek, `TWOSTR`. Bu programda iki karakter katarı tanımlanır. Bu tanımların biri dizi notasyonu kullanılarak, diğeri de işaretçi notasyonu kullanılarak gerçekleştirilir:

```
// twostr.cpp
// dizi ve isaretci notasyonu kullanilarak tanimlanan karakter katarlari
#include <iostream>
using namespace std;

int main()
   {
   char str1[] = "Defined as an array";
   char* str2 = "Defined as a pointer";

   cout << str1 << endl;          //her iki karakter katarini ekranda goster
   cout << str2 << endl;
// str1++;                        //bunu yapamazsiniz; str1 bir sabittir
   str2++;                        //bu tamam, str2 bir isaretcidir

   cout << str2 << endl;          //simdi str2'nin basi "efined..." seklindedir
   return 0;
   }
```

Birçok açıdan bu iki tip tanım eşdeğerdir. Her iki karakter katarını da, örnekte gösterildiği gibi, ekranda yazdırabilirsiniz, fonksiyon argümanı olarak kullanabilirsiniz vs. Fakat ikisi arasında ince bir fark vardır: `str1` bir adres, yani bir işaretçi sabiti, `str2` ise işaretçi değişkenidir. Bu nedenle, programdan görülebileceği gibi, `str2` değiştirilebilir, fakat `str1` değiştirilemez. Şekil 10.10, bu iki tip karakter katarının bellekte nasıl göründüğünü gösteriyor.



ŞEKİL 10.10: Dizi ve işaretçi olarak tanımlanan karakter katarları.

`str2` işaretçi olduğu için onu artırabiliriz; ancak, bunu bir kez yaparsak, `str2` artık karakter katarındaki ilk karaktere işaret etmez. `TWOSTR`'ın çıktısı şöyledir:

```
Defined as an array
Defined as a pointer
efined as a pointer          <==== str2++ ifadesinin ardından 'D' harfi gitti)
```

Bir işaretçi olarak tanımlanan bir karakter katarı, dizi olarak tanımlanan karakter katarına kıyasla çok daha esnektir. Aşağıdaki örneklerde bu esneklikten faydalanılır.

Fonksiyon Argümanı Olarak Karakter Katarları

Şimdi, bir karakter katarının fonksiyon argümanı olarak kullanıldığı bir örnek verelim. Fonksiyon, her karaktere sırayla erişerek karakter katarını ekrana basmaktan ibarettir. `PTRSAT` programının listesi şöyledir:

```
// ptrstr.cpp
// bir karakter katarini isaretci notasyonu ile gosterir
#include <iostream>
using namespace std;

int main()
   {
   void dispstr(char*);                  //prototip
   char str[] = "Idle people have the least leisure.";

   dispstr(str);                         // karakter katarini goster
   return 0;
   }
//-------------------------------------------------------------
void dispstr(char*ps)
   {
   while( *ps )                          //null karakterine kadar,
      cout << *ps++;                     //karakterleri yazdir
   cout << endl;
   }
```

`str` dizi adresi, `disptr()` fonksiyonunu çağırırken argüman olarak kullanılır. Bu adres bir sabittir, fakat fonksiyona değer olarak aktarıldığı için `disptr()` içinde bu argümanın bir kopyası hazırlanır. Bu kopya bir işaretçi olan `ps`'tir. Bir işaretçi değiştirilebilir. Bu nedenle, fonksiyon, karakter katarını ekranda göstermek için `ps`'i artırır. `*ps++` deyimi, karakter katarının peş peşe gelen karakterlerini döndürür. Döngü, karakter katarının sonunda null karakterini (`'\0'`) bulana kadar devam eder. Bu karakterin değeri 0 olduğu için, yani *false* değerini simgelediği için `while` döngüsü bu aşamada sona erer.

İşaretçi Kullanarak Bir Karakter Katarını Kopyalamak

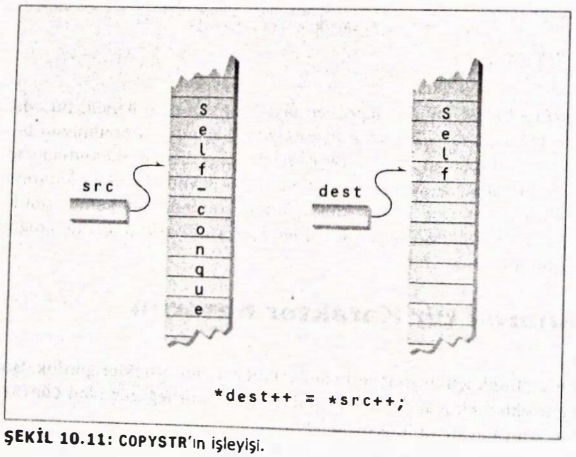
Bir dizinin değerlerini elde etmek için işaretçilerin kullanıldığına dair örnekler gördük. İşaretçiler ayrıca, bir diziye değer eklemek için de kullanılabilir. Sıradaki örneğimiz olan `COPYSTR`, bir karakter katarını diğerine kopyalayan bir fonksiyon içerir:

```
// copystr.cpp
// isaretciler yardimiyla bir karakter katarini digerine kopyalar
#include <iostream>
using namespace std;

int main()
   {
   void copystr(char*, const char*);       //prototip
   char* str1 = "Self-conquest is the greatest victory.";
   char str2[80];                          //bos karakter katari

   copystr(str2, str1);                    //str1'i str2'ye kopyala
   cout << str2 << endl;                   //str2'yi goster
   return 0;
   }
//--------------------------------------------------------------
void copystr(char* dest, const char* src)
   {
   while(*src)                             //null karakterine kadar,
      *dest++ = *src++;                    //karakterleri src'den dest'e kopyala
   *dest = '\0';                           //dest'i sonlandir
   }
```

Burada, programın **main()** kısmı, **str1**'i **str2**'ye kopyalamak için **copystr()** fonksiyonunu çağırır. Bu fonksiyonda, şu deyim

```
*dest++ = *src++;
```

src tarafından işaret edilen adresteki değeri alır ve bunu **dest** tarafında işaret edilen adrese yerleştirir. Her iki işaretçi de sonra artırılır. Böylece, döngünün bir sonraki adımında bir sonraki karakter transfer edilebilir. Döngü, **src** içinde bir null karakteri bulunduğu zaman sona erer. Bu noktada **dest**'e de bir null karakteri eklenir ve fonksiyon döner. Şekil 10.11 işaretçilerin karakter katarları içinde nasıl ilerlediğini gösteriyor.



ŞEKİL 10.11: COPYSTR'ın işleyişi.

Kütüphane Karakter Katarı Fonksiyonları

Karakter katarları için şimdiye kadar kullandığımız kütüphane fonksiyonlarının birçoğu, işaretçi notasyonu kullanılarak belirtilen karakter katarı argümanları içerir. Örnek olarak derleyicinizin dokümantasyonunda (veya **STRING.H** başlık dosyasında) **strcpy()** fonksiyonunun tanımına bakabilirsiniz. Bu fonksiyon bir karakter katarını diğerine kopyalar. Bunu, **COPYSTR** örneğindeki kendi tanımladığımız **copystr()** fonksiyonu ile karşılaştırabiliriz. **strcpy()** kütüphane fonksiyonunun söz dizimi şöyledir:

```
char* strcpy(char* dest, const char* src);
```

Bu fonksiyon **char*** tipinde iki argüman alır. (Bir sonraki, "**const** Niteleyicisi ve İşaretçiler" konusunda, **const**'un bu bağlamda ne anlama geldiği açıklanıyor.) **strcpy()** fonksiyonu ayrıca, bir **char**'a işaret eden bir işaretçi döndürür; bu, **dest** karakter katarının adresidir. Diğer yönlerden bu fonksiyon kendi tanımladığımız **copystr()**'a çok benzer şekilde çalışır.

const Niteleyicisi ve İşaretçiler

İşaretçi deklarasyonlarında **const** niteleyicisinin kullanımı karışık gelebilir; çünkü, bu nereye yerleştirildiğine bağlı olarak iki anlam taşıyabilir. Aşağıdaki ifadeler muhtemel iki durumu gösteriyor:

```
const int* cptrInt;     //cptrInt, sabit bir int'e isaret eden isaretci
int* const ptrcInt;     //ptrcInt, int'e isaret eden sabit bir isaretci
```

İlk deklarasyona uyarsanız, **cptrInt**'in kendisini değiştirebiliyor olmanıza rağmen, işaret ettiği değeri değiştiremezsiniz. İkinci deklarasyona uyarsanız, **ptrcInc**'in işaret ettiğini değiştirebilirsiniz; fakat, **ptrcInc**'in kendi değerini değiştiremezsiniz. İkisi arasındaki farkı, açıklamalarda da belirtildiği gibi, deklarasyonları sağdan sola doğru okuyarak hatırlayabilirsiniz. İşaretçiyi ve işaretçinin işaret ettiğini sabit yapmak için **const** niteleyicisini her iki pozisyonda kullanabilirsiniz.

Az önce gösterilen **strcpy()** fonksiyonunun deklarasyonunda, **const char* src** argümanı, **src** tarafından işaret edilen karakterlerin **strcpy()** tarafından değiştirilemeyeceğini belirtir. **src** işaretçisinin kendisinin değiştirilemeyeceği anlamına gelmez. Bunun gerçekleştirilmesi için argüman deklarasyonunun **char* const src** şeklinde olması gerekir.

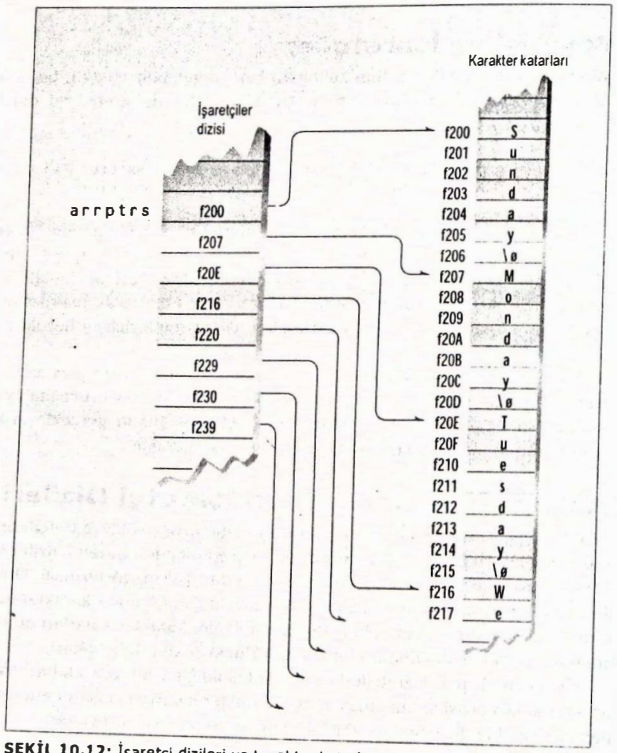
Karakter Katarlarına İşaret Eden İşaretçi Dizileri

int tipinde veya **float** tipinde değişkenlerin dizileri olduğu gibi, aynı şekilde işaretçilerin dizileri de olabilir. Bunun en yaygın kullanımı karakter katarlarına işaret eden işaretçi dizileridir.

Bölüm 7'de **STRARAY** programı **char*** karakter katarlarının bir dizisini göstermişti. Önceden vurguladığımız gibi, karakter katarı dizileri kullanmak dezavantajlıdır, çünkü karakter katarlarını tutan alt dizilerin tümü aynı uzunlukta olmalıdır. Bu nedenle, karakter katarları alt dizilerden daha kısa olduğunda bellek israf edilmiş olur (Bölüm 7'deki Şekil 7.10'a bakın).

Şimdi, bu problemi çözmek için işaretçilerin nasıl kullanıldığına bir göz atalım. **STRARAY** programını, bir karakter katarı dizisi tanımlamak yerine, karakter katarlarına işaret eden bir işaretçi dizisi tanımlayacak şekilde değiştirelim. **PTRTOSTR** programının listesi şöyledir:

```
// ptrtostr.cpp
// karakter katarlarina isaret eden isaretci dizisi
#include <iostream>
using namespace std;
const int DAYS = 7;                  //dizideki isaretci sayisi

int main()
   {                                 //char'a isaret eden isaretci dizisi
   char* arrptrs[DAYS] = { "Sunday", "Monday", "Tuesday",
                  "Wednesday", "Thursday",
                  "Friday", "Saturday" };

   for(int j=0; j<DAYS; j++)     // karakter katarlarinin hepsini goster
      cout << arrptrs[j] << endl;
   return 0;
   }
```



ŞEKİL 10.12: İşaretçi dizileri ve karakter katarları.

Bu programın çıktısı STRARAY'in çıktısı ile aynıdır:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Karakter katarları bir dizinin parçası olmadıkları zaman C++ bunları bellekte peş peşe yerleştirir; böylece, bellek israf edilmemiş olur. Yine de, karakter katarlarını bulmak için, bu karakter katarlarına işaret eden işaretçileri tutan bir dizi olması gerekir. Karakter katarının kendisi **char** tipinde bir dizidir; bu nedenle, karakter katarlarına işaret eden işaretçilerin dizisi aslında **char**'a işaret eden işaretçilerin dizisidir. **PTRTOSTR** programındaki **arrptrs** tanımının anlamı budur. Şimdi de, bir karakter katarının her zaman tek bir adresle simgelendiğini hatırlayın: Bu, karakter katarının içindeki ilk karakterin adresidir. Dizide işte bu adresler saklanırlar. Şekil 10.12'de bu durum gösteriliyor.

Bellek Yönetimi: new ve delete

Bellekte yer ayırmak için dizilerin kullanıldığı birçok örnek gördük. Şu ifade, 100 tane tamsayı için bellekte yer ayırır:

```
int arr1[100];
```

Diziler, veri saklamak için kullanışlı bir yaklaşımdır, ancak ciddi bir dezavantajları da vardır: Programı yazdığımız sırada dizinin ne kadar büyük olacağını bilmemiz gerekir. Programın çalışırken dizi boyutunu belirtmesini bekleyemeyiz. Aşağıdaki yaklaşım işe yaramayacaktır:

```
cin >> size;              //kullanicidan boyutu al
int arr[size];            //hatali;dizi boyutu sabit olmali
```

Derleyici, dizi boyutunun sabit olmasını gerektirir.

Ancak birçok durumda, programın çalıştığı ana kadar ne kadar belleğe ihtiyacımız olduğunu bilmeyiz. Örneğin, kullanıcının tuşladığı bir karakter katarını saklamak isteyebiliriz. Bu durumda, muhtemel en büyük karakter katarını tutacak boyutta bir dizi tanımlayabiliriz; fakat, bu bellek harcar. ("Standart Şablon Kütüphanesi" adlı Bölüm 15'te göreceğimiz gibi, bir çeşit genişletilebilir dizi olan vektörü de ayrıca kullanabilirsiniz.)

new Operatörü

C++, bellek bloklarını elde etmek için farklı bir yaklaşım sunuyor: **new** operatörü. Bu çok yönlü operatör, işletim sisteminden bellek alır ve bu belleğin başladığı yere işaret eden bir işaretçi döndürür. **NEWINTRO** örneği, **new** operatörünün nasıl kullanıldığını gösteriyor:

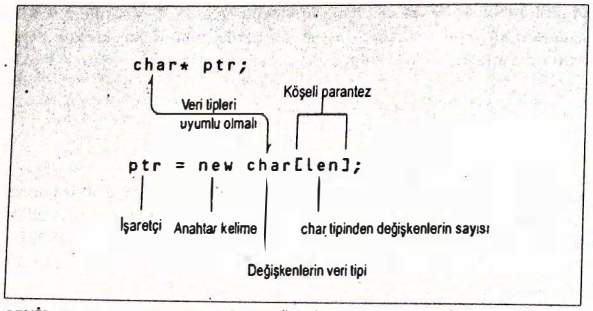
```
// newintro.cpp
// new operatoru
#include <iostream>
#include <cstring>                    //strlen icin
using namespace std;
```

```
int main()
  {
  char* str = "Idle hands are the devil 's workshop.";
  int len = strlen(str);          //str'in uzunlugunu al

  char* ptr;                      //char'a isaret eden bir isaretci tanimla
  ptr = new char[len+1];          //bellekten yer ayir: karakter katari +'\0'

  strcpy(ptr, str);               //ptr'in isaret ettigi yeni bellek
                                  //bolgesine str'i kopyala
  cout << "ptr=" << ptr << endl;  //ptr'in str'in icinde oldugunu goster

  delete [] ptr;                  //ptr'in bellegini serbest birak
  return 0;
  }
```

Aşağıdaki deyim **str** karakter katarını tutmak için yeterince büyük bir bellek parçasına bir işaretçi döndürür:

```
ptr = new char[len+1];
```

str'ın uzunluğu, yani **len**, **strlen()** kütüphane fonksiyonu kullanılarak bulunur; ilaveten karakter katarının sonundaki null karakteri ('\0') için de ekstra bir byte eklememiz lazımdır. Şekil 10.13'te **new** operatörü kullanılan bir ifadenin sözdizimi gösteriliyor. Dizi boyutunu köşeli parantez içine almayı hatırınızdan çıkarmayın; eğer kazara normal parantez kullanırsanız derleyici buna itiraz etmez; fakat, elde edilecek sonuç hatalı olur.

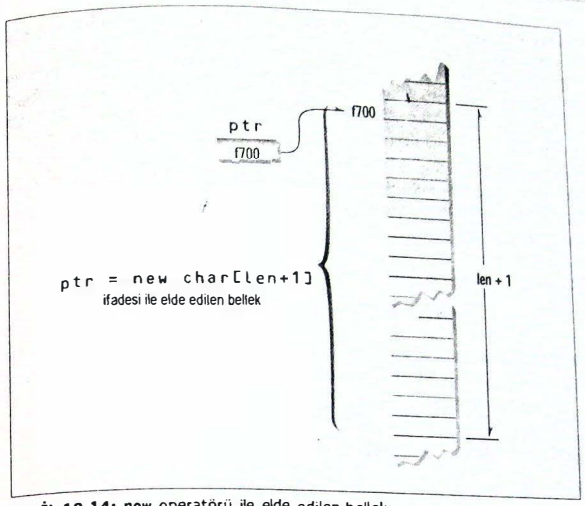


ŞEKİL 10.13: **new** operatörünün sözdizimi.

Şekil 10.14, **new** ile elde edilen belleği ve bu bellek parçasına işaret eden işaretçiyi gösteriyor.

NEWINTRO programında, **str** karakter katarını, yeni hazırlanan ve **ptr** tarafında işaret edilen bellek alanına kopyalamak için **strcpy()** fonksiyonunu kullanıyoruz. Bu bellek alanını **str**'ın uzunluğuna eşit bir boyutta hazırladığımız için karakter katarı buraya tam olarak oturur. Bunun neticesinde, **NEWINTRO**'nun çıktısı şöyle olur:

```
ptr = Idle hands are the devil 's workshop.
```

ŞEKİL 10.14: **new** operatörü ile elde edilen bellek.

C programcıları, **new** operatörünün **malloc()** kütüphane fonksiyonları ailesine benzer bir role sahip olduğunu fark edeceklerdir. **new** yaklaşımı, ilgili veri tipine bir işaretçi döndürdüğü için **malloc()**'a göre daha üstündür; çünkü **malloc()**'ta kullanılan işaretçi, ilgili tipe bir tip ataması (cast) yardımıyla dönüştürülmek zorundadır. Ayrıca başka avantajları daha vardır.

C programcıları, önceden tahsis edilmiş bir belleğin boyutunu değiştirmek için kullanılan **realloc()** fonksiyonunun C++'ta bir eşdeğeri olup olmadığını merak edebilirler. Üzgünüz, C++'ta **renew** diye bir operatör yoktur. Bunun için geriye dönüp **new** ile daha büyük (veya daha küçük) bir alan hazırlamanız ve verilerinizi eski alandan yenisine kopyalamanız gerekir.

delete Operatörü

Eğer programınız **new** kullanarak bellekten büyük miktarlarda yer ayırıyorsa, eninde sonunda mevcut belleğin tümü tahsis edilmiş olur ve sistem çöker. Belleğin emniyetli ve verimli kullanımını garanti etmek için **new** operatörü, belleği işletim sistemine döndüren **delete** operatörü ile eşlendirilmiştir. **NEWINTRO** programındaki şu ifade, **ptr**'ın işaret ettiği belleği sisteme döndürür:

```
delete[] ptr;
```

Aslında, **NEWINTRO**'da bu operatörü kullanmaya ihtiyaç yoktur, çünkü program sona erdiğinde zaten bellek otomatik olarak sisteme döndürülür. Yine de, diyelim ki **new** operatörünü bir fonksiyonda kullanıyorsunuz. Eğer fonksiyon, bu belleği gösteren işaretçi için bir yerel değişken kullanıyorsa, fonksiyon sona erdiğinde bu işaretçi yok edilecektir; fakat bellek, programın geri kalanı tarafından erişilemeyen bir yer kaplayarak "öksüz" kalacaktır. Bu nedenle, bir bellek

alanıyla işiniz bittiğinde **delete** operatörü ile onu elinizden çıkarmanız her zaman iyi ve çoğunlukla da gerekli olan bir uygulamadır.

Belleği elden çıkarmak, bu belleğe işaret eden işaretçiyi de elden çıkarmak anlamına gelmez (**NEWINTRO** programı söz konusu ise); üstelik, işaretçinin içindeki adres değerini de değiştirmez. Ancak, bu adres artık geçerli de olmaz; işaretçinin işaret ettiği bellek tamamen farklı bir şeylerle değiştirilmiş bile olabilir. İşaret ettiği bellek elden çıkarılmış olan işaretçileri kullanmamak konusunda dikkatli olun.

delete'in peşinden gelen köşeli parantezler bir diziyi elden çıkardığınıza işaret eder. Eğer **new** kullanarak tek bir nesne tanımlamışsanız, bu nesneye ait belleği temizlerken köşeli parantez kullanmanıza gerek yoktur.

```
ptr = new SomeClass;     //tek bir nesne icin bellek tahsis et
...
delete ptr;              //delete'ten sonra koseli parantez yok
```

Yine de, nesne dizilerini temizlerken köşeli parantez kullanmayı unutmayın. Köşeli parantez kullanırsanız, dizinin tüm üyelerinin temizlenmesini ve yok edici fonksiyonun her üye için tek tek çağrılmasını garanti etmiş olursunuz.

New Kullanarak Bir Karakter Katarı Sınıfı Tanımlamak

new operatörüne genellikle kurucu fonksiyonlarda rastlarız. Örnek olarak, en son "Operatörlerin Aşırı Yüklenmesi" adlı Bölüm 8'deki örneklerde, mesela **STRPLUS**'ta karşımıza çıkan **String** sınıfının üzerinde değişiklikler yapacağız. Hatırlarsanız, bu sınıfın sahip olduğu kusur, tüm **String** nesnelerinin bellekte aynı sabit uzunlukta yer kaplamasıydı. Bu sabit uzunluktan daha kısa bir karakter katarı, bellek israf etmiş olurdu; daha uzunu ise – eğer kazara böyle bir karakter katarı üretilmişse – dizinin ötesine uzandığı için sistemin çökmesine neden olabilirdi. Bir sonraki örneğimiz, tam olarak doğru miktarda bellek tahsis etmek için **new** operatörünü kullanır. **NEWSTR**'ın program listesi şöyledir:

```
// newstr.cpp
// karakter katarlari icin bellek almak amaciyla new kullanmak
#include <iostream>
#include <cstring>          //strcpy() vs icin
using namespace std;
////////////////////////////////////////////////////////////
class String          //kullanici tarafindan tanimlanan karakter katari tipi
   {
   private:
      char* str;                          //string'e isaret eden isaretci
   public:
      String(char* s)                     //kurucu fonksiyon,tek argumanli
         {
         int length = strlen(s);          //string argumaninin uzunlugu
         str = new char[length+1];        //bellek al
         strcpy(str, s);                  //argumani oraya kopyala
         }
      ~String()                           //destructor
         {
         cout << "Deleting str.\n";
         delete[] str;                    //bellegi serbest birak
         }
      void display()                      //String'i goster
         {
         cout << str << endl;
         }
   };
////////////////////////////////////////////////////////////
int main()
   {                                      //1-argumanli kurucu fonksiyon kullanir
   String s1 = "Who knows nothing doubts nothing.";
   cout << "s1=";                         // karakter katarini goster
   s1.display();
   return 0;
   }
```

Bu programın çıktısı şöyledir:

```
s1=Who knows nothing doubts nothing.
Deleting str.
```

String sınıfının sadece bir tane veri öğesi vardır: **str** adında ve **char**'a işaret eden bir işaretçi. Bu işaretçi, **String** nesnesinde tutulan karakter katarına işaret eder. Nesnenin içinde karakter katarını tutmak için bir dizi yoktur. Karakter katarı başka bir yerde saklanır; yalnızca, karakter katarına işaret eden işaretçi **String** sınıfının bir üyesi olarak yer alır.

NEWSTR Programında Kurucu Fonksiyon

Bu örnekteki kurucu fonksiyon normal bir **char*** karakter katarını argüman olarak alır. Kurucu fonksiyon, **new** operatörünü kullanarak bu karakter katarı için bellekte yer ayırır; **str**, yeni ayrılan belleğe işaret eder. Kurucu fonksiyon daha sonra **strcpy()** fonksiyonunu kullanarak karakter katarını bu yeni bellek bölgesine kopyalar.

NEWSTR Programında Yok Edici Fonksiyon

Şimdiye kadar örneklerimizde çok fazla yok edici fonksiyon görmedik. Fakat, artık **new** ile bellek ayırdığımız için yok edici fonksiyonlar önem kazanmaya başlıyor. Eğer bir nesne tanımladığımızda bunun için bellekte yer ayırıyorsak, nesneye artık ihtiyacımız kalmadığında da ona ayrılan bellek alanını elden çıkarmak son derece mantıklıdır. Bölüm 6'dan hatırlayacağımız gibi, bir yok edici fonksiyon, bir nesne ortadan kaldırılmak istendiğinde otomatik olarak çağrılan bir rutindir. **NEWSTR** programındaki yok edici fonksiyon şöyledir:

```
~String()
   {
   cout << "Deleting str.";
   delete[] str;;
   }
```

Bu yok edici fonksiyon, nesne tanımlanırken tahsis edilen belleği, sisteme geri verir. Programın çıktısına bakarak, yok edici fonksiyonun programın sonunda çalıştırıldığını söyleyebilirsiniz. Nesneler (diğer değişkenler gibi) genellikle, içinde tanımlandıkları fonksiyon sona erdiğinde yok edilirler. Bu yok edici fonksiyon, **String** nesnesine tahsis edilen belleğin, nesne ortadan kaldırıldığında belirsiz olarak kalmasındansa, sisteme döndürülmesini garanti ediyor.

Yok edici fonksiyonları kullanırken **NEWSTR** programında gösterildiği gibi, ortaya çıkabilecek potansiyel bir pürüze dikkat çekmek istiyoruz. Eğer bir **String** nesnesini, diyelim ki **s2 = s1** gibi bir ifade kullanarak, bir diğerine kopyalıyorsanız, gerçekte sadece esas (**char***) karakter katarına işaret eden işaretçiyi kopyalıyorsunuz. Artık her iki nesne de bellekte aynı karakter katarına işaret eder. Fakat, şimdi bir karakter katarını yok ederseniz, yok edici fonksiyon **char*** karakter katarını ortadan kaldıracaktır, böylece diğer nesne geçersiz bir işaretçi ile kalacaktır. Böyle bir durum güç fark edilebilir, çünkü nesneler açık olmayan yöntemlerle yok edilebilirler; mesela, içinde yerel bir nesne tanımlanan fonksiyon döndüğünde yerel nesnenin yok edilmesi gibi. Bölüm 11'de, bir karakter katarına kaç tane **String** nesnesinin işaret ettiğini sayan daha akıllı bir yok edici fonksiyonun nasıl tanımlandığını öğreneceğiz.

Nesnelere İşaret Eden İşaretçiler

İşaretçiler, temel veri tiplerinin ve dizilerin yanı sıra nesnelere de işaret edebilirler. Aşağıdaki gibi bir ifadeyle tanımlanan ve bir isim verilen birçok nesne örneği gördük:

```
Distance dist;
```

Burada **dist** adında bir nesne **Distance** sınıfına ait olarak tanımlanır.

Yine de, programlarımızı yazdığımız esnada kimi zaman kaç tane nesne tanımlayacağımızı bilmeyiz. Böyle bir durum söz konusu olduğunda, nesneleri program çalıştırken tanımlamak için **new** operatörünü kullanabiliriz. Önceden gördüğümüz gibi, **new**, isim verilmemiş bir nesneye bir işaretçi döndürür. Şimdi kısa bir örnek programa bakalım. **ENGLPTR**, nesne tanımlamak için mümkün olan iki yöntemi karşılaştırır.

```
// englptr.cpp
// uye fonksiyonlara isaretcilerle erismek
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance       //Ingiliz olcu sistemine dayanan Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      void getdist()                  //uzakligi kullanicidan al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }
      void showdist()                 //uzakligi goruntule
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist;                     //ismi olan bir Distance nesnesi tanimla
   dist.getdist();                    //nokta operatoru kullanarak
   dist.showdist();                   //nesnenin uyelerin eris

   Distance* distptr;              //Distance'a isaret eden isaretci
   distptr = new Distance;         //yeni Distance nesnesine isaret eden isaretci
```



```
   distptr->getdist();            // -> operatoru kullanarak
   distptr->showdist();           //nesnenin uyelerine eris
   cout << endl;
   return 0;
   }
```

Bu program önceki bölümlerde gördüğümüz İngiliz ölçü sistemine dayanan **Distance** sınıfının bir varyasyonunu kullanır. **main()** fonksiyonu **dist**'i tanımlar, kullanıcıdan bir uzaklık değeri almak için **Distance**'ın üye fonksiyonu olan **getdist()**'i çağırır, sonra bu değeri ekranda göstermek için **showdist()** fonksiyonunu kullanır.

Üyelere Erişmek

ENGLPTR daha sonra **new** operatörünü kullanarak **Distance** tipinde bir başka nesne daha tanımlar ve buna işaret eden **distptr** adında bir işaretçi döndürür.

Sorun şudur: **distptr** tarafından işaret edilen bir nesnenin üye fonksiyonlarını nasıl ele alacağız? Aşağıda gösterildiği gibi nokta (.) üye erişim operatörünü kullanacağımızı tahmin ediyor olabilirsiniz:

```
distptr.getdist();       //calismaz; distptr bir degisken degildir
```

Ancak, bu işe yaramaz. Nokta operatörü, sol tarafındaki tanımlayıcının bir değişken olmasını gerektirir. **disptr**, bir değişkeni işaret eden bir işaretçi olduğu için bir başka söz dizimine ihtiyacımız vardır. Bir yaklaşım, işaretçiyi dereferans etmektir (işaretçinin işaret ettiği değişkenin içeriğini almaktır).

```
(*distptr).getdist();   //tamam, fakat kaba saba
```

Yine de, bu, parantezlerden dolayı bir parça külfetli bir iştir. (Parantezler gereklidir, çünkü nokta operatörü (.), dereferans operatöründen (*) daha yüksek önceliğe sahiptir.) Eşdeğer ama daha kısa ve öz bir yaklaşım ise, bir tire ve büyüktür işaretinden oluşan bir üye erişim operatörü ile gerçekleştirilir:

```
distptr->getdist();      //daha iyi bir yaklasim
```

ENGLPTR'da görebileceğiniz gibi, . operatörü nesneler üzerinde nasıl çalışıyorsa, -> operatörü de nesnelere işaret eden işaretçiler üzerinde tamamen aynı şekilde çalışır. Programın çıktısı şöyledir:

```
Enter feet: 10          <===== bu nesne nokta operatörünü kullanır
Enter inches: 6.25
10'-6.25"

Enter feet: 6           <===== bu nesne -> operatörünü kullanır
Enter inches: 4.75
6'-4.75"
```

new'a Yeni Bir Bakış

Nesnelere bellek tahsis etmek amacıyla bir başka – daha az yaygın – `new` kullanımı ile karşılaşabilirsiniz.

`new`, bir nesnenin saklandığı bellek alanına işaret eden bir işaretçi döndürdüğü için işaretçinin içeriğine ulaşarak orijinal nesneye erişmemiz mümkün olabilmelidir. `ENGLREF` örneği bunun nasıl yapılabileceğini gösteriyor.

```
// englref.cpp
// new operatorunun dondurdugu isaretcinin icerigine ulasmak
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance       //Ingiliz olcu sistemine dayanan Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      void getdist()                   //uzakligi kullanicidan al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }
      void showdist()                  //uzakligi goster
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance& dist = *(new Distance);  //"dist " takma isimli
                                      //Distance nesnesi tanimla
   dist.getdist();                    //nokta operatoru ile
   dist.showdist();                   //nesnenin uyelerine eris
   cout << endl;
   return 0;
   }
```

Aşağıdaki deyim bir `Distance` nesnesi için yeteri kadar büyük bir bellek alanına işaret eden bir işaretçi döndürür:

```
new Distance
```

Böylelikle, orijinal nesneye şu şekilde erişebiliriz:

```
*(new Distance)
```

Yukarıdaki, işaretçi tarafından işaret edilen nesnedir. `dist`, bir referans kullanarak `Distance` tipinde bir nesne olarak tanımlanır ve `*(new Distance)` değerine eşitlenir. Artık, `->` yerine nokta operatörünü kullanarak `dist`'in üyelerine erişebiliriz.

Bu yöntem, `new` ile elde edilen nesnelere işaret eden işaretçiler kullanmaktan veya sadece bir nesne tanımlamaktan daha az yaygındır, fakat aynı şekilde çalışır.

Nesnelere İşaret Eden İşaretçiler Dizisi

Sıkça kullanılan bir programlama yapısı, nesnelere işaret eden işaretçilerin oluşturduğu dizidir. Bu düzenleme, bir grup nesneye kolay erişim imkanı verir; ayrıca, nesnelerin kendilerini bir diziye yerleştirmekten daha esnektir. (Söz gelişi, bu bölümdeki `PERSORT` örneğinde bir grup nesnenin, nesnelerin kendilerini sıralamak yerine, bu nesnelere işaret eden işaretçi dizisinin sıralanması ile sıralanabileceğini öğreneceğiz.)

Şimdiki örneğimiz olan `PTROBJS`, `person` sınıfına işaret eden işaretçilerden oluşan bir dizi tanımlar. Programın listesi şöyledir:

```
// ptrobjs.cpp
// nesnelere isaret eden isaretciler dizisi
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person                          //kisi sinifi
   {
   protected:
      char name[40];                  //kisinin ismi
   public:
      void setName()                  //isim ver
         {
         cout << "Enter name:";
         cin >> name;
         }
      void printName()                //ismi al
         {
         cout << "\nName is: " << name;
         }
   };
////////////////////////////////////////////////////////////////
int main()
  {
   person* persPtr[100];         //kisilere isaret eden isaretciler dizisi
   int n = 0;                    //dizi icindeki kisi sayisi
   char choice;

   do                            //kisileri diziye yerlestir
      {
      persPtr[n] = new person;   //yeni bir nesne tanimla
      persPtr[n]->setName();     //kisinin ismini gir
      n++;                       //yeni kisiyi say
      cout << "Enter another (y/n)? ";     //bir baska kisi
      cin >> choice;                       //girmek istiyor musunuz?
      }
   while(choice=='y' );          // 'n' ile cik
   for(int j=0; j<n; j++)        //kisilerin tumunun
      {                          //ismini yazdir
      cout << "\nPerson number " << j+1;
      persPtr[j]->printName();
      }
   cout << endl;
   return 0;
   }     //main()'in sonu
```

`person` sınıfı bir tek veri öğesine sahiptir: `name`. `name`, kişinin ismini simgeleyen bir karakter katarı tutar. `setName()` ve `printName()` adlı iki üye fonksiyon, ismin düzenlenmesine ve ekranda gösterilmesine imkan verir.

Programın İşleyişi

`main()` fonksiyonu, `person` tipine işaret eden 100 işaretçiden oluşan bir `persPtr` dizisi tanımlar. Program daha sonra bir `do` döngüsü içinde kullanıcıdan bir isim girmesini ister. Program, bu isimle birlikte `new` operatörünü kullanarak bir `person` nesnesi oluşturur ve bu nesneyi işaret eden bir işaretçiyi de `persPtr` dizisinde saklar. İşaretçi kullanarak nesnelere erişimin ne kadar kolay olduğunu göstermek için program daha sonra `person` nesnelerinin her biri için `name` verisinin dökümünü verir.

Programla örnek bir etkileşim şu şekilde gerçekleşebilir:

```
Enter name:Stroustrup          <==== kullanıcı ismi giriyor
Enter another (y/n)? y
Enter name: Ritchie
Enter another (y/n)? y
Enter name: Kernighan
Enter another (y/n)? n
Person number 1                <==== program, isimlerin tümünü ekranda gösterir
   Name is:Stroustrup
Person number 2
   Name is:Ritchie
Person number 3
   Name is:Kernighan
```

Üye Fonksiyonlara Erişim

`persPtr` dizisindeki işaretçilerin işaret ettiği `person` nesnelerindeki `setName()` ve `printName()` üye fonksiyonlarına erişmemiz gerekir. `persPtr` dizisinin elemanlarının her biri, dizi notasyonunda `persPtr[j]` şeklinde (veya işaretçi notasyonunda `*(persPtr+j)` şeklinde) belirtiliyor. Dizi elemanları `person` tipindeki nesnelere işaret eden işaretçilerden oluşur. Bir nesnenin bir üyesine işaretçi kullanarak erişmek için `->` operatörü kullanılır. Bütün bu anlattıklarımızı bir araya getirirsek `getname()` için aşağıdaki söz dizimi elde edilir:

```
persPtr[j]->getName()
```

Bu ifade, `persPtr` dizisinin `j`. elemanı tarafından işaret edilen `person` nesnesinin `getname()` fonksiyonunu çalıştırır. (İngilizce söz dizimini kullanarak programlama yapmak zorunda olmamamız iyi bir şeydir.)

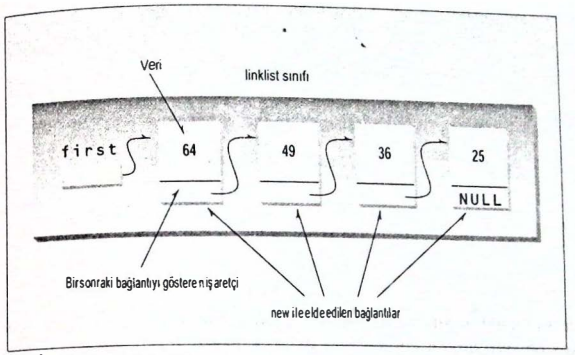
Bir Bağlı Liste Örneği

Bir sonraki örneğimizde basit bir bağlı liste gösteriliyor. Bağlı liste, verileri saklamanın bir başka yoludur. Verilerin dizilerde saklamasına ilişkin sayısız örnek gördünüz. Bir başka veri yapısı ise, veri üyelerine işaret eden işaretçilerden oluşan bir dizidir; bunun örneklerini `PTRTOSTRS` ve `PTROBJS` programlarında gördük. Hem diziler hem de işaretçi dizileri, programı çalıştırmadan önce sabit boyutta bir dizi tanımlamak zorunluluğunun sıkıntısını çekerler.

İşaretçiler Zinciri

Bağlı listeler daha esnek saklama sistemi sağlarlar, çünkü hiç dizi kullanmazlar. Bunun yerine, her veri öğesi için gerekli oldukça, `new` kullanılarak bellek alanı elde edilir ve her veri öğesi bir sonrakine bir işaretçi kullanılarak birleştirilir veya *bağlanıyor*. Öğelerin her biri, dizi elemanları gibi, bellekte peş peşe yerleştirilmek zorunda değildir; bellekte her yere dağıtılmış olabilirler.

Örneğimizde, bağlı listenin tümü `linklist` sınıfının bir nesnesidir. Veri öğelerinin her biri veya bağlantıları `link` tipinde yapılarla simgelenir. Bu tür yapıların her biri, bir tamsayıdır – bu tamsayı, nesnenin tek veri öğesini simgeler – ve bir sonraki bağlantıya bir işaretçi içerir. Listenin kendisi de, listenin en başındaki bağlantıya işaret eden bir işaretçi saklar. Bu düzenleme Şekil 10.15'te gösteriliyor.



ŞEKİL 10.15: Bağlı liste.

`LINKLIST`'in program listesi şöyledir:

```
// linklist.cpp
// bagli liste
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct link                        //listenin bir elemani
   {
   int data;                       //veri ogesi
   link* next;                     //sonraki baglantiya isaret eden isaretci
   };
////////////////////////////////////////////////////////////////
class linklist                     //baglantilarin listesi
   {
   private:
      link* first;                 //ilk baglantiya isaret eden isaretci
   public:
      linklist()                   //argumansiz kurucu fonksiyon
         { first = NULL; }         //ilk baglanti yok
      void additem(int d);         //veri ogesini ekle (bir tane baglanti)
      void display();              //tum baglantilari goster
   };
```

```
//-------------------------------------------------------------
void linklist::additem(int d)     //veri ogesini ekle
   {
   link* newlink = new link;      //yeni bir baglanti tanimla
   newlink->data = d;             //bunun verisine degerini ver
   newlink->next = first;         //bu, bir sonraki baglantiyi gosterir
   first = newlink;               //artik ilk baglanti bunu gosteriyor
   }
//-------------------------------------------------------------
void linklist::display()          //tum baglantilari goster
   {
   link* current = first;         //isaretci ilk baglantiyi gostersin
   while( current != NULL )       //son baglantiya gelince cik
      {
      cout << current->data << endl;   //veriyi yazdir
      current = current->next;         //sonraki baglantiya gec
      }
   }
////////////////////////////////////////////////////////////////
int main()
   {
   linklist li;                   //bagli liste olustur

   li.additem(25);                //listeye dort oge ekle
   li.additem(36);
   li.additem(49);
   li.additem(64);
   li.display();                  //listenin tumunu goster
   return 0;
   }
```

linklist sınıfının sadece bir tek veri öğesi vardır:Bu, listenin başına işaret eden bir işaretçidir. Liste ilk kez oluşturulurken kurucu fonksiyon, **first** olarak adlandırılan bu işaretçiye **NULL** değerini veriyor. **NULL** sabiti 0 olarak tanımlanır. Bir işaretçi geçerli bir adres tutmuyorsa bu değer bir uyarı görevi görür. Programımızda, bir sonraki üyesinin değeri **NULL** olan bir bağlantının listenin sonunda yer aldığı farz edilebilir.

Listeye Bir Öğe Eklemek

additem() üye fonksiyonu bağlı listeye bir öğe ekler. Yeni bağlantı, listenin başına ekleniyor. (**additem()** fonksiyonunu, öğeleri listenin sonuna ekleyecek şekilde de yazabilirdik, fakat bunu programlamak biraz daha karmaşık olur.) Şimdi, yeni bir bağlantı eklemek için gerekli adımlara bir göz atalım.

Önce, **link** tipinde yeni bir yapı tanımlanır:

```
link* newlink = new link;
```

Bu ifade **new** operatörü sayesinde, yeni **link** yapısı için bellek alanı oluşturur ve bu alana işaret eden işaretçiyi **newlink** değişkeninde saklar.

Sonra, yeni tanımlanan yapının üyelerine uygun değerler vermek istiyoruz. Yapı, bir özelliğinden dolayı sınıfa benzer. Bir yapıya ismi ile değil de bir işaretçi ile ulaşıyorsanız, yapının üyelerine **->** üye erişim operatörünü kullanarak erişebilirsiniz. Aşağıdaki iki satırda yer alan ifadeler, **additem()**'a argüman olarak aktarılan değeri **data** değişkenine atarlar ve **first**'in

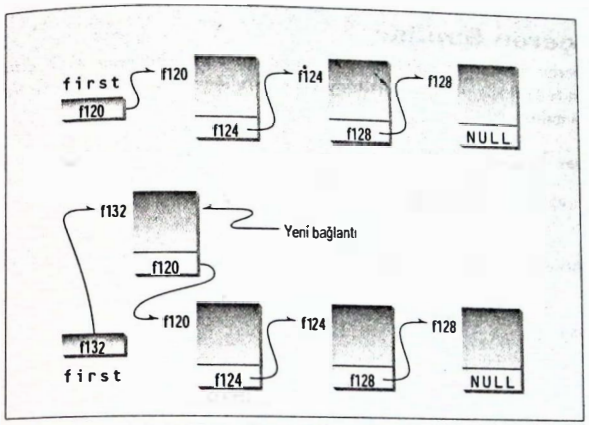
içindeki adrese **next** işaretçisinin işaret etmesini sağlarlar. **first**, listenin başına işaret eden işaretçiyi tutar.

```
newlink->data = d;
newlink->next = first;
```

Son olarak, **first** değişkeninin yeni bağlantıya işaret etmesini istiyoruz:

```
first = newlink;
```

Bu ifadenin etkisi, **first** ile eski **first** bağlantısı arasındaki bağlantıyı çözer, yeni bağlantıyı ekler ve eski **first** bağlantısını ikinci pozisyona kaydırır. Bu işlemler Şekil 10.16'da gösterilmiştir.



ŞEKİL 10.16: Bir bağlı listeye öğe eklemek.

Listenin İçeriğini Görüntülemek

Liste bir kez oluşturulduktan sonra, listenin üyelerinin tümü üzerinden adım adım ilerlemek ve bu üyeleri ekranda göstermek (veya diğer işlemleri gerçekleştirmek) çok kolaydır. Yapmamız gereken tek şey, bir **next** işaretçisinden diğerine geçmektir. Listenin sonunu gösteren **NULL**'a eşit olan bir **next** işaretçisi ile karşılaşana kadar bu işleme devam edilir. **display()** fonksiyonunda şu ifade, verinin değerini ekrana yazar:

```
cout <<endl <<current->data;
```

Şu ifade ise bir bağlantıdan diğerine ilerlememizi sağlar:

```
current=current->next;
```

ta ki **while** ifadesinde yer alan aşağıdaki deyim yanlış olana kadar:

```
current !=NULL
```

LINKLIST'in çıktısı işte şöyledir:

```
64
49
36
25
```

Bağlı listeler, dizilerden sonra belki de en yaygın olarak kullanılan veri saklama düzenekleridir. Önceden belirttiğimiz gibi, bağlı listeler, dizilerde ortaya çıkan bellek alanının israfını önlerler. Fakat, bağlı listelerin de dezavantajı vardır: Listedeki belirli bir öğeyi aramak, listenin başından başlayarak istenilen bağlantıya ulaşana kadar bağlantılar zincirini takip etmeyi gerektirir. Bu, zaman kaybına neden olabilir. Öte yandan, indeksi önceden bilinmek şartıyla bir dizi elemanına çabucak erişilebilir. Bölüm 15'te "Standart Şablon Kütüphanesi" bahsinde bağlı listeler ve veri saklama teknikleri hakkında daha fazla konuşacağız.

Kendisini İçeren Sınıflar

Kendi kendini gösteren sınıflar ve yapılarla ilgili muhtemel bir tehlikenin altını çizmeliyiz. `LINKLIST` programında `link` yapısı, aynı tipte bir yapıya işaret eden bir işaretçi içeriyordu. Aynı şeyi sınıflarla da yapabilirsiniz:

```
class sampleclass
   {
   sampleclass* ptr;    //bu tamam
   };
```

Bir sınıf kendi tipinde bir nesneye işaret eden bir işaretçi içerebilirken, kendi tipinde bir *nesne* içeremez:

```
class sampleclass
   {
   sampleclass obj;     //bunu yapamazsiniz
   };
```

Bu, sınıflar için geçerli olduğu gibi yapılar için de geçerlidir.

LINKLIST'i Değerlendirelim

`LINKLIST`'in genel organizasyonu burada gösterilenden daha karmaşık durumları da ele alabilir. Her bağlantıda daha fazla veri bulunabilir. Bir bağlantı, bir tamsayı tutmak yerine birkaç tane veri öğesi tutabilir; ya da bir yapıya veya bir nesneye işaret eden bir işaretçi bulundurabilir.

İlave üye fonksiyonlar, zincirin herhangi bir yerine bağlantı ekleme veya çıkarma gibi etkinlikleri gerçekleştirebilirler. Bir başka önemli üye fonksiyon ise yok edici fonksiyondur. Önceden bahsettiğimiz gibi, kullanımda olmayan bellek bloklarını elden çıkarmak önemlidir. Bu işi gerçekleştirecek bir yok edici fonksiyon, `linklist` sınıfı için oldukça hoş bir ilave olacaktır. Yok edici fonksiyon, bağlantıların her birinin işgal ettiği belleği serbest bırakmak amacıyla `delete`'i kullanarak liste boyunca ilerleyebilir.

İşaretçilere İşaret Eden İşaretçiler

Sıradaki örneğimiz, nesnelere işaret eden işaretçilerin oluşturduğu bir diziyi gösterir ve nesnelerin içindeki verileri baz alarak bu işaretçilerin nasıl sıralanacağını açıklar. Bu fikir,

işaretçilere işaret eden işaretçileri içerir. Üstelik, insanların işaretçilerden dolayı neden uykusuz kaldıklarını da açıklamaya yardımcı olabilir.

Sıradaki programın dayandığı fikir, `person` sınıfının nesnelerine işaret eden işaretçilerden oluşan bir dizi tanımlamaktır. Bu, `PTROBJS` programına benzer, fakat bu programda biraz daha ilerliyoruz ve `PTRSORT` örneğindeki `order()` ve `bsort()` fonksiyonlarının varyasyonlarını bu programa ekliyoruz. Böylece, bir grup `person` nesnesini, isimlerinin alfabetik sırasına göre sıralayabiliriz. `PERSORT` programının listesi işte şöyledir:

```
// persort.cpp
// isaretci dizisi kullanarak person nesnelerini sirala
#include <iostream>
#include <string>
using namespace std;                    //string sinifi icin
////////////////////////////////////////////////////////////
class person                            //kisi sinifi
   {
   protected:
      string name;                      //kisinin ismi
   public:
      void setName()                    //isim ver
         { cout << "Enter name: "; cin >> name; }
      void printName()                  //ismi goster
         { cout << endl << name; }
      string getName()                  //ismi dondur
         { return name; }
   };
////////////////////////////////////////////////////////////
int main()
   {
   void bsort(person**, int);           //prototip
   person* persPtr[100];                //kisileri isaret eden isaretciler dizisi
   int n = 0;                           //dizi icindeki kisilerin sayisi
   char choice;                         //girdi olarak kullanilacak karakter

   do {                                 //kisileri diziye yerlestir
      persPtr[n] = new person;          //yeni bir nesne tanimla
      persPtr[n]->setName();            //kisinin ismini gir
      n++;                              //yeni kisiyi say
      cout << "Enter another (y/n)? "; //baska bir kisi
      cin >> choice;                    //girmek istiyor musunuz?
      }
   while( choice=='y' );                //'n' ile cik

   cout << "\nUnsorted list:";
   for(int j=0; j<n; j++)               //siralanmamis listeyi yazdir
      { persPtr[j]->printName(); }

   bsort(persPtr,n);                    //isaretcileri sirala
   cout << "\nSorted list:";
   for(j=0; j<n; j++)                   //sirali listeyi yazdir
      { persPtr[j]->printName(); }
   cout << endl;
   return 0;
   }     //main()'in sonu
//--------------------------------------------------------------
void bsort(person** pp, int n)          //kisileri isaret eden isaretcileri sirala
   {
```

```
    void order(person**, person**);     //prototip
    int j, k;                           //dizi indeksleri

    for(j=0; j<n-1; j++)            //distaki dongu
       for(k=j+1; k<n; k++)         //icteki dongu distakinin kaldigi yerden baslar
          order(pp+j, pp+k);        //isaretcilerin icerigini sirala
    }
//-------------------------------------------------------------
void order(person** pp1, person** pp2)      //iki isaretciyi sirala
   {                                        //eger 1. 2.'den buyukse,
   if( (*pp1)->getName() > (*pp2)->getName() )
      {
      person* tempptr = *pp1;               //isaretcileri degistir
      *pp1 = *pp2;
      *pp2 = tempptr;
      }
   }
```

Program çalışmaya başladığında bir isim sorar. Kullanıcı bir isim girdiğinde, program **person** tipinde bir nesne oluşturur ve bu nesnenin **name** verisine kullanıcı tarafından girilen ismi yerleştirir. Program ayrıca, bu nesneye işaret eden bir işaretçiyi **persPtr** dizisinde saklar.

Kullanıcı daha fazla isim girilmeyeceği bildiren **n** harfini tuşladığında program, **person** nesnelerini, **name** üye değişkenlerini baz alarak sıralamak için **bsort()** fonksiyonunu çağırır. Programla gerçekleştirilen örnek bir etkileşim şöyledir:

```
Enter name: Washington
Enter another (y/n)? y
Enter name: Adams
Enter another (y/n)? y
Enter name: Jefferson
Enter another (y/n)? y
Enter name: Madison
Enter another (y/n)? n
Unsorted list:
Washington
Adams
Jefferson
Madison

Sorted list:
Adams
Jefferson
Madison
Washington
```

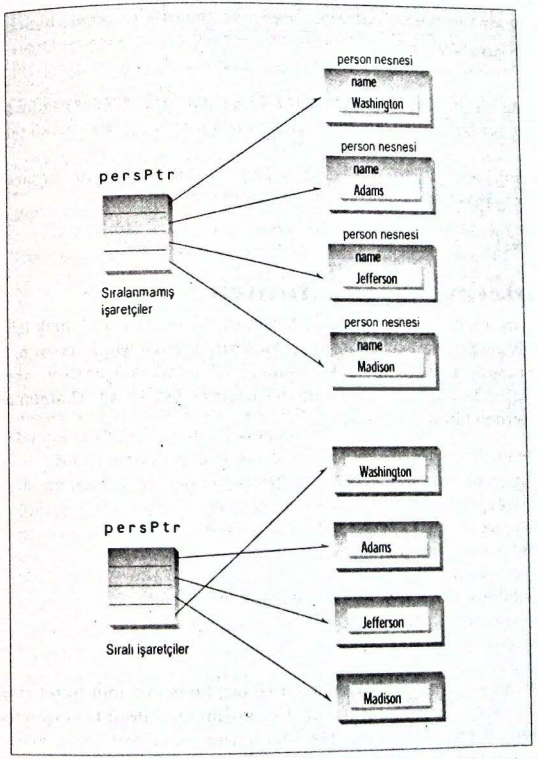
İşaretçileri Sıralamak

Aslında, **person** nesnelerini sıraladığımız zaman nesnelerin kendileri değil, nesnelere işaret eden işaretçiler taşınır. Bu bizi, nesneleri bellekte oradan oraya taşımak zorunluluğundan kurtarır. Nesneler büyük olduğunda bu işlem çok zaman alıcı olabilir. Bu yaklaşım ayrıca, birkaç sıralama işlemini – mesela, önce isme sonra telefon numarasına göre – nesneleri birkaç kez yer değiştirmeden bellekte aynı anda yapma imkanı verir. Şekil 10.17'de bu işlem gösteriliyor.

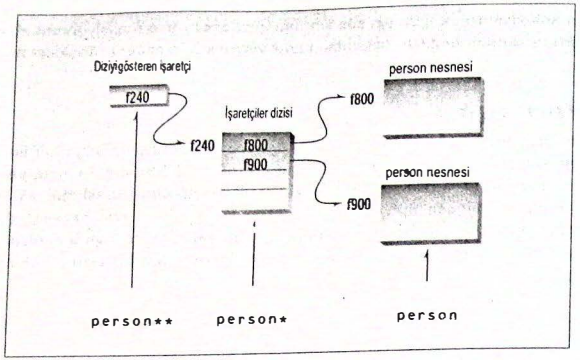
Sıralama işlemini kolaylaştırmak için, **person** sınıfına **getName()** üye fonksiyonunu ekledik. Bu sayede, işaretçileri ne zaman değiştireceğimize karar vermek için **order()** fonksiyonundan isimlere erişebiliriz.

person** Veri Tipi

Dikkat ederseniz, **bsort()** fonksiyonunun ilk argümanı ve **order()** fonksiyonunun her iki argümanı **person**** tipine sahipler. İki tane asteriks (**) ne anlama gelir? Bu argümanlar, **persPtr** dizisinin adresini veya **order()** fonksiyonunda, dizinin elemanlarının adreslerini aktarmak amacıyla kullanılır. Eğer bu **person** tipinde bir dizi olsaydı, dizinin adresi **person*** tipinde olurdu. Ancak, dizi, **person**'a işaret eden *işaretçi* tipinde, yani **person***, olduğu için dizinin adresi **person**** tipinde olur. Bir işaretçinin adresi işaretçiye işaret eden bir işaretçi olur. Şekil 10.18'de bu açıklanmıştır.



ŞEKİL 10.17: İşaretçi dizisini sıralamak.



ŞEKİL 10.18: İşaretçi dizisine işaret eden işaretçi.

Bu programı, `int` tipinde bir diziyi sıralayan `PTRSORT` ile karşılaştırın. `PERSORT`'ta fonksiyonlara aktarılan veri tiplerinin her biri, `PTRSORT`'takilerden bir fazla asteriks içerir, çünkü buradaki dizi bir işaretçi dizisidir.

`persPtr` dizisi işaretçileri içerdiği için, aşağıdaki ifade, `persPtr`'ın `j`. elemanı tarafından işaret edilen nesnenin `printName()` isimli fonksiyonunu çalıştıracaktır:

```
persPtr [j]->printName()
```

Karakter Katarlarını Karşılaştırmak

`PERSORT`'taki `order()` fonksiyonu iki karakter katarını leksikografik açıdan sıralamak için, yani alfabetik sıraya koymak için değiştirilmiştir. Bunu gerçekleştirmek amacıyla fonksiyon, karakter katarlarını C++'ın `strcmp()` kütüphane fonksiyonunu kullanarak karşılaştırır. `strcmp()` fonksiyonu, `strcmp(s1, s2)`ifadesindeki gibi, iki karakter katarını, (`s1` ve `s2`'yi) argüman olarak alır ve aşağıdaki değerlerden birini döndürür:

Değer	Koşul
<0	`s1`, `s2`'den önce gelir
0	`s1`, `s2` ile aynı
>0	`s1`, `s2`'den sonra gelir

Karakter katarlarına aşağıda verilen söz dizimi kullanılarak erişilir:

```
(*pp1)->getname()
```

`pp1` argümanı, bir işaretçiye işaret eden bir işaretçidir ve biz, bu işaretçinin işaret ettiği işaretçi tarafından işaret edilen nesnenin ismini istiyoruz. Üye erişim operatörü (`->`) işaretçi üzerinde bir seviye ilerlememizi (dereferans) sağlar, fakat bir seviye daha ilerlemeye ihtiyacımız vardır. Bu nedenle, `pp1`'den önce bir asteriks kullanılır.

Tıpkı işaretçilere işaret eden işaretçiler olabildiği gibi, işaretçilere işaret eden işaretçilere işaret eden işaretçiler de olabilir, vs. Neyse ki bu tür karmaşıklarla çok ender karşılaşılır.

Bir "Parsing" Örneği

Programcılar, simgelerden oluşan bir karakter katarını çözmek veya "parse" etmek problemiyle sık sık karşılaşırlar. Kullanıcının klavyeden tuşladığı komutlar, doğal dillerdeki (mesela, İngilizce) ifadeler, bir programlama dilindeki ifadeler veya cebirsel deyimler bu tür karakter katarı örnekleri arasında yer alır. Artık işaretçileri ve karakter katarlarını öğrendiğimize göre, bu tür problemleri ele alabiliriz. Sıradaki (biraz da uzun) örneğimiz aşağıdaki gibi aritmetik deyimlerin nasıl "parse" edildiğini gösterecek:

```
6/3+2*3-1
```

Kullanıcı aritmetik deyimi girer. Program, deyim üzerinde karakter karakter ilerleyerek kendi çabasıyla aritmetiksel olarak anlamını çıkarır ve elde edilen sonucu (bu örnekte, 7) ekranda gösterir. Deyimlerimiz dört aritmetik operatör kullanacaktır: `+`, `-`, `*` ve `/`. Programlamayı kolaylaştırmak için kullandığımız sayıları tek basamakla sınırlandıracağız. Ayrıca, parantezlere de izin vermeyeceğiz.

Bu program, eski dostumuz `Stack` sınıfından faydalanır (Bölüm 7'deki `STAKARAY` programına bakın). Bu sınıfı, `char` tipinde verileri saklayacak biçimde değiştirdik. Yığını hem sayıları, hem de operatörleri (her ikisini karakter olarak) saklamak için kullanıyoruz. Yığın, kullanışlı bir saklama mekanizmasıdır, çünkü deyimleri "parse" ederken sık sık en son saklanan öğeye erişmemiz gerekir; yığın ise son giren ilk çıkar (LIFO) mantığına dayanan bir konteynerdir.

`Stack` sınıfının yanı sıra, aritmetik deyimi bütünüyle simgeleyen `express` (expression'ın kısaltması olarak) adında bir sınıf daha kullanacağız. Bu sınıfın üye fonksiyonları, bir nesneye karakter katarı formunda bir deyimi, değer olarak atamaya ve deyimi "parse" etmeye imkan verir. Ayrıca, sonuçta elde edilen aritmetik değeri döndürmeyi mümkün kılar.

Aritmetik Deyimleri "Parse" Etmek

Bir aritmetik deyimi şu şekilde "parse" ederiz: Soldan başlarız ve sırayla karakterlerin her birini inceleriz. Karakterler bir *sayı* (her zaman tek bir rakam – 0 ve 9 arasındaki bir karakter) veya bir *operatör* (`+`, `-`, `*` ve `/` karakterleri) olabilir.

Eğer karakter bir sayı ise, bunu her zaman yığının üstüne yerleştiririz. Ayrıca, rastladığımız ilk operatörü de yığına yerleştiririz. Asıl hüner sonraki operatörleri ele alış şeklimizde yatar. Dikkat ederseniz, şu andaki operatörün işlemini gerçekleştiremeyiz, çünkü bu operatörün peşinden gelen sayıyı henüz okumadık. Bir operatör ile karşılaşmak yalnızca, yığında saklı tutulan bir önceki operatörün işleminin yapılabileceğine işaret eder. Yani, eğer `2+3` karakter serisi yığında ise, toplama işlemini gerçekleştirmeden önce bir başka operatör ile karşılaşmayı bekleriz.

Böylelikle, şu anki karakterin bir operatör olduğunu fark ettiğimiz her seferde (ilk operatör hariç), önceki sayıyı (önceki örnekteki 3) ve önceki operatörü (`+`) yığından alırız ve bunları sırasıyla `lastval` ve `lastop` değişkenlerine yerleştiririz. En son olarak, ilk sayıyı (2) yığından alırız ve iki sayı üzerinde aritmetik işlemi gerçekleştiririz (elde edilen 5 olur). Bir önceki operatörün işlemini her zaman gerçekleştirebilir miyiz? Hayır. Hatırlarsanız, `*` ve `/` operatörleri `+` ve `–` operatörlerinden daha yüksek önceliğe sahiptir. `3+4/2` deyiminde toplamayı, bölme işlemini tamamlayana kadar gerçekleştiremeyiz. Bu nedenle, bu deyimde `/` operatörüne rastla-

yınca 2 ve + operatörünü yığına geri koymamız gerekir. Bölme işlemini tamamlayana kadar 2 ve + yığında kalmalıdır.

Öte yandan, eğer şu anki operatör + veya – ise, önceki operatörün işlemini her zaman gerçekleştirebileceğimizi biliyoruz. Yani, `4-5+6` deyiminde + operatörünü görünce, – operatörünün işleminin gerçekleştirilmesinde bir problem olmadığını biliyoruz. `6/2-3` deyiminde ise bölmeyi yapmanın uygun olduğunu biliyoruz. Tablo 10.1'de dört muhtemel durum gösteriliyor.

TABLO 10.1: Operatörler ve "Parse" İşlemleri

Önceki Operatör	Şu Andaki Operatör	Örnek	İşlem
+ veya -	* veya /	3+4/	Önceki operatörü ve önceki sayıyı yığına ekle (+, 4)
* veya /	* veya /	9/3*	Önceki operatörün işlemini yap, sonucu yığına ekle (3)
+ veya -	+ veya -	6+3+	Önceki operatörün işlemini yap, sonucu yığına ekle (9)
* veya /	+ veya -	8/2-	Önceki operatörün işlemini yap, sonucu yığına ekle (4)

`parse()` üye fonksiyonu, programın girdisi olan deyim üzerinden ilerleyerek ve yapabileceği aritmetik hesaplamaları gerçekleştirerek bu işlemi sürdürür. Yine de, işin tamamı bu kadarla sınırlı değildir. Yığın halen ya tek bir sayı ya da sayı-operatör-sayı serisinden birkaç tane içerir. Yığın üzerinde aşağıya doğru çalışarak işlemleri gerçekleştirebiliriz. Son olarak, yığında tek bir sayı kalır; bu sayı orijinal deyimin değeridir. Bütün bu işlemleri, yığında tek bir sayı kalana kadar yığın üzerinde ilerleyerek `solve()` üye fonksiyonu yürütür. Genellikle, `parse()` yığına ekleme işini, `solve()` ise yığından çıkarma işini yürütür.

PARSE Programı

PARSE programı ile tipik bir etkileşim şu şekilde görünebilir:

```
Enter an arithmetic expression
of the form 2+3*4/3-2.
No number may have more than one digit.
Don't use any spaces or parentheses.
Expression: 9+6/3

The numerical value is: 11
Do another (Enter y or n)?
```

Dikkat ederseniz, aritmetik işlemlerin *sonuçlarının* birden fazla basamak içermesinde bir problem yok. Sonuçlar yalnızca `char` tipinin nümerik büyüklüğü ile sınırlıdır (-128 ile +127 arası). Sadece girdi karakter katarındaki sayılar 0 ile 9 arasında olmak zorundadır.

Programın listesi işte şöyledir:

```
// parse.cpp
// 1 basamakli sayilardan olusan aritmetik deyimleri hesaplar
#include <iostream>
#include <cstring>
using namespace std;                    //strlen() vs icin
```



```
const int LEN = 80;                     //deyimlerin uzunlugu, karakter cinsinden
const int MAX = 40;                     //yiginin buyuklugu
////////////////////////////////////////////////////////////
class Stack
   {
   private:
      char st[MAX];                     //yigin:char dizisi
      int top;                          //yiginin en ustunu gosteren sayi (indeks)
   public:
      Stack()                           //kurucu fonksiyon
         { top = 0; }
      void push(char var)               //karakteri yigina koy
         { st[++top] = var; }
      char pop()                        //karakteri yigindan al
         { return st[top--]; }
      int gettop()                      //yiginin en ustunu al
         { return top; }
   };
////////////////////////////////////////////////////////////
class express                           //deyim sinifi
   {
   private:
      Stack s;                          //analizde kullanilacak yigin
      char* pStr;                       //girdi katarina isaret eden isaretci
      int len;                          //girdi katarinin uzunlugu
   public:
      express(char* ptr)                //kurucu fonksiyon
         {
         pStr = ptr;                    //isaretciyi katari gosterecek sekilde ayarla
         len = strlen(pStr);            //uzunlugu ayarla
         }
      void parse();                     //girdi karakter katarini "parse" et
      int solve();                      //yigindakileri hesapla
   };
//--------------------------------------------------------------
void express::parse()                   //ogeleri yigina ekle
   {
   char ch;                             //girdide kullanilacak karakter
   char lastval;                        //son deger
   char lastop;                         //son operator

   for(int j=0; j<len; j++)             //her girdi karakteri icin
      {
      ch = pStr[j];                     //bir sonraki karakteri al

      if(ch>='0' && ch<='9' )           //bu bir rakamsa,
         s.push(ch-'0');                //numerik degeri sakla
                                        //eger operator ise,
      else if(ch=='+' || ch=='-' || ch=='*' || ch=='/')
         {
         if(s.gettop()==1)              //eger ilk operator ise
            s.push(ch);                 //yigina koy
         else                           //ilk operator degil
            {
            lastval = s.pop();          //onceki rakami al
            lastop = s.pop();           //onceki operatoru al
            //eger bu * veya / ise VE son operator + veya - ise
            if( (ch=='*' || ch=='/') &&
                  (lastop=='+' || lastop=='-') )
```

```
                {
                s.push(lastop);         //yigindaki son iki yerlestirmeyi geri al
                s.push(lastval);
                }
             else                       //diger tum durumlarda
                {
                switch(lastop)          //son islemi yap
                   {                    //sonucu yigina koy
                   case '+': s.push(s.pop() + lastval); break;
                   case '-': s.push(s.pop() - lastval); break;
                   case '*': s.push(s.pop() * lastval); break;
                   case '/': s.push(s.pop() / lastval); break;
                   default: cout << "\nUnknown oper"; exit(1);
                   }     //switch'in sonu
                }     //else'in sonu, diger tum durumlar
             s.push(ch);                //su andaki operatoru yigina koy
             }     //else'in sonu, ilk operator degil
          }     //else if'in sonu, bu bir operator degil
       else                             //bilinen bir karakter degil
          { cout << "\nUnknown input character"; exit(1); }
       }     //for'un sonu
    }     //parse()'in sonu
//--------------------------------------------------------------
int express::solve()                    //ogeleri yigindan cikar
   {
   char lastval;                        //onceki deger

   while(s.gettop()>1)
      {
      lastval = s.pop();                //onceki degeri al
      switch( s.pop() )                 //onceki operatoru al
         {                              //islemi yap, sonucu yigina koy
         case '+': s.push(s.pop() + lastval); break;
         case '-': s.push(s.pop() - lastval); break;
         case '*': s.push(s.pop() * lastval); break;
         case '/': s.push(s.pop() / lastval); break;
         default: cout << "\nUnknown operator"; exit(1);
         }     //switch'in sonu
      }     //while'in sonu
   return int( s.pop() );               //yigindaki son oge yanittir
   }     //solve()'un sonu
////////////////////////////////////////////////////////////
int main()
   {
   char ans;                            //'y' veya 'n'
   char string[LEN];                    //kullanicidan alinan girdi karakter katari

   cout << "\nEnter an arithmetic expression"
           "\nof the form 2+3*4/3-2."
           "\nNo number may have more than one digit."
           "\nDon't use any spaces or parentheses.";
   do {
      cout << "\nEnter expresssion: ";
      cin >> string;                                  //kullanicidan alinan girdi
      express* eptr = new express(string);            //deyimi tanimla
      eptr->parse();                                  //deyimi "parse" et
      cout << "\nThe numerical value is: "
           <<eptr->solve();                           //coz
      delete eptr;                                    //deyimi temizle
```



```
      cout << "\nDo another (Enter y or n)? ";
      cin >> ans;
      } while(ans == 'y');
   return 0;
   }
```

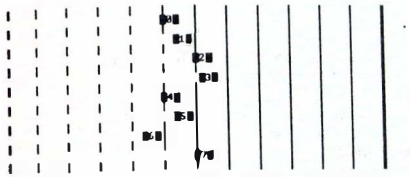
Bu uzunca bir programdır. Fakat, bu program, önceden tanımlanmış bir sınıfın (`Stack`) yeni bir durumda nasıl kullanışlı olabileceğini, çeşitli açılardan işaretçi kullanımını ve bir karakter katarını bir karakter dizisi olarak ele almanın ne kadar yararlı olabileceğini gösterir.

Simülasyon: Bir At Yarışı

Bu bölümdeki son örneğimiz olarak bir at yarışı oyununu göstereceğiz. Bu oyunda, birkaç tane at ekranda görünür ve sol taraftan başlayarak sağ tarafta bulunan bitiş çizgisine doğru yarışırlar. Bu program, işaretçilerin yeni bir kullanımını gösterecek ve nesne yönelimli tasarıma da çok az değinecektir.

Her atın hızı rasgele belirlenir; bu nedenle, hangi atın kazanacağını önceden bilmenin yolu yoktur. Program konsol grafikleri kullanır. Böylece atlar, gerçi biraz kabaca olsa da, kolaylıkla ekranda gösterilebilir. Bu programı, (derleyicinize bağlı olarak) `MSOFTCON.H` veya `BORLACON.H` başlık dosyası ile ve `MSOFTCON.CPP` veya `BORLACON.CPP` kaynak dosyası ile derlemeniz gerekir. (Daha fazla bilgi için "Console Graphic Lite" adlı Ek E'ye bakın.)

`HORSE` isimli programımız çalışmaya başladığında kullanıcının yarış mesafesini ve koşacak atların sayısını girmesini ister. At yarışlarının klasik uzaklık birimi (en azından ana dili İngilizce olan ülkelerde) bir milin 1/8'i olan (200 metrelik mesafe) *furlong*'dur. Yarışlar tipik olarak 6, 8, 10 veya 12 furlong'dur. 1 ila 7 atı yarışa sokabilirsiniz. Program, başlangıç ve bitiş çizgileri arasına her furlong'a karşılık bir düşey çizgi çizer. Atların her biri, ortasında bir sayı olan bir dikdörtgen ile simgelenir. Şekil 10.19 yarış sürerken ekranda atların konumlarını gösterir.



ŞEKİL 10.19: `HORSE` programının çıktısı.

At Yarışını Tasarlamak

At yarışımız için ne tür bir nesne yönelimli programlama tasarımı ele alacağız? İlk sorumuz belki de şu olabilir: Modellemeye çalıştıklarımız arasında benzer özelliklere sahip varlıklar var mı? Yanıt: Evet, atlar. Bu nedenle, her atı bir nesne yapmak mantıklı görünüyor. `horse` adında bir sınıf olacaktır. Bu sınıf, her atın kendisine özgü verilerini, mesela numarasını ve şimdiye dek koştuğu mesafeyi (bu bilgi, atın ekranda doğru konumda gösterilmesi için kullanılır) içerecektir.

Bunun yanı sıra, tek tek atlar üzerinde değil de, yarışın bütününde etkisi olan başka veriler de vardır. Yarış pistinin uzunluğu, geçen süre (dakika ve saniye cinsinden; yarışın başlangıcında bu değer 0:00'dır) ve toplam at sayısı bunların arasında yer alır. Bu nedenle, `track` sınıfı-

nın tek üyesi olan bir **track** nesnesine sahip olmak mantıklı görünür. At yarışıyla ilgili, gerçek dünyaya ait başka nesneler de aklınıza gelmiş olabilir, jokeyler ve eğerler gibi; fakat, bunların bu programla bir ilgisi yoktur.

Bu programı tasarlamanın başka yöntemleri de mevcut mu? Örneğin, kalıtım kullanarak atları yarış pistinden geliyormuş gibi tanımlamaya ne dersiniz? Bu pek anlamlı değil, çünkü atlar, yarış pistinin "bir türü" değil, onlar tamamen farklı bir varlık. Bir başka seçenek, pist verilerini **horse** sınıfının statik verileri haline getirmektir. Yine de, problem uzayındaki (gerçek dünyadaki) farklı türdeki her şeyi programda ayrı birer sınıf olarak tanımlamak genellikle daha iyi bir yaklaşımdır. Bunun bir avantajı, sınıfları başka bir bağlamda kullanmanın daha kolay olmasıdır. Örneğin yarış pistini atlar yerine arabalar için kullanabilirsiniz.

horse nesneleri ile **track** nesnesi nasıl haberleşecekler? (Veya UML terimleri ile ifade edersek, bunların bağlantıları (associations) nelerden oluşacak?) **horse** nesnelerine işaret eden işaretçilerin oluşturduğu bir dizi, **track** sınıfının bir üyesi olabilir; böylece, **track** (yarış pisti) bu işaretçiler yardımıyla atlara erişebilir. **track** tanımlandığında, atları da tanımlayacaktır. Böyle yaptığı takdirde, her at için kendisine bir işaretçi aktarmış olacak, böylece, söz konusu at, **track**'e erişebilecektir.

HORSE programının listesi şöyledir:

```
// horse.cpp
// bir at yarisini modeller
#include "msoftcon.h "              //konsol grafikleri icin
#include <iostream>                 //I/O icin
#include <cstdlib>                  //random() icin
#include <ctime>                    //time() icin
using namespace std;
const int CPF = 5;                  //her furlong'daki (200 mt) sutun sayisi
const int maxHorses = 7;            //atlarin maksimum sayisi
class track;                        //daha sonra kullanilmak uzere
////////////////////////////////////////////////////////////////
class horse
   {
   private:
      const track* ptrTrack;        //track'a isaret eden isaretci
      const int horse_number;       //bu atin numarasi
      float finish_time;            //bu atin yarisi bitirme suresi
      float distance_run;           //simdiye kadar kosulan mesafe
   public:                          //ati tanimla
      horse(const int n, const track* ptrT) :
               horse_number(n), ptrTrack(ptrT),
               distance_run(0.0)    //henuz hareket etmiyor
         { }
      ~horse()                      //ati yok et
         { /*bos*/ }                //ati goster
      void display_horse(const float elapsed_time);
   };    //class horse'un sonu
////////////////////////////////////////////////////////////////
class track
   {
   private:
      horse* hArray[maxHorses];     //atlara isaret eden isaretci dizisi
      int total_horses;             //atlari toplam sayisi
      int horse_count;              //simdiye dek tanimlanan atlar
      const float track_length;     //furlong cinsiden yaris pistinin uzunlugu
      float elapsed_time;           //yarisin basindan beri gecen sure
```



```
   public:
      track(float lenT, int nH);    //2-argumanli kurucu fonksiyon
      ~track();                     //yok edici fonksiyon
      void display_track();         //track'i goster
      void run();                   //yarisi baslat
      float get_track_len() const;  //toplam pist uzunlugunu dondur
   };    //class track'in sonu
//-------------------------------------------------------------
void horse::display_horse(float elapsed_time)     //her at icin
   {                                              //ati ve numarasini goster
   set_cursor_pos( 1 + int(distance_run * CPF),
                2 + horse_number*2);
                                                  //0 numarali at, mavi renkli
   set_color(static_cast<color>(cBLUE+horse_number));
                                                  //ati ciz
   char horse_char = '0' + static_cast<char>(horse_number);
   putch(' '); putch('\xDB'); putch(horse_char); putch('\xDB');
                                                  //bitise kadar,
   if( distance_run < ptrTrack->get_track_len() + 1.0 / CPF )
      {
      if( rand() % 3 )                            //3 tikin birini atla
         distance_run += 0.2F;                    // 0.2 furlong ilerle
      finish_time = elapsed_time;                 //bitis suresini guncelle
      }
   else
      {                                           //bitis suresini goster
      int mins = int(finish_time)/60;
      int secs = int(finish_time) - mins*60;
      cout << " Time=" << mins << ":" << secs;
      }
   }     //display_horse()'un sonu
//-------------------------------------------------------------
track::track(float lenT, int nH) :  //track kurucu fonksiyonu
                  track_length(lenT), total_horses(nH),
                  horse_count(0), elapsed_time(0.0)
   {
   init_graphics();                 //graphigi ilk kullanima hazirla
   total_horses =                   //7 attan fazla olmasin
      (total_horses > maxHorses) ? maxHorses : total_horses;
   for(int j=0; j<total_horses; j++)  //atlarin her birini tanimla
      hArray[j] = new horse(horse_count++, this);

   time_t aTime;                    //rasgele sayilari ilk kullanima hazirla
   srand( static_cast<unsigned>(time(&aTime) );
   display_track();
   }     //track kurucu fonksiyonunun sonu
//-------------------------------------------------------------
track::~track()                     //track yok edici fonksiyonu
   {
   for(int j=0; j<total_horses; j++)  //atlarin her birini yok et
      delete hArray[j];
   }
//-------------------------------------------------------------
void track::display_track()
   {
   clear_screen();                  //ekrani temizle
                                    //yaris pistini goster
   for(int f=0; f<=track_length; f++)      //her furlong icin
      for(int r=1; r<=total_horses*2 + 1; r++)    //ve ekran satiri icin
```

```
            {
            set_cursor_pos(f*CPF + 5, r);
            if(f==0 || f==track_length)
               cout << '\xDE';          //baslangic veya bitis cizgisini ciz
            else
               cout << '\xB3';          //furlong isaretini ciz
            }
      }     //display_track()'in sonu
//--------------------------------------------------------------
void track::run()
   {
   while( !kbhit() )
      {
      elapsed_time += 1.75;            //zamani guncelle
                                       //atlarin her birini guncelle
      for(int j=0; j<total_horses; j++)
         hArray[j]->display_horse(elapsed_time);
      wait(500);
      }
   getch();                            //basilan tusu al
   cout << endl;
   }
//--------------------------------------------------------------
float track::get_track_len() const
   { return track_length; }
////////////////////////////////////////////////////////////////
int main()
   {
   float length;
   int total;
                                       //kullanicidan verileri al
   cout << "\nEnter track length (furlongs; 1 to 12): ";
   cin >> length;
   cout << "\nEnter number of horses (1 to 7): ";
   cin >> total;
   track theTrack(length, total);   //yaris pistini tanimla
   theTrack.run();                  //yarisi baslat
   return 0;
   }     //main()'in sonu
```

Süreyi Tutmak

Simülasyon programları genellikle bir zaman aralığında meydana gelen bir etkinlik içerirler. Geçen süreyi modellemek için bu tür programlar tipik olarak sabit aralıklarla iş yaparlar. **HORSE** programında, **main()** programı **track**'ın **run()** fonksiyonunu çağırır. Bu fonksiyon, bir **while** döngüsü içinde, her at için bir kez olmak üzere **display_horse()** fonksiyonuna bir dizi çağrı yapar. Bu fonksiyon, atların her birini yeni konumunda yeniden çizer. Sonra **while** döngüsü, konsol grafik **wait()** fonksiyonu kullanılarak 500 milisaniye durur. Daha sonra, yarış bitene kadar ya da kullanıcı bir tuşa basana kadar aynı işlemler tekrarlanır.

Nesnelere İşaret Eden İşaretçi Dizisini Yok Etmek

Programın sonunda **track**'e ait bir yok edici fonksiyon, **horse** nesnelerini yok etmelidir. **horse** nesneleri, **track**'ın kurucu fonksiyonundaki **new** operatörünü kullanarak elde edilmişti. Dikkat ederseniz, şöyle yazamayız:

```
delete[] hArray;     //isaretcileri yok eder, ama atlari yok etmez
```


Bu, işaretçi dizisini yok eder, fakat işaretçilerin işaret ettiklerini yok etmez. Bunun yerine, dizinin elemanlarını birer birer gezmeli ve her atı tek tek yok etmeliyiz:

```
for(int j=0; j<total_horses; j++)      //atlari yok eder
      delete hArray[j];
```

putch() Fonksiyonu

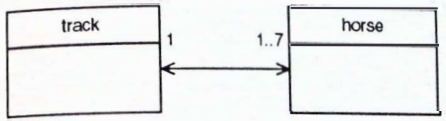
Her atın farklı bir renkte olmasını istiyoruz; fakat, derleyicilerin tümü **cout**'un renk üretmesine izin vermez. Bu, Borland C++ Builder'ın şimdiki sürümü için de geçerlidir. Buna rağmen, bazı eski C fonksiyonları, renk üretecektir. Bu nedenle, atları ekranda göstereceğimiz zaman **putch()** fonksiyonunu kullanılır:

```
putch(' '); putch('\xDB'); putch(horse_char); putch('\xDB');
```

Bu fonksiyon **CONIO.H** başlık dosyasını (derleyici ile birlikte gelir) gerektirir. Bu dosyayı **HORSE.CPP** içine açıkça dahil etmemize gerek yoktur, çünkü bu dosya **MSOFTCON.H** veya **BORLACON.H** dosyalarının içinde zaten vardır.

UML'de Çeşitlilik

Şimdi, **HORSE** programının Şekil 10.20'de gösterilen UML sınıf şemasına göz atalım. Bu şema, *çeşitlilik* (*multiplicity*) denilen bir UML kavramını tanıtacak.



ŞEKİL 10.20: HORSE programı için UML sınıf şeması.

Kimi zaman, **A** sınıfının sadece tek bir nesnesi, **B** sınıfının sadece tek bir nesnesi ile ilişkili olabilir. Diğer durumlarda, bir sınıfın birçok veya belli sayıdaki nesnesi bir bağlantı (association) içinde yer alabilir. Bir bağlantı içinde yer alan nesnelerin sayısına bağlantının *çeşitliliği* (multiplicity) denir. Sınıf şemalarında, çeşitliliği belirtmek için bağlantı çizgisinin her iki ucundaki sayılar veya simgeler kullanılır. Tablo 10.2'de UML çeşitlilik simgeleri listeleniyor.

TABLO 10.2: UML Çeşitlilik Simgeleri

Simge	Anlamı
1	Bir
*	Birkaç (0'dan sonsuza)
0..1	Hiç veya 1
1..*	Bir veya daha fazla
2..4	İki, üç veya dört
7,11	Yedi veya on bir

Eğer bir bağlantı çizgisi `A` sınıfı ucunda `1` ve `B` tarafı ucunda `*` simgesine sahipse bu, `A` sınıfının bir nesnesi `B` sınıfının belirsiz sayıda nesnesi ile etkileşim içinde demektir.

`HORSE` programında bir tane yarış pisti var ama, en fazla 7 tane at olabilir. Bu, bağlantı çizgisinin `track` ucunda `1` ve `horse` ucunda `1..7` ile belirtilir. Bir tane atın yarış için yeterli olduğunu varsayıyoruz. Zamana karşı yapılan antrenmanlarda bu söz konusu olabilir.

UML Durum Şemaları

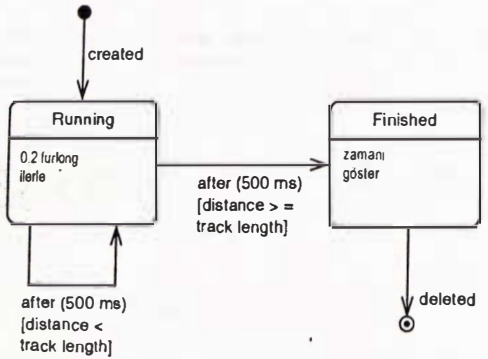
Bu bölümde yeni bir tür UML şeması tanıtacağız: *durum şeması* (durum grafik şeması da denir).

Önceki bölümlerde incelediğimiz UML sınıf şemaları sınıflar arası ilişkileri gösteriyordu. Sınıf şemaları, program kodunun organizasyonunu yansıtırlar. Bunlar statik şemalardır. Yani, bu ilişkiler (mesela, bağlantı ve genelleştirme) program çalıştığı sürece değişmezler. Bununla birlikte, belirli sınıf nesnelerinin zaman içindeki *dinamik* davranışlarını incelemek kimi zaman yararlı olur. Bir nesne oluşturulur, olaylardan veya programın diğer parçalarından gelen mesajlardan etkilenir, belki kararlar vermek durumunda kalır, çeşitli işler yapar ve son olarak yok edilir. Yani, nesnenin durumu zaman içinde değişir. Durum şemaları bunu grafik olarak gösterir.

Herkes *durum* kavramının günlük yaşantımızdaki cihazlara uygulanmış şekline aşinadır. Bir radyonun `açık` veya `kapalı` olmak üzere iki durumu vardır. Bir çamaşır makinesinin `Yıkama`, `Durulama`, `Sıkma` ve `Durma` durumları olabilir. Bir televizyonun halihazırda çektiği kanalların her biri için bir durumu vardır (`Channel 7 Active` durumu vs).

Durumlar arasında *geçişler* (*transitions*) yer alır. Kronometrenin diyelim ki 20 dakikaya erişmesi neticesinde çamaşır makinesi `Durulama` durumundan `Sıkma` durumuna bir geçiş yapar. Uzaktan kumanda cihazından gelen mesaj neticesinde TV, `Channel 7 Active` durumundan `Channel 2 Active` durumuna geçiş yapar.

Şekil 10.21'de, bu bölümde daha önce görülen `HORSE` programını baz alan bir durum şeması gösteriliyor. Şekil, bir `horse` nesnesinin program çalıştıkça içinde bulunabileceği farklı durumları gösteriyor.



ŞEKİL 10.21: Bir `horse` nesnesinin durum şeması.

Durumlar

UML durum şemalarında bir durum, köşeleri yuvarlatılmış bir dikdörtgen ile simgelenir. Durum ismi dikdörtgenin en üstünde yer alır. Durum isimleri genellikle büyük harfle başlar. İsmin altında, nesnenin bu duruma girdiği zaman gerçekleştirdiği *etkinlikler* (*activities*) yer alır.

Durum şemaları iki özel durum içerirler: Siyah yuvarlak, *başlangıç durumunu*; bir çember ile çevrelenmiş siyah yuvarlak ise *bitiş durumunu* simgeler. Bunlar şekilde gösterilmiştir.

Tanımlandıktan sonra bir `horse` nesnesi iki ana durumdan birinde olabilir: Bitiş çizgisine varmadan önce `Running` (çalışıyor) durumunda, sonrasında ise `Finished` durumundadır.

Sınıf şemasındaki sınıflardan farklı olarak, bir program kodunda durum şemasındaki durumların tam karşılığı yoktur. Hangi durumların dahil edileceğini bilmek için bir nesnenin kendisini hangi şartlar içinde bulacağı ve bunun neticesinde nasıl davranacağı hakkında fikriniz olması lazımdır. Bundan sonra durumlar için uygun isimler türetebilirsiniz.

Geçişler

Durumlar arası geçişler, bir dikdörtgenden diğerine yönlü oklar ile simgelenir. Eğer bir geçiş bir olay neticesinde gerçekleşiyorsa olay ismi, oka etiket olarak verilebilir. Şekildeki `created` (tanımlandı) ve `deleted` (silindi) geçişlerinde olduğu gibi... Geçiş isimlerinde büyük harf kullanılmaz. İsimler C++ kullanımından çok, İngilizce'ye daha yakındır.

Diğer iki geçiş, 500 milisaniyelik kronometre ile yapılan zamanlamaya bağlı olarak gerçekleştirilir. Bu geçişleri isimlendirmek için `after` anahtar sözcüğü kullanılır ve zaman `after`'a parametre olarak geçirilir.

Geçişler ayrıca, UML'in *koruyucu* (*guard*) olarak adlandırdığı bir isimle de etiketlendirilebilir. Eğer bir geçiş gerçeklenecekse, yerine getirilmesi gereken koşula koruyucu denir. Koruyucular köşeli parantez içinde yazılır. Şekildeki iki `after()` geçişi, ismin yanı sıra koruyucu da içerirler. Olaylar aynı olduğu için hangi geçişin gerçekleneceğini koruyucular belirler.

Dikkat ederseniz, bu geçişlerden biri *kendisine dönen* bir geçiştir (*self transition*); yani başladığı duruma geri döner.

Durumdan Duruma Yarışmak

`horse` nesnesi `Running` durumuna her girişinde, koşmuş olduğu mesafeyi 0.2 furlong artıran bir etkinliğin içine girer. At bitiş çizgisine varmadığı sürece, `[distance <track length]` koruyucusu doğru olur ve `Running` durumu yine kendisine döner. At bitiş çizgisine varınca, `[distance >= track length]` doğru olur ve `horse`, `Finished` durumuna geçer. `Finished` durumunda yarışın toplam süresi ekranda gösterilir. `horse`, sonra yok edilmeyi bekler.

Durum şemalarının ne işe yaradığı hakkında fikir vermesi için yeterince örnek gösterdik. Bunlar hakkında öğrenilecek elbette daha çok şey vardır. Çok daha karmaşık bir durum şemasını "Birden Fazla Dosya Kullanan Programlar" adlı Bölüm 13'te bir `elevator` nesnesini tanımlarken göreceğiz.

Hata Ayıklamak İçin Kullanılan İşaretçiler

İşaretçiler, gizemli ve felakete neden olabilecek program hatalarının kaynağı olabilirler. En çok yaşanan problem, programcının bir işaretçi değişkenine geçerli bir adres yerleştirmeyi başaramamış olmasıdır. Böyle bir durumda işaretçi bellekte herhangi bir yeri gösteriyor olabilir. Hatta program koduna veya işletim sistemine bile işaret ediyor olabilir. Eğer programcı daha sonra işaretçiyi kullanarak belleğe bir değer eklerse, bu değer programın veya işletim sistemi

komutlarının üzerine yazılacaktır. Bu durumda bilgisayar çökebilir veya hoş olmayan başka bir davranış ortaya çıkabilir.

Bu senaryonun özel bir versiyonu, işaretçi 0 adresine (0 adresi, `NULL` olarak adlandırılır) işaret ettiğinde yaşanır. Bu durum, örneğin, işaretçi değişkeni bir *global değişken* olarak tanımlandığında ortaya çıkar, çünkü global değişkenlere otomatik olarak 0 başlangıç değeri atanır. Aşağıdaki kısa program böyle bir durumu gösteriyor:

```
int* intptr;          //global degisken,baslangicta 0 degeri verilir
void main()
   {                  //intptr icine gecerli bir adres yerlestirilemez
   *intptr = 37;      //0 adresine 37 degerini koymaya calisilir
   }                  //sonuc hatalidir
```

`intptr` bir global değişken olduğu için `intptr`'a tanımlanırken 0 değeri verilir. Program gövdesindeki tek ifade, 0 adresine 37 değerini eklemeye çalışır.

Neyse ki, derleyici tarafından programa yerleştirilen çalışma zamanı hata kontrolü ünitesi (programın çalışması esnasındaki hataları kontrol eden ünite), 0 adresine erişmek için yapılan girişimleri bekler ve bir hata mesajı vererek (mesela, *access violation*, *null pointer assignment* veya *page fault* muhtemel mesajlar olabilir) programı sona erdirir. Eğer böyle bir mesaj görürseniz, bir işaretçiye doğru başlangıç değeri vermemiş olmanız, ihtimallerden biridir.

Özet

Bu, işaretçiler dünyasında fırtına gibi bir gezintiydi. Daha öğrenecek çok fazla konu vardır, ancak burada gördüklerimiz kitabın kalan bölümlerindeki örnekler için ve işaretçilerle ilgili daha sonraki çalışmalar için bir temel oluşturacaktır.

Bilgisayarın belleğindeki her şeyin bir adresi olduğunu ve bu adreslerin bir *işaretçi sabiti* olduğunu öğrendik. Değişkenlerin adreslerini adres operatörü (`&`) ile bulabiliriz.

İşaretçiler, adres değerlerini tutan değişkenlerdir. İşaretçiler, *işaret eder* anlamına gelen asteriks (`*`) kullanılarak tanımlanırlar. Derleyicinin neye işaret edildiğini bilmesi gerektiği için işaretçi tanımlarına her zaman bir veri tipi dahil edilir (`void*` hariç). Böylece derleyici, işaretçi üzerinde doğru aritmetik işlemleri gerçekleştirebilir. Asteriksi farklı biçimde, *dereferans* operatörü olarak kullanarak işaret edilen şeye erişebiliriz. Dereferans operatörü, *işaret edilen değişkenin içeriği* anlamını taşır.

Özel `void*` tipi, *herhangi* bir tipe işaret eden işaretçi demektir. Bu tipte bir işaretçi, aynı işaretçinin farklı tipteki adresleri tutması gerektiği bazı zor durumlarda kullanılır.

Dizi elemanları, köşeli parantezli dizi notasyonu ile veya asteriksli işaretçi notasyonu ile erişilebilir. Diğer adresler gibi, bir dizinin adresi de sabittir, fakat bu adres, artırılabilen veya başka yöntemlerle değiştirilebilen bir değişkene atanabilir.

Bir değişkenin adresi bir fonksiyona aktarıldığı zaman, fonksiyon orijinal değişken üzerinde işlem yapabilir. (Argümanlar değer olarak aktarıldığında bu geçerli değildir.) Bu bağlamda, argümanların işaretçiler üzerinden aktarılması referans yoluyla aktarma ile aynı avantajları sağlar. Gerçi, işaretçi argümanları *dereferans* edilmelidir veya dereferans operatörü kullanılarak erişilmelidir. Yine de, bazı durumlarda işaretçiler daha fazla esneklik sunarlar.

Bir karakter katarı sabiti, bir dizi veya bir işaretçi olarak tanımlanabilir. İşaretçi yaklaşımı daha esnek olabilir, fakat işaretçi değerinin bozulması gibi bir tehlike söz konusudur. Karakter katarları, `char` tipinde diziler olarak, sık sık fonksiyonlara aktarılır ve işaretçiler kullanılarak erişilirler.

`new` operatörü sistemden belirli bir miktar bellek alır ve bu belleğe işaret eden bir işaretçi döndürür. Bu operatör, programın çalışması sırasında değişkenlerin ve veri yapılarının oluşturulması için kullanılır. `delete` operatörü, `new` ile elde edilen belleği serbest bırakır.

Bir işaretçi bir nesneye işaret ettiği zaman nesnenin ait olduğu sınıfın üyeleri, erişim operatörü (`->`) kullanılarak erişilebilir. Yapı üyelerine erişmek için de aynı söz dizimi kullanılır.

Sınıflar ve yapılar, kendi tiplerine işaret eden veri üyeleri içerebilirler. Bu, daha karmaşık veri yapılarının geliştirilmesine imkan verir. Bağlı listeler bu tür veri yapılarına örnek teşkil eder.

İşaretçilere işaret eden işaretçiler de olabilir. Bu tür değişkenler çift asteriks kullanılarak tanımlanır; örneğin, `int ** pptr`.

UML sınıf şemalarındaki çeşitlilik, bir *bağlantı* (*association*) içinde yer alan nesnelerin sayısını gösterir.

UML durum şemaları, belirli bir nesnenin durumunun zaman içinde değişimini gösterir. Durumlar, köşeleri yuvarlatılmış dikdörtgenlerle, durumlar arası *geçişler* (*transitions*) ise yönlü çizgilerle temsil edilir.

Sorular

Soruların cevaplarını Ek G'de bulabilirsiniz.

1. `testvar` değişkeninin adresini ekrandan gösteren bir ifade yazın.z
2. `float` tipinde peş peşe iki değişkene işaret eden iki işaretçinin içeriği arasındaki fark _____.
3. Bir işaretçi
   a. bir değişkenin adresidir.
   b. bir sonra erişilecek olan değişkeni işaret eder.
   c. adresleri saklamak için kullanılan bir değişkendir.
   d. bir adres değişkeninin veri tipidir.
4. Aşağıdakiler için karşılık gelen deyimleri yazın.
   a. `var` değişkeninin adresi
   b. `var` tarafından işaret edilen değişkenin içeriği
   c. referans argümanı olarak kullanılan `var` değişkeni
   d. `char`'a işaret eden bir veri tipi
5. Bir adres, bir _____; bir işaretçi, bir _____.
6. `float`'a işaret eden bir değişkenin tanımını yazın.
7. _____ olmayan bir bellek adresine erişmek için işaretçiler kullanışlı bir yöntemdir.
8. Eğer `testptr` işaretçisi, `testvar` değişkenine işaret ediyorsa, `testvar`'ın ismini kullanmadan içeriğini simgeleyen bir ifade yazın.
9. Bir veri tipinin arkasına yerleştirilen bir asteriks _____ anlamına gelir. Bir değişken isminin önüne yerleştirilen asteriks _____ anlamına gelir.
10. `*test` deyimi için aşağıdakilerden hangisi (hangileri) söylenebilir?
    a. `test`'e işaret eden bir işaretçidir.
    b. `test`'in içeriği ile ilgilidir.
    c. `test`'in içeriğini gösterir.
    d. `test` tarafından işaret edilen değişkenin değeri ile ilgilidir.

11. Aşağıdaki kod doğru mudur?

```
int intvar = 333;
int* intptr;
cout << *intptr;
```

12. `void`'a işaret eden bir işaretçi, _____ işaret eden işaretçileri içerebilir.
13. `intarr[3]` ile `*(intarr+3)` arasındaki fark nedir?
14. 77 elemanı olan bir `intarr` dizisindeki değerlerin her birini ekranda göstermek için bir işaretçi notasyonu kullanan bir kod yazın.
15. Eğer `intarr` bir tamsayı dizisi ise, `intarr++` notasyonu niçin kurallara uygun olmuyor?
16. Fonksiyonlara argüman aktarmanın üç yolundan sadece _____ olarak aktarmak ve _____ yoluyla aktarmak, fonksiyonu çağıran programdaki argümanları fonksiyonun değiştirmesine imkan verir.
17. Bir işaretçinin işaret ettiği değişkenin tipi işaretçinin tanımının bir parçası olmalıdır. Böylece,
    a. veri tipleri, üzerinde aritmetik işlemler yapılınca birbirine karışmaz.
    b. yapı üyelerine erişmek için işaretçiler birbiri üzerine eklenebilir.
    c. kimsenin dini inançlarına saldırılamaz.
    d. derleyici, dizi elemanlarına erişmek için doğru aritmetik işlemlerini gerçekleştirebilir.
18. İşaretçi notasyonu kullanarak, `func()` isminde bir fonksiyon için bir prototip yazın. Fonksiyonun döndürdüğü tip `void` olmalı ve fonksiyon, tek argüman olarak `char` tipinde bir dizi almalı.
19. İşaretçi notasyonunu kullanarak `s1` karakter katarından `s2` karakter katarına 80 karakter transfer edecek bir kod yazın.
20. Bir karakter katarının ilk elemanı
    a. karakter katarının ismidir.
    b. karakter katarındaki ilk karakterdir.
    c. karakter katarının uzunluğudur.
    d. karakter katarını tutan dizinin ismidir.
21. İşaretçi notasyonunu kullanarak, `revstr()` isminde bir fonksiyon için bir prototip yazın. Fonksiyon bir karakter katarı değeri döndürmeli ve tek argüman olarak bir karakter katarı almalı.
22. `One`, `Two` ve `Three` string'lerine işaret eden işaretçileri tutan bir `numptrs` dizisinin tanımını yazın.
23. `new` operatörü
    a. bir değişkene işaret eden bir işaretçi döndürür.
    b. `new` adında bir değişken tanımlar.
    c. yeni bir değişken için bellekte yer ayırır.
    d. ne kadar kullanılabilir bellek alanı olduğunu söyler.
24. `new` operatörünü kullanmak, dizi kullanmaya kıyasla daha az bellek _____ neden olur.
25. `delete` operatörü _____ işletim sistemine döndürür.
26. `upperclass` tipinde bir nesneye işaret eden bir `p` işaretçisi olduğunu varsayarak, bu nesnenin `exclu()` üye fonksiyonunu çalıştıran bir deyim yazın.

27. `objarr` dizisinde indeks numarası 7 olan bir nesne olduğunu varsayarak, bu nesnenin `exclu()` üye fonksiyonunu çalıştıran bir deyim yazın.
28. Bağlı bir listede
    a. her bağlantı bir sonraki bağlantıya işaret eden bir işaretçi içerir.
    b. bir işaretçiler dizisi bağlantılara işaret eder.
    c. her bağlantı bir veri veya veriye işaret eden bir işaretçi içerir.
    d. bağlantılar bir dizide saklanır.
29. `float` tipinde değişkenlere işaret eden 8 işaretçiden oluşan bir `arr` dizisinin tanımını yazın.
30. Eğer bir çok sayıda büyük nesneleri veya yapıları sıralamak isterseniz, en etkili yöntem
    a. bunları bir diziye yerleştirmek ve diziyi sıralamaktır.
    b. bunlara işaret eden işaretçileri bir diziye yerleştirmek ve diziyi sıralamaktır.
    c. bunları bir bağlı listeye yerleştirmek ve listeyi sıralamaktır.
    d. bunların referanslarını bir diziye yerleştirmek ve diziyi sıralamaktır.
31. Bir uçta 10'dan daha az nesne içeren, diğer uçta 2'den daha fazla nesne içeren bir bağlantının (association) çeşitliliğini ifade edin.
32. Bir durum şemasındaki durumlar aşağıdakilerden hangisine (hangilerine) karşılık gelir?
    a. Nesneler arası mesajlara.
    b. Bir nesnenin içinde bulunduğu şartlara.
    c. Programın nesnelerine.
    d. Bir nesnenin konumundaki değişikliklere.
33. Doğru mu, Yanlış mı? Durumlar arasındaki geçişler, programın süresi boyunca mevcut kalır.
34. Bir durum şemasındaki koruyucu (guard)
    a. bir geçişin gerçeklenmesi için yerine getirilmesi gereken koşuldur.
    b. belirli geçişlere verilen bir isimdir.
    c. belirli durumlara verilen bir isimdir.
    d. belirli durumların oluşturulmasını kısıtlayan bir kısıtlamadır.

Alıştırmalar

Yıldızla işaretli soruların cevaplarını Ek G'de bulabilirsiniz.

1. Kullanıcının girdiği bir grup sayıyı okuyan ve bunları `float` tipinde bir diziye yerleştiren bir program yazın. Sayılar diziye yerleştirildikten sonra program sayıların ortalamasını bulmalı ve sonucu ekrana yazmalı. Mümkün olduğunca işaretçi notasyonu kullanın.*
2. Bu bölümde `NEWSTR` örneğindeki `String` sınıfı ile başlayın. Bir karakter katarını tamamen büyük harflere çeviren `upit()` adında bir üye fonksiyon ilave edin. `toupper()` kütüphane fonksiyonunu kullanabilirsiniz. `toupper()`, argüman olarak tek bir karakter alır ve büyük harfe çevrilmiş karakteri döndürür (eğer çevrilmesi gerekiyorsa). Bu fonksiyon `CCTYPE.H` başlık dosyasını kullanır. `upit()`'i denemek için `main()` içinde biraz kod yazın.*
3. Haftanın günlerini simgeleyen karakter katarlarına işaret eden bir işaretçi dizisi ile başlayın. Haftanın günlerini simgeleyen karakter katarlarını bu bölümdeki `PTRTOSTR` programında bulabilirsiniz. Bu bölümdeki `PTRSORT` programındaki `bsort()` ve `order()` fonksiyonlarının varyasyonlarını kullanarak karakter katarlarını alfabetik sıraya sokan

fonksiyonlar temin edin. Karakter katarlarının kendilerini değil, karakter katarlarına işaret eden işaretçileri sıralayın.*

4. **LINKLIST** programına bir yok edici fonksiyon ekleyin. Bu fonksiyon, bir **linklist** nesnesi yok edileceği zaman bağlantıların tümünü ortadan kaldırmalıdır. Zinciri takip ederek ve bir yandan ilerlerken diğer yandan bağlantıların her birini ortadan kaldırarak bu işlemi gerçekleştirebilir. Bağlantıların her biri yok edildikçe ekrana bir mesaj yazdırarak yok edici fonksiyonu test edebilirsiniz. Fonksiyon, listeye eklenenle aynı sayıda bağlantıyı ortadan kaldırmalıdır. (Program sona erdiğinde mevcut herhangi bir nesne için bir yok edici fonksiyon sistem tarafından otomatik olarak çağrılır.)*
5. Tümü aynı uzunlukta ve tipte (diyelim ki, **float** tipinde) üç tane yerel dizi içeren bir **main()** programınız olduğunu varsayın. İlk iki diziye başlangıçta değer atanmış olsun. **addarrays()** adında bir fonksiyon yazın. Bu fonksiyon üç dizinin adresini argüman olarak alsın; ilk iki dizinin içeriğini, elemanlarını tek tek ele alarak toplasın ve dönmeden önce sonucu üçüncü diziye yerleştirsin. Dizilerin boyutunu taşıyan dördüncü bir argüman bu fonksiyona aktarılabilir. Tüm işlemlerde işaretçi notasyonunu kullanın; köşeli parantezlere sadece dizilerin tanımı sırasında ihtiyaç duyacaksınız.
6. **strcmp(s1, s2)** kütüphane fonksiyonunun kendinize özgü versiyonunu yazın. Fonksiyon iki karakter katarını karşılaştırır ve eğer **s1** alfabetik olarak önce geliyorsa -1, **s1** ve **s2** aynı ise 0, **s2** alfabetik olarak önce geliyorsa 1 döndürür. Kendi fonksiyonunuza **compstr()** adını verin. Fonksiyonunuz argüman olarak iki **char*** karakter katarı almalı, bunları karakter karakter karşılaştırmalı ve bir **int** döndürmelidir. Değişik karakter katarı kombinasyonları ile fonksiyonunuzu test etmek için bir **main()** programı yazın. Tüm işlemlerde işaretçi notasyonu kullanın.
7. Bu bölüm yer alan **PERSORT** programındaki **person** sınıfını değiştirin. Öyle ki, **person** sınıfı sadece bir isim içermesin; ayrıca, kişinin maaşını simgeleyen **float** tipinde bir **salary** ögesi de içersin. **setName()** ve **printName()** üye fonksiyonlarını **setData()** ve **printData()** olarak değiştirmek zorunda kalacaksınız. Üstelik, yeni fonksiyonlara, ismin aynı sıra maaşı da ayarlayıp ekranda gösterecek beceriler dahil etmeniz gerekecek. Bir de **getSalary()** fonksiyonuna ihtiyacınız olacak. İşaretçi notasyonu kullanarak **persPtr** dizisindeki işaretçileri isme göre değil de, maaşa göre sıralayan bir **salsort()** fonksiyonu yazın. Sıralama için **PERSORT**'ta olduğu gibi başka bir fonksiyon çağırmaktansa tüm sıralama işlemlerini **salsort()** içinde yapmayı deneyin. Eğer bunu yapıyorsanız, -> operatörünün * operatöründen daha yüksek önceliğe sahip olduğunu unutmayın. Şu tür bir ifadeye de ihtiyacınız olacaktır:

```
if( (*(pp+j))->getSalary() > (*(pp+k))->getSalary() )
  { /*isaretcileri birbiriyle degistirin*/ }
```

8. **LINKLIST** programındaki **additem()** fonksiyonunu, eklemeleri listenin başı yerine sonuna yapacak şekilde değiştirin. Böylece, listeye ilk olarak eklenen öğe ekranda ilk olarak gösterilecektir. Programın çıktısı şöyle olur:

```
25
36
49
64
```


Bir öğe eklemek için işaretçi zincirini listenin sonuna kadar takip etmeniz gerekecek; sonra, en son bağlantıyı, yeni bağlantıya işaret edecek şekilde değiştirmeniz gerekecektir.

9. Diyelim ki, 100 tane tamsayıyı kolaylıkla erişilebilecek şekilde saklamanız gerekiyor. Bununla birlikte, bir de problem olduğunu varsayalım: Bilgisayarınızın belleği öylesine bölünmüş ki kullanabileceğiniz en büyük dizi sadece 10 tamsayı tutabiliyor. (Bu tür problemler gerçekten ortaya çıkar. Gerçi genellikle daha büyük bellek nesnelerinde bu tür problemler yaşanır.) Bu problemi, her biri 10 tamsayı içeren 10 tane ayrı **int** dizisi ve bu dizilere işaret eden 10 işaretçiden oluşan başka bir dizi tanımlayarak çözebilirsiniz. **int** dizileri **a0**, **a1**, **a2** gibi isimler alabilir. Bu dizilerin her birinin adresi **int*** tipinde bir işaretçi dizisinde saklanabilir. Bu işaretçi dizisine **ap** (array of pointers – işaretçi dizisi) gibi bir isim verebilirsiniz. Sonra, **ap[j][k]** gibi deyimler kullanarak tamsayılara tek tek erişebilirsiniz. Bu deyimde **j**, **ap** dizisi içindeki işaretçiler üzerinden adımlar; **k** ise dizilerin her birindeki tamsayılar üzerinden adımlar.
Bu dizileri test verileri ile (diyelim ki, 0, 10, 20 ve en son 990'a kadar olan sayılarla) doldurun. Sonra, verilerin doğruluğundan emin olmak için verileri ekranda yazdırın.
10. Anlatıldığı kadarıyla 9. alıştırma oldukça kaba sabadır; çünkü, 10 **int** dizisinin her biri ayrı isimler kullanılarak ayrı program ifadeleri ile tanımlanır. Ayrıca, her birinin adresi de ayrı birer ifade kullanılarak elde edilmelidir. **new** kullanarak işleri kolaylaştırabilirsiniz. **new**, diziler için gerekli belleği bir döngü içinde tahsis etmenize ve aynı anda bu dizilere birer işaretçi atamanıza imkan verir:

```
for(j=0; j<NUMARRAYS; j++)          //NUMARRAYS tane diziye yer ayir
  *(ap+j) = new int[MAXSIZE];       //dizilerin her biri MAXSIZE uzunlugunda
```

Bu yaklaşımı kullanarak 9. alıştırmadaki programı yeniden yazın. Her bir dizinin elemanlarına 9. alıştırmada bahsedilen deyimin aynısını kullanarak erişebilirsiniz veya işaretçi notasyonunu kullanabilirsiniz: ***(*(ap+j)+k)**. Her iki notasyon eşdeğerdir.
11. Onuncu alıştırmadaki 10 ayrı diziyi, tek indeksli dizi notasyonu kullanarak, tek bir-boyutlu dizi olarak ele almanıza imkan veren bir sınıf tanımlayın. Yani, sınıf üye fonksiyonlarının verilere iki adımlı yöntem kullanarak erişmeleri gerekiyor olsa bile **main()** içindeki ifadeler **a[j]** gibi deyimler kullanarak dizinin elemanlarına erişebilirler. Bu sonuca ulaşmak için indeks operatörünü ([]) ("Kalıtım" adlı Bölüm 9'a bakın) aşırı yükleyin. Dizileri test verileri ile doldurun, sonra bunları ekranda gösterin. "dizi" elemanlarına erişmek için **main()**'deki sınıf arayüzünde dizi notasyonu kullanılıyor olsa da, uygulamanızda tüm işlemler için (sınıf üye fonksiyonları içindekiler için) sadece işaretçi notasyonu kullanmalısınız.
12. İşaretçiler karışıktır. Öyleyse, işaretçilerin işleyişini bir sınıf ile simüle ederek işleyişlerini daha anlaşılır (ya da daha anlaşılmaz) yapabilir miyiz, bir bakalım.
"Ev yapımı" işaretçilerimizin işleyişine netlik kazandırmak için diziler kullanarak bilgisayarın belleğini modelleyeceğiz. Dizi erişimi iyi anlaşıldığı için bu yöntem sayesinde belleğe işaretçiler ile eriştiğimiz zaman gerçekte ne olduğunu anlayabilirsiniz.
Değişken tiplerinin tümünü saklamak için **char** tipinde tek bir dizi kullanmayı tercih ediyoruz. Bilgisayarın belleği de gerçekte böyledir: Her biri bir adrese (veya dizi dilinde söylersek, bir indekse) sahip olan byte'lar dizisi (bir byte, **char** tipiyle aynı büyüklüktedir). Bununla birlikte, C++ çoğunlukla bir **float** veya **int**'i bir **char** dizisine yerleştirmemize izin vermez. (Birleşimleri (unions) kullanabiliriz, fakat bu da bir başka

hikaye...) Bu nedenle, saklamak istediğimiz her veri tipi için ayrı bir dizi kullanarak belleği simüle edeceğiz. Bu alıştırmada kendimizi tek bir nümerik tip ile sınırlandıracağız: **float**. Bu nedenle, bu tipte bir diziye ihtiyacımız olacaktır. Bu diziyi **fmemory** olarak adlandıralım. Ancak, işaretçi değerleri (adresler) de bellekte saklanır. Yani, bunları saklamak için de bir başka diziye ihtiyacımız olacaktır. Dizi indekslerini adresleri modellemek için kullandığımız için ve büyük diziler haricindeki tüm dizilerin indeksleri **int** tipinde saklanabileceği için bu "işaretçileri" tutmak amacıyla bu tipte (bunu **pmemory** olarak adlandırın) bir dizi tanımlayacağız.

fmemory'in indeksi (buna **fmem_top** diyelim), bir **float** değerinin saklanabileceği ilk uygun yere işaret eder. **pmemory** için de benzer bir indeks vardır (buna da **pmem_top** diyelim). "Belleği" tüketeceğimizden endişelenmeyin. Bu dizilerin yeterince büyük olduğunu varsayacağız. Öyle ki, ne zaman bir şey saklamak istersek bu işlem, dizinin bir sonraki indeks numarasına saklayacağımız şeyi eklemekten ibaret olacaktır. Bunun haricinde, bellek yönetimi hakkında endişelenmemize gerek kalmayacak.

Float adında bir sınıf tanımlayın. Bu sınıfı, gerçek bellek yerine **fmemory**'de saklanacak **float** tipindeki sayıları modellemek için kullanacağız. **Float** içindeki tek örnek veri kendi "adresi"dir. Yani, **fmemory** içinde **float** değerinin saklı olduğu indekstir. Bu örnek değişkenine **addr** ismini verin. **Float** sınıfı ayrıca iki üye fonksiyona gereksinim duyar. İlki, **Float**'a bir **float** değeri atayacak olan tek argümanlı bir kurucu fonksiyon olmalıdır. Bu kurucu fonksiyon, **float** değerini **fmem_top** tarafından işaret edilen elemanın içinde saklar; ayrıca, **fmem_top**'un değerini de **addr** içine yerleştirir. Bu, derleyici ve bağlayıcının (linker) sıradan bir değişkeni gerçek belleğe yerleştirmek için yürütmeleri gereken düzenlemelere benzer. İkinci üye fonksiyon ise aşırı yüklenmiş **&** operatörü olmalıdır. Bu fonksiyon sadece **addr** içindeki işaretçi değerini (gerçekte **int** tipindeki indeksi) döndürmelidir.

ptrFloat adında ikinci bir sınıf tanımlayın. Bu sınıfın içindeki örnek veri, **pmemory** içindeki adresi (indeksi) tutar. Bu adreste de aslında başka bir adres (indeks) saklıdır. Bir üye fonksiyon bu "işaretçiye" bir **int** indeks değeri vererek başlangıç değeri atanmış olur. İkinci üye fonksiyon ise aşırı yüklenmiş ***** (dereferans veya "içerik") operatörüdür. Bunun işleyişi biraz daha karmaşıktır. Bu fonksiyon, **pmemory**'den adresi alır; bu adreste kendi verisi saklıdır (aslında bu veri de bir adrestir). Fonksiyon sonra bu yeni adresi **fmemory** için indeks olarak kullanarak, adres verisi tarafından işaret edilen **float** değerine ulaşır.

```
float& ptrFloat::operator*()
  {
  return fmemory[ pmemory [addr] ];
  }
```

Fonksiyon bu şekilde dereferans operatörünün (*****) işleyişini modeller. Dikkat ederseniz, bu fonksiyondan referans olarak dönmek zorundasınız. Bu nedenle, eşittir işaretinin sol tarafında ***** kullanabilirsiniz.

Float ve **ptrFloat** sınıfları aynıdır; fakat **Float**, **float s** değerini belleği simgeleyen bir dizi içinde saklar, **prtFloat** ise **int s** değerini (bellek işaretçisini simgeler ama aslında bir dizi indeksidir) yine belleği simgeleyen bir başka dizi içinde saklar.

Örnek bir **main()** programında, bu sınıfların tipik kullanımı aşağıdaki gibidir:


```
Float var1 = 1.234;                 //iki Float tanimla ve deger ata
Float var2 = 5.678;

ptrFloat ptr1 = &var1;              //Float'lara isaret eden iki isaretci tanımla,
ptrFloat ptr2 = &var2;              //bunlara Float'larin adres degerleri ata

cout << " *ptr1=" << *ptr1;         //Float'larin degerlerini dolayli yoldan
cout << " *ptr2=" << *ptr2;         //al ve bu degerleri ekranda goster

*ptr1 = 7.123;                      //ptr 1 ve ptr2 tarafindan isaret edilen
*ptr2 = 8.456;                      //degiskenlere yeni degerler ata

cout << " *ptr1=" << *ptr1;         //yeni degerleri dolayli yoldan al
cout << " *ptr2=" << *ptr2;         //ve ekranda goster
```

Dikkat ederseniz, değişken tiplerinin farklı isimlerinden başka bu, gerçek değişkenler üzerinde gerçekleştirilen işlemlerle tamamen aynı gözükür. Programın çıktısı işte şöyledir:

```
*ptr1=1.234
*ptr2=2.678
*ptr1=7.123
*ptr2=8.456
```

İşaretçileri uygulamak için bu dolambaçlı bir yol gibi görünebilir. Fakat, işaretçilerin ve adres operatörünün işleyişini açığa çıkararak bunların gerçek yapıları üzerine farklı bir perspektif sağlamış olduk.

ONBİRİNCİ BÖLÜM

11

SANAL FONKSİYONLAR

Sanal Fonksiyonlar

Arkadaş Fonksiyonlar

Statik Fonksiyonlar

Atama veya Kopyalama Yaparken İlk Kullanıma Hazırlamak

this İşaretçisi

Dinamik Tip Bilgisi

Artık işaretçiler hakkında biraz bilgi sahibi olduğumuza göre, daha ileri seviyeli C++ konularını derinlemesine araştırabiliriz. Bu bölüm nispeten birbiriyle gevşek bir ilişki içinde olan konuları ele alır: Sanal fonksiyonlar, arkadaş (`friend`) fonksiyonlar, statik fonksiyonlar, aşırı yüklenmiş = operatörü, aşırı yüklenmiş kopyalama kurucu fonksiyonu ve `this` işaretçisi. Bunlar ileri düzeyde özelliklerdir. Her C++ programı için, özellikle de çok kısa programlar için bunlar gerekli değildir. Buna rağmen, yaygın olarak kullanılırlar ve birçok tam kapsamlı program için gereklidirler. Özellikle sanal fonksiyonlar, nesne yönelimli programla. anın esaslarından biri olan çok biçimlilik için şarttır.

Sanal Fonksiyonlar

Sanal demek *görünüşte mevcut olan fakat gerçekte mevcut olmayan* demektir. Sanal fonksiyonlar kullanıldığında, bir sınıfın bir fonksiyonunu çağırıyor görünen bir fonksiyon, gerçekte farklı bir sınıfın fonksiyonunu çağırıyor olabilir. Sanal fonksiyonlara neden ihtiyaç duyulur? Varsayalım, farklı sınıflardan birkaç tane nesneye sahipsiniz; fakat, bunların hepsini bir diziye yerleştirmek istiyorsunuz ve aynık fonksiyon çağrısırı kullanarak bunlar üzerinde belli bir işlem yapmayı arzu ediyorsunuz. Örneğin, bir grafik programının birkaç çeşit şekil içerdiğini varsayalım: Bir üçgen, bir daire, bir kare vs. (Bölüm 9'da, "Kalıtım" bahsindeki `MULTSHAP` programındaki gibi.) Bu sınıfların her biri, nesnelerin ekranda görüntülenmesini sağlayan bir `draw()` üye fonksiyonu içerir.

Şimdi diyelim ki, bu elemanların birkaç tanesini bir arada gruplayarak bir resim yapmayı planlıyorsunuz ve resmi uygun bir yöntemle çizmek istiyorsunuz. Resimdeki değişik nesnelerin her biri için bir işaretçi tutan bir dizi tanımlamak yöntemlerden biri olabilir. Dizi şöyle tanımlanabilir:

```
shape* ptrarr[100];  //sekillere isaret eden 100 isaretcilik bir dizi
```

Şekillerin işaret eden işaretçileri bu diziye eklerseniz, basit bir döngü kullanarak resmin tamamını çizebilirsiniz:

```
for(int j=0; j<N; j++)
   ptrarr[j]->draw();
```

Bu hayret verici bir beceridir. Tamamen farklı fonksiyonlar aynı fonksiyon çağrısı ile çalıştırılır. `ptrarr`'daki işaretçi eğer bir daireye işaret ediyorsa daire çizen fonksiyon, eğer bir üçgene işaret ediyorsa üçgen çizen fonksiyon çağrılır. Buna *çok biçimlilik* (*polymorphism*) denir. Çok biçimlilik, *farklı formlar* anlamına gelir. Fonksiyonların görünümleri aynıdır: `draw()` deyimi. Fakat, `ptrarr[j]`'in içeriğine bağlı olarak gerçekte farklı fonksiyonlar çağrılır. Çok biçimlilik, nesne yönelimli programlamanın sınıflar ve kalıtımdan sonra en önemli özelliklerinden biridir.

Çok biçimli yaklaşımın işe yaraması için birkaç şartın yerine getirilmesi lazımdır. İlk önce, şekillerin farklı sınıflarının tümü, mesela daireler ve üçgenler, tek bir temel sınıftan türetilmelidir (`MULTSHAP` programında bu `shape` olarak adlandırılır). İkincisi, `draw()` fonksiyonu temel sınıf içinde `virtual` (sanal) olarak tanımlanmalıdır.

Bunların tümü biraz soyuttur. Bu nedenle, şimdi, bu durumun parçalarını gösteren bazı kısa örneklerle başlayalım ve sonra her şeyi bir araya getirelim.

İşaretçilerle Erişilen Normal Üye Fonksiyonlar

İlk örneğimiz, bir temel sınıf ve türetilmiş sınıfların tümünün aynı isimli fonksiyonlar içermesi durumunda ne olacağını gösterir. Üstelik, bu fonksiyonlara işaretçiler kullanarak, fakat sanal fonksiyonlar kullanmadan erişiyorsunuz. `NOTVIRT` programının listesi şöyledir:

```
// notvirt.cpp
// isaretci ile erisilen normal fonksiyonlar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Base                          //temel sinif
   {
   public:
      void show()                   //normal fonksiyon
         { cout << "Base\n"; }
   };
////////////////////////////////////////////////////////////
class Derv1 : public Base           //1. turetilmis sinif
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////
class Derv2 : public Base           //2. turetilmis sinif
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////
int main()
   {
   Derv1 dv1;                       //1. turetilmis sinif nesnesi
   Derv2 dv2;                       //2. turetilmis sinif nesnesi
   Base* ptr;                       //temel sinifa isaret eden isaretci

   ptr = &dv1;                      //dv1'in adresini isaretciye yerlestir
   ptr->show();                     //show()'u calistir
   ptr = &dv2;                      //dv2'nin adresini isaretciye yerlestir
   ptr->show();                     //show()'u calistir
   return 0;
   }
```

`Derv1` ve `Derv2` sınıfları `Base` temel sınıfından türetilir. Bu üç sınıfın her birinin bir `show()` üye fonksiyonu vardır. `main()`'de, `Derv1` ve `Derv2` sınıflarına ait nesneler ve `Base` sınıfına işaret eden bir işaretçi tanımlanır. Sonra, türetilmiş bir sınıf nesnesinin adresi, aşağıdaki ifade kullanılarak temel sınıf işaretçisinin içine konur:

```
ptr = &dv1;       //turetilmis sinifin adresi temel sinifin isaretcisine konur
```

Durun bir dakika, bundan nasıl kurtulabiliriz? Bir tipin (`Derv1`) adresini başka bir tipteki (`Base`) işaretçiye atıyoruz diye derleyici şikayetçi olmasın? Bilakis, derleyici son derece mutlu; çünkü, daha sonra netleşecek nedenlerden dolayı, bu durumda tip kontrolü gevşetiliyor. Kural

şu: Bir türetilmiş sınıfın nesnelerine işaret eden işaretçilerin tipleri, temel sınıfın nesnelerine işaret eden işaretçilerin tipleri ile uyumludur.

Şimdi sorun şu: Aşağıdaki satırı çalıştırdığınızda hangi fonksiyon çağrılır?

```
ptr->show();
```

Base::show() fonksiyonu mu, **Derv1::show()** fonksiyonu mu çağrılacaktır? **NOTVIRT**'in son iki satırında **Derv2** sınıfının bir nesnesinin adresi yine bir işaretçiye yerleştirilir ve yine şu satırı çalıştırılır:

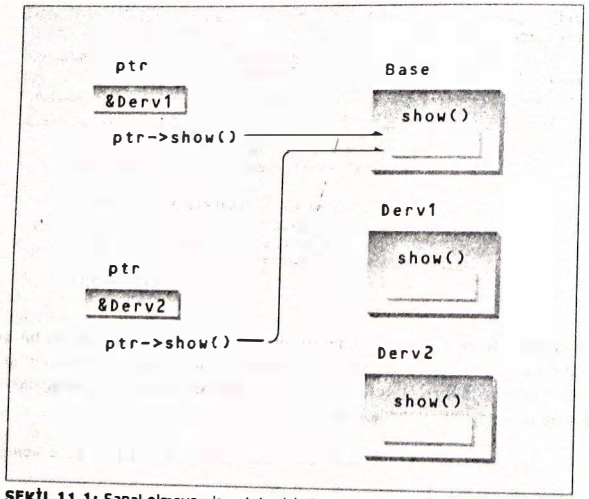
```
ptr->show();
```

Burada hangi **show()** fonksiyonu çağrılır? Programın çıktısı bu soruların cevabını verir:

```
Base
Base
```

Gördüğünüz gibi, her zaman temel sınıfın içindeki fonksiyon çalıştırılır. Derleyici, **ptr** işaretçisinin *içeriğini* dikkate almaz ve işaretçinin *tipi* ile eşlenen üye fonksiyonu tercih eder. Şekil 11.1'de bu durum gösterilmiştir.

Kimi zaman istediğimiz budur, fakat bu, bu bölümün başında ortaya atılan problemi çözmez: Aynı ifadeyi kullanarak farklı sınıfların nesnelerine erişmek.



ŞEKİL 11.1: Sanal olmayan işaretçi erişimi.

İşaretçilerle Erişilen Sanal Üye Fonksiyonlar

Programımızda tek bir değişiklik yapalım: Temel sınıftaki **show()** fonksiyonunun deklarasyonunun önüne **virtual** anahtar kelimesini yerleştirelim. Ortaya çıkan programın listesi şöyledir:

```
// virt.cpp
// isaretci ile erisilen sanal fonksiyonlar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Base                              //temel sinif
   {
   public:
      virtual void show()               //sanal fonksiyon
         { cout << "Base\n"; }
   };
////////////////////////////////////////////////////////////
class Derv1 : public Base               //1. turetilmis sinif
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////
class Derv2 : public Base               //2. turetilmis sinif
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////
int main()
   {
   Derv1 dv1;                           //1. Turetilmis sinifin nesnesi
   Derv2 dv2;                           //2. Turetilmis sinifin nesnesi
   Base* ptr;                           //temel sinifa isaret eden isaretci

   ptr = &dv1;                          //dv1'in adresini isaretciye yerlestir
   ptr->show();                         //show()'u calistir

   ptr = &dv2;                          //dv2'in adresini isaretciye yerlestir
   ptr->show();                         //show()'u calistir
   return 0;
   }
```

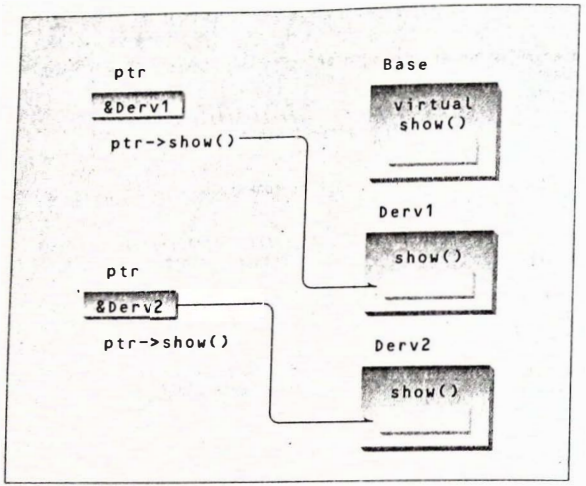
Bu programın çıktısı şöyledir:

```
Derv1
Derv2
```

Şimdi, gördüğünüz gibi, temel sınıfın değil de türetilmiş sınıfların üye fonksiyonları çalıştırılır. **ptr**'ın içeriği, **Derv1**'in adresi yerine **Derv2**'nin adresini gösterecek şekilde değiştirilir. **show()**'un çalıştırılan tek örneği de değişmiş olur. Böylece, aynı fonksiyon çağrısı, **ptr**'ın içeriğine bağlı olarak farklı fonksiyonları çalıştırır:

```
ptr->show();
```

Kural şudur: Derleyici, `ptr` işaretçisinin *tipine* değil de, *içeriğine* bağlı olarak fonksiyonu seçer; `NOTVIRT`'te olduğu gibi. Bu durum Şekil 11.2'de gösterilmiştir.



ŞEKİL 11.2: Sanal işaretçi erişimi.

Gecikmiş Bağlama

Zeki okuyucular, derleyicinin hangi fonksiyonu derleyeceğini nasıl bildiğini merak edebilirler. `NOTVIRT`'te derleyici aşağıdaki deyim ile bir problem yaşamaz:

```
ptr->show();
```

Derleyici, temel sınıfın `show()` fonksiyonuna yapılan çağrıları her zaman derler. Fakat, `VIRT` programında derleyici, `ptr`'ın hangi sınıfı içerebileceğini bilmez. `ptr`'ın içeriğinde `Derv1` sınıfına ait bir nesnenin adresi de olabilir; `Derv2` sınıfına ait bir nesnenin adresi de. Derleyici `draw()` fonksiyonunun hangi versiyonunu çağırır? Aslında, derleyici ne yapması gerektiğini bilmez; bu nedenle, bu kararın program çalışana kadar ertelenmesini sağlayacak bir düzenleme içine girer. Programın çalışması esnasında, `ptr`'ın hangi sınıfı işaret ettiği bilindiğinde, `draw()` fonksiyonunun uygun versiyonu çağrılacaktır. Buna *gecikmiş bağlama* (*late binding*) veya *dinamik bağlama* denir. (Fonksiyonları normal biçimde, derleme sırasında seçmeye *erken bağlama* (*early binding*) veya *statik bağlama* denir.) Gecikmiş bağlama bir miktar yük getirir; ancak, artırılmış güç ve esneklik sağlar.

Bu fikirleri kısa bir süre sonra bir araya getireceğiz, fakat öncelikle, sanal fonksiyonlar fikrine biraz incelik katalım.

Soyut Sınıflar ve Saf Sanal Fonksiyonlar

Bölüm 9'daki `MULTSHAP` programındaki `shape` sınıfını hatırlayın. `shape` sınıfının bir nesnesini asla oluşturmayacak, sadece daire ve üçgen gibi spesifik şekiller hazırlayacağız. Bir temel sınıfın nesnelerini asla örneklendirmeyeceksek, bu sınıf *soyut sınıf* olarak adlandırılır. Böyle bir sınıf, sadece türetilmiş sınıfların ebeveyni olarak davranmak üzere mevcuttur. Nesneleri örneklendirmek amacıyla türetilmiş sınıflar kullanılır. Bu özellik ayrıca sınıf hiyerarşisi için bir arayüz sağlar.

Sınıf ailemizi kullanan birine, temel sınıfın nesnelerini hiç kimsenin örneklemesini istemediğimizi nasıl açıkça ifade edebiliriz? Bunu dokümantasyonda belirtebiliriz ve sınıf kullanıcılarının bunu hatırlayacağına güvenebiliriz. Fakat, sınıflarımızı bu tür bir örneklemeye imkan vermeyecek şekilde yazmak elbette çok daha iyi bir yöntemdir. Bunu nasıl yapabiliriz? Temel sınıfın içine en azından bir tane *saf sanal fonksiyon* (*pure virtual function*) yerleştirerek. Saf sanal fonksiyon, deklarasyonuna `=0` deyimi ilave edilmiş olan fonksiyondur. Bu, `VIRTPURE` örneğinde gösterilmiştir.

```
// virtpure.cpp
// saf sanal fonksiyon
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base                          //temel sinif
   {
   public:
      virtual void show() = 0;      //saf sanal fonksiyon
   };
////////////////////////////////////////////////////////////////
class Derv1 : public Base           //1. turetilmis sinif
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv2 : public Base           //2. turetilmis sinif
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
// Base bad;                        //soyut sinifa ait bir nesne tanimlayamazsiniz
   Base* arr[2];                    //temel sinifa isaret eden isaretciler dizisi
   Derv1 dv1;                       //1. turetilmis sinifin nesnesi
   Derv2 dv2;                       //2. turetilmis sinifin nesnesi

   arr[0] = &dv1;                   //dv1'in adresini diziye yerlestir
   arr[1] = &dv2;                   //dv2'in adresini diziye yerlestir

   arr[0]->show();                  //her iki nesne icindeki show()'u calistir
   arr[1]->show();
   return 0;
   }
```

Bu programda sanal fonksiyon **show()**'un deklarasyonu şu şekilde yapılır:

```
virtual void show()=0;      //saf sanal fonksiyon
```

Buradaki eşittir işaretinin değer atama ile hiç ilgisi yoktur; 0 değeri hiçbir şeye atanmaz. =0 sözdizimi sadece, bir sanal fonksiyonun saf olacağını derleyiciye nasıl söyleyeceğimizi gösterir. Şimdi **main()**'de **Base** sınıfının nesnelerini tanımlamaya kalkışırsanız, derleyici, bir soyut sınıfın nesnesini örneklemeye çalıştığınız için şikayetçi olacaktır. Ayrıca, temel sınıfı soyut sınıf yapan saf sanal fonksiyonun ismini de size söyleyecektir. Dikkat ederseniz, bu sadece bir deklarasyon olsa da, temel sınıfın **show()** fonksiyonu için asla bir tanım yazmak zorunda değilsiniz. Gerçi, eğer gerekiyorsa, elbette yazabilirsiniz.

Bir saf sanal fonksiyonu bir temel sınıfın içine yerleştirdikten sonra, nesne örneklemek istediğiniz türetilmiş sınıfların hiç birinin, bu fonksiyonu dikkate almamasını sağlamalısınız. Eğer bir sınıf, bir saf sanal fonksiyonu dikkate almamazlık etmezse, bu kez kendisi sanal sınıf olur ve bu sınıftan nesne örnekleyemezsiniz (gerçi bu sınıftan türetilen sınıflardan örnekleyebilirsiniz). Tutarlı olmak açısından, temel sınıfın içindeki tüm sanal fonksiyonları saf yapmak isteyebilirsiniz.

Gördüğünüz gibi, **VIRTPURE** programında bir başka, alakasız bir değişiklik daha yaptık: Üye fonksiyonlarının adresleri bir işaretçi dizisinde saklanır ve dizi elemanları kullanılarak erişilir. Bu, tek bir işaretçi kullanmakla tamamen aynı şekilde çalışır. **VIRTPURE** programının çıktısı **VIRT**'inki ile aynıdır:

```
Derv1
Derv2
```

Sanal Fonksiyonlar ve person Sınıfı

Artık sanal fonksiyonların mekaniğinin bir kısmını öğrendiğimize göre, şimdi bunları kullanmanın mantıklı olduğu bir duruma göz atalım. Örneğimiz, "İşaretçiler" adlı Bölüm 10'daki **PTROBJ** ve **PERSORT** örneklerinin genişletilmiş bir versiyonudur. Örneğimiz de aynı **person** sınıfını kullanır; fakat, iki tane de türetilmiş sınıf ekler: **student** ve **professor**. Bu türetilmiş sınıfların her biri **isOutstanding()** isimli bir fonksiyon içerirler. Bu fonksiyon, Ödül Günü töreni için üstün öğrencilerin ve profesörlerin bir listesini oluşturarak okul idarecilerinin işini kolaylaştırır. **VIRTPERS** programının listesi şöyledir:

```
// virtpers.cpp
// person sinifi ile sanal fonksiyonlar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person                            //kisi sinifi
   {
   protected:
      char name[40];
   public:
      void getName()
         { cout << " Enter name: "; cin >> name; }
      void putName()
         { cout << "Name is: " << name << endl; }
      virtual void getData() = 0;       //saf sanal fonksiyon
      virtual bool isOutstanding() = 0; //saf sanal fonksiyon
   };
////////////////////////////////////////////////////////////////
class student : public person           //ogrenci sinifi
   {
   private:
      float gpa;                        //genel not ortalaması
   public:
      void getData()                    //kullanicidan ogrenci verilerini al
         {
         person::getName();
         cout << "   Enter student's GPA: "; cin >> gpa;
         }
      bool isOutstanding()
         { return (gpa > 3.5) ? true : false; }
   };
////////////////////////////////////////////////////////////////
class professor : public person         //profesor sinifi
   {
   private:
      int numPubs;                      //yayinlanan bildiri sayisi
   public:
      void getData()                    //kullanicidan profesor verilerini al
         {
         person::getName();
         cout << "      Enter number of professor's publications: ";
         cin >> numPubs;
         }
      bool isOutstanding()
         { return (numPubs > 100) ? true : false;}
   };
////////////////////////////////////////////////////////////////
int main()
   {
   person* persPtr[100];                //kisilere isaret eden isaretciler dizisi
   int n = 0;                           //listedeki kisi sayisi
   char choice;

   do {
      cout << "Enter student or professor (s/p): ";
      cin >> choice;
      if(choice=='s')                   //yeni ogrenciyi
         persPtr[n] = new student;      //diziye yerlestir
      else                              //yeni profesoru
         persPtr[n] = new professor;    //diziye yerlestir
      persPtr[n++]->getData();          //kisinin verilerini al
      cout << "   Enter another (y/n)?";   //bir kisi daha ister misiniz?
      cin >> choice;
      } while( choice=='y');            // 'y' girilene kadar don

   for(int j=0; j<n; j++)               //tum kisilerin
      {                                 //isimleri yazdir ve
      persPtr[j]->putName();            //ustun biriyse, belirt
      if( persPtr[j]->isOutstanding() )
         cout << "   This person is outstanding\n";
      }
   return 0;
   }     //main()'in sonu
```


Sınıflar

person sınıfı bir soyut sınıftır, çünkü saf sanal fonksiyonlar olan **getData()** ve **isOutstanding()**'i içerir. **person** nesnesi asla tanımlanamaz. Bu sınıf sadece, **student** ve **professor** sınıfları için bir temel sınıf olmak üzere mevcuttur. **student** ve **professor** sınıfları

temel sınıfa yeni veri öğeleri eklerler. **student** sınıfı, öğrencinin genel not ortalamasını (GPA) simgeleyen **float** tipinde bir **gpa** değişkeni içerir. **professor** sınıfı, profesörün yayınladığı akademik yayınların sayısını simgeleyen **int** tipinde bir **numPubs** değişkeni içerir. Genel not ortalaması 3.5'un üzerinde olan öğrenciler ile 100'den fazla bildiri yayınlamış profesörler üstün olarak nitelendirilir. (Bu kriterlerin akademik üstünlüğü ölçmek için ne kadar arzu edilir olduğu konusunda yorum yapmaktan kaçınacağız.)

isOutstanding() Fonksiyonu

isOutstanding() fonksiyonunun deklarasyonu **person** sınıfı içinde saf sanal fonksiyon olarak yapılır. **student** sınıfı içinde, eğer öğrencinin GPA'yi 3.5'tan yüksekse bu fonksiyon **true** değerini döndürür; aksi halde, **false** döndürür. **professor** sınıfı içinde, eğer profesörün **numPubs** değişkeni 100'den büyükse fonksiyon **true** değerini döndürür. **getData()** fonksiyonu bir **student** için kullanıcıya GPA verisini, bir **professor** için ise yayınlarının sayısını sorar.

main() Programı

main()'de öncelikle kullanıcının birkaç tane öğrenci ve öğretmen ismi girmesini sağlanır. Program ayrıca öğrenciler için GPA'i, profesörler için ise yayınlarının sayısını sorar. Kullanıcı veri girme işlemini bitirdikten sonra, program tüm öğrenci ve öğretmenlerin isimlerini, üstün özellikli olanlara da dikkat çekerek listeler. Programla gerçekleştirilebilecek örnek etkileşim şöyledir:

```
Enter student or professor (s/p): s
   Enter name: Timmy
   Enter student's GPA: 1.2
   Enter another (y/n)? y
Enter student or professor (s/p): s
   Enter name: Brenda
   Enter student's GPA: 3.9
   Enter another (y/n)? y
Enter student or professor (s/p): s
   Enter name: Sandy
   Enter student's GPA: 2.4
   Enter another (y/n)? y
Enter student or professor (s/p): p
   Enter name: Shipley
   Enter number of professor 's publications: 714
   Enter another (y/n)? y
Enter student or professor (s/p): p
   Enter name: Wainright
   Enter number of professor 's publications: 13
   Enter another (y/n)? n

Name is: Timmy
Name is: Brenda
This person is outstanding
Name is: Sandy
Name is: Shipley
This person is outstanding
Name is: Wainright
```


Bir Grafik Örneğinde Sanal Fonksiyonlar

Sanal fonksiyonları bir örnek üzerinde daha deneyelim. Bu kez "Kalıtım" adlı Bölüm 9'da yer alan **MULTSHAP** programından türetilen bir grafik örneğine bakalım. Bu bölümün başında belirttiğimiz gibi, aynı ifadeyi kullanarak birkaç sayıda şekil çizmek isteyebilirsiniz. **VIRTSHAP** programı bu işi gerçekleştirir. Hatırlarsanız, bu programı uygun bir konsol grafik dosyası ile kurmanız gerekir. Bunun nasıl yapılacağı "Console Graphics Lite" adlı Ek E'de anlatılmıştır.

```
// virtshap.cpp
// sekilleri ele alan sanal fonksiyonlar
#include <iostream>
using namespace std;
#include "msoftcon.h"                    //grafik fonksiyonlari icin
////////////////////////////////////////////////////////////////
class shape                              //temel sinif
   {
   protected:
      int xCo, yCo;                      //merkezin koordinatlari
      color fillcolor;                   //renk
      fstyle fillstyle;                  //dongu deseni
   public:                               //argumansiz kurucu fonk.
      shape() : xCo(0), yCo(0), fillcolor(cWHITE),
                                fillstyle(SOLID_FILL)
         { }                             //4 argumanli kurucu fonk.
      shape(int x, int y, color fc, fstyle fs) :
                           xCo(x), yCo(y), fillcolor(fc), fillstyle(fs)
         { }
      virtual void draw()=0              //saf sanal draw() fonksiyonu
         {
         set_color(fillcolor);
         set_fill_style(fillstyle);
         }
   };
////////////////////////////////////////////////////////////////
class ball : public shape
   {
   private:
      int radius;                        //(xCo, yCo): merkez
   public:
      ball() : shape()                   //argumansiz kurucu fonk.
         { }
                                         //5 argumanli kurucu fonk.
      ball(int x, int y, int r, color fc, fstyle fs)
                              : shape(x, y, fc, fs), radius(r)
         { }
      void draw()                        //topu ciz
         {
         shape::draw();
         draw_circle(xCo, yCo, radius);
         }
   };
////////////////////////////////////////////////////////////////
class rect ; public shape
   {
   private:
      int width, height;                 //(xCo, yCo): sol ust kose
   public:
```

```
        rect() : shape(), height(0), width(0)       //argumansiz kurucu fonk.
           { }                                      //6 argumanli kurucu fonk.
        rect(int x, int y, int h, int w, color fc, fstyle fs) :
                        shape(x, y, fc, fs), height(h), width(w)
           { }
        void draw()                                 //dikdortgeni ciz
           {
           shape::draw();
           draw_rectangle(xCo, yCo, xCo+width, yCo+height);
           set_color(cWHITE);                       //kosegeni ciz
           draw_line(xCo, yCo, xCo+width, yCo+height);
           }
   };
////////////////////////////////////////////////////////////////
class tria : public shape
   {
   private:
      int height;                                   //(xCo,     yCo): piramidin tepesi
   public:
      tria() : shape(), height(0)                   //argumansiz kurucu fonk.
         { }                                        //5 argumanli kurucu fonk.
      tria(int x, int y, int h, color fc, fstyle fs) :
                                shape(x, y, fc, fs), height(h)
         { }
      void draw()                                   //ucgeni ciz
         {
         shape::draw();
         draw_pyramid(xCo, yCo, height);
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   int j;
   init_graphics();                                 //grafik sistemini ilk kullanima hazirla

   shape* pShapes[3];                               //sekillere isaret eden isaretci dizisi
                                                    //uc sekil tanimla
   pShapes[0] = new ball(40, 12, 5, cBLUE, X_FILL);
   pShapes[1] = new rect(12, 7, 10, 15, cRED, SOLID_FILL);
   pShapes[2] = new tria(60, 7, 11, cGREEN, MEDIUM_FILL);

   for(j=0; j<3; j++)                               //sekillerin tumunu ciz
      pShapes[j]->draw();

   for(j=0; j<3; j++)                               //sekillerin tumunu yok et
      delete pShapes[j];
   set_cursor_pos(1,25);
   return 0;
   }
```

VIRTSHAP programındaki sınıf belirteçleri **MULTSHAP**'takilerle aynıdır; yalnızca, **shape** sınıfındaki **draw()** fonksiyonu bu kez saf sanal fonksiyon halini almıştır.

main()'de, şekillere işaret eden işaretçileri tutan bir dizi (**ptrarr**) kurulur. Sonra her sınıftan bir tane olmak üzere üç nesne tanımlanır ve bunların adresleri diziye yerleştirilir. Şimdi şekillerin üçünü de çizmek çok kolaydır. Aşağıdaki ifade, döngü değişkeni olan **j** değiştikçe bu işi gerçekleştirir:


```
ptrarr[j]->draw();
```

Bu yöntem, grafik elemanlarını birleştirmek için, özellikle çok sayıda nesnenin bir arada gruplanıp tek bir birim olarak çizilmesi gerektiği durumlarda güçlü bir yaklaşım olarak karşımıza çıkar.

Sanal Yok Edici Fonksiyonlar

Temel sınıfın yok edici fonksiyonları her zaman sanal olmalıdır. Diyelim ki, bir türetilmiş sınıf nesnesini yok etmek için türetilmiş sınıf nesnesine işaret eden bir temel sınıf işaretçisi ile birlikte **delete** operatörünü kullanıyorsunuz. Eğer temel sınıfın yok edici fonksiyonu sanal değilse, **delete**, türetilmiş sınıfın yok edici fonksiyonunu çağırmaz; normal bir üye fonksiyon gibi temel sınıfın yok edici fonksiyonunu çağırır. Bu, nesnenin sadece temel kısmının yok edilmesine neden olacaktır. **VIRTDEST** programı bunun nasıl gerçekleştiğini gösterir.

```
// vertdest.cpp
// sanal olmayan ve sanal yok edici fonksiyonlar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base
   {
   public:
      ~Base()                          //sanal olmayan yok edici fonk.
//    virtual ~Base()                  //sanal yok edici fonk.
         { cout << "Base destroyed\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv : public Base
   {
   public:
      ~Derv()
         { cout << "Derv destroyed\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Base* pBase = new Derv;
   delete pBase;
   return 0;
   }
```

Bu programın çıktısı şöyledir:

```
Base destroyed
```

Çıktı, nesnenin **Derv** kısmı için bir yok edici fonksiyon çağrılmadığını gösterir. Program listesinde temel sınıfın yok edici fonksiyonunun deklarasyonu sanal değildir. Fakat, deklarasyonun ilk tanımını açıklama haline getirip listeden çıkararak ve ikinci tanımı bunun yerine yerleştirerek yok edici fonksiyonunun deklarasyonunu sanal yapabilirsiniz. Çıktı artık şöyle olur:

```
Derv destroyed
Base destroyed
```

Şimdi, türetilmiş sınıf nesnesinin her iki kısmı da tam anlamıyla yok edilir. Elbette, yok edici fonksiyonların hiç birinin yapacak önemli bir işi yoksa (mesela, **new** ile elde edilen belleğin serbest bırakılması gibi), o zaman sanal yok edici fonksiyonların bir önemi yoktur. Ancak genel olarak, türetilmiş sınıf nesnelerinin tam olarak yok edildiğinden emin olmak için temel sınıfların yok edici fonksiyonlarının tümünü sanal yapmalısınız.

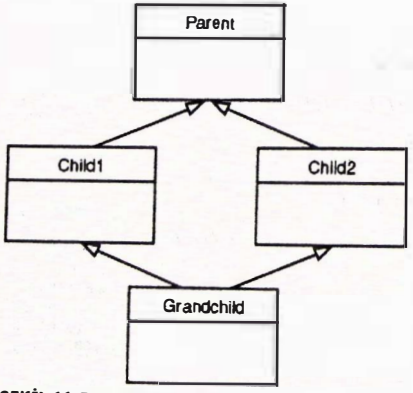
Sınıf kütüphanelerinin birçoğu sanal yok edici fonksiyon içeren bir temel sınıfa sahiptir. Sanal yok edici fonksiyon, türetilmiş sınıfların tümünde sanal yok edici fonksiyonların olmasını garanti eder.

Sanal Temel Sınıflar

Sanal programlama elemanları konusunu terk etmeden önce çoklu kalıtımla ilgisi olduğu için *sanal temel sınıflardan* da bahsetmeliyiz.

Şekil 11.3'te gösterilen duruma bir göz atalım. Şekilde bir temel sınıf olan **Parent**; iki türetilmiş sınıf olan **Child1** ve **Child2**; bir de, hem **Child1**'den hem de **Child2**'den türetilmiş dördüncü bir sınıf olan **Grandchild** mevcuttur.

Bu düzenlemede, **Grandchild** sınıfının bir üye fonksiyonu eğer **Parent** sınıfının verilerine veya fonksiyonlarına erişmek isterse problem ortaya çıkabilir. **NORMBASE** programı bu durumu açıklar.



ŞEKİL 11.3: Sanal temel sınıflar

```
// normbase.cpp
// temel sinifa yonelik belirsiz referans
class Parent
   {
   protected:
      int basedata;
   };
class Child1 : public Parent
   { };
class Child2 : public Parent
   { };
class Grandchild : public Child1, public Child2
   {
   public:
      int getdata()
         { return basedata; }       //HATA:belirsiz
   };
```


Grandchild'ın içindeki **getdata()** üye fonksiyonu **Parent**'ın içindeki **basedata**'ya erişmeye çalışırsa bir derleyici hatası ortaya çıkar. Niçin? **Child1** ve **Child2** sınıfları **Parent**'tan türetildiğinde, her biri **Parent**'ın bir kopyasını devralır; bu kopyaya *alt nesne* (subobject) denir. İki alt nesnenin her ikisi de, **basedata** da dahil olmak üzere **Parent**'ın verilerinin kendi kopyalarına sahiptir. Şimdi, **Grandchild**, **basedata**'yı kullanmak istediğinde iki kopyadan hangisine erişecektir? Durum belirsizdir ve derleyici de bunu rapor eder.

Bu belirsizliği ortadan kaldırmak için, **VIRTBASE** örneğinde gösterildiği gibi, **Child1** ve **Child2**, sanal temel sınıf haline çevrilir.

```
// virtbase.cpp
// sanal temel siniflar
class Parent
   {
   protected:
      int basedata;
   };
class Child1 : virtual public Parent       //Parent'in kopyasini paylasir
   { };
class Child2 : virtual public Parent       //Parent'in kopyasini paylasir
   { };
class Grandchild : public Child1, public Child2
   {
   public:
   int getdata()
   { return basedata; }                    //tamam:Parent'in sadece tek kopyasi
   };
```

Bu iki sınıfta **virtual** anahtar kelimesinin kullanılıyor olması, bu sınıfların temel sınıfı olan **Parent**'ın tek ortak alt nesnesini paylaşmalarına neden olur. **Basedata**'nın sadece tek kopyası olduğu için **Grandchild** içinden **basedata**'ya erişilmek istenirse belirsiz bir durum kalmaz.

Sanal temel sınıflara duyulan ihtiyaç, çoklu kalıtımın kullanımıyla ilgili kavramsal problemleriniz olduğuna işaret edebilir. Bu nedenle, sanal temel sınıflar dikkatle kullanılmalıdır.

Arkadaş Fonksiyonlar

Ortak özelliğe sahip verilerin paketlenmesi (encapsulation) ve veri gizliliği (data hiding) kavramları, üye olmayan fonksiyonların bir nesnenin private veya protected verilerine erişemeyeceklerine işaret eder. Kural şudur: Eğer üye değilseniz, içeriye giremezsiniz. Yine de, bu tür katı ayrımcılığın önemli ölçüde sıkıntıya sebep olduğu durumlar da söz konusudur.

Köprü Olarak Arkadaşlar

İki farklı sınıfın nesneleri üzerinde işlem yapacak bir fonksiyon istediğinizi hayal edin. Fonksiyon belki iki sınıfın nesnelerini argüman olarak alacak ve sınıfların özel verileri üzerinde işlem-

ler gerçekleştirecektir. Böyle bir durumda, **friend** fonksiyon gibisi yoktur. Aşağıda, **friend** fonksiyonların iki sınıf arasında nasıl köprü görevi görebileceğini gösteren basit bir örnek olan **FRIEND** programı görülüyor:

```
// friend.cpp
// arkadas fonksiyonlar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class beta;       //frifunc deklarasyonu icin gerekli

class alpha
   {
   private:
      int data;
   public:
      alpha() : data(3) { }              //argumansiz kurucu fonk.
      friend int frifunc(alpha, beta);   //arkadas fonksiyon
   };
////////////////////////////////////////////////////////////
class beta
   {
   private:
      int data;
   public:
      beta() : data(7) { }               //argumansiz kurucu fonk.
      friend int frifunc(alpha, beta);   //arkadas fonksiyon
   };
////////////////////////////////////////////////////////////
int frifunc(alpha a, beta b)             //fonksiyon tanimi
   {
   return(a.data + b.data);
   }
//--------------------------------------------------------
int main()
   {
   alpha aa;
   beta bb;

   cout << frifunc(aa, bb) << endl;      //fonksiyonu cagir
   return 0;
   }
```

Bu programda, söz konusu iki sınıf **alpha** ve **beta**'dır. Bu sınıfların içindeki kurucu fonksiyonlar, sınıfların tek veri öğelerine sabit başlangıç değerleri atar (**alpha**'ya 3 ve **beta**'ya 7 değeri atanır).

frifunc() fonksiyonunun her iki sınıfın private veri üyelerine erişmesini isteniyor. Bu nedenle, bu fonksiyonu **friend** fonksiyon yapılır. Fonksiyonun deklarasyonu her iki sınıfta da **friend** anahtar kelimesi kullanılarak yapılır:

```
friend int frifunc(alpha, beta);
```

Bu deklarasyon sınıf içinde herhangi bir yerde yer alabilir; **public** veya **private** bölümünde bulunması problem değildir.

Sınıfların her birinin bir nesnesi **frifunc()** fonksiyonuna argüman olarak aktarılır; fonksiyon, bu argümanlar üzerinden her iki sınıfın private **data** üyesine erişir. Fonksiyon fazla bir iş yapmaz. Veri öğelerini toplar ve toplamı döndürür. **main()** programı bu fonksiyonu çağırır ve sonucu ekrana yazar.

Önemsiz bir konu: Bir sınıfın deklarasyonu yapılana kadar kendisine erişilemeyeceğini hatırınızdan çıkarmayın. **alpha** sınıfının içindeki **frifunc()** fonksiyonunun deklarasyonunda **beta** geçer; bu nedenle, **beta**, **alpha**'dan önce tanımlanmalıdır. Aşağıdaki deklarasyon bu sebepten ötürü programın en başında yer alır.

```
class beta;
```

Duvarları Delmek

friend fonksiyonlarının tartışmalı olduğuna dikkat çekmek isteriz. C++'ın geliştirilmesi esnasında bu özelliği dahil etme isteğine yönelik argümanlar moda idi. Bir yandan, **friend** fonksiyonları dile esneklik kazandırırlar; öte yandan, veri gizliliği esasına uymazlar. Veri gizliliği, bir sınıfın private verilerine yalnızca bu sınıfın üye fonksiyonlarının erişebilmesi esasına dayanan bir felsefedir.

friend fonksiyonları kullanıldığında veri bütünlüğünün delinmesi ne kadar ciddidir? Bir **friend** fonksiyonu, verilerini erişeceği sınıfın içindeki fonksiyonlar gibi bildirilmelidir. Böylece, sınıfın kaynak koduna erişim izni olmayan bir programcı bir fonksiyonu **friend** olarak tanımlayamaz. Bu bağlamda, sınıfın bütünlüğü halen korunur. Böyleyken bile, **friend** fonksiyonları kavramsal olarak karmakarışıktır. Üstelik, eğer birkaç tane **friend** fonksiyonu sınıflar arasındaki net sınırları bulandırıyorsa, **friend** fonksiyonları potansiyel olarak karmaşık kodlara da yol açabilirler. Bu nedenle, **friend** fonksiyonları tedbirli kullanılmalıdır. Şayet kendinizi çok fazla **friend** içinde buluyorsanız, programınızın tasarımını yeniden düşünmeniz gerekiyor olabilir.

İngiliz Ölçü Sistemine Dayanan Distance Örneği

Yine de, kimi zaman **friend** fonksiyonları engel olunamayacak kadar uygun olabilirler. Belki de en yaygın örnek, aşırı yüklenmiş operatörlerin çok yönlülüğünü zenginleştirmek için **friend**'lerin kullanılmasıdır. Aşağıdaki program, **friend**'ler kullanılmadığında bu tür operatörlerin kullanımında yaşanan kısıtlamaları gösterir. Bu örnek, "Operatörlerin Aşırı Yüklenmesi" adlı Bölüm 8'deki **ENGLPLUS** ve **ENGLCONV** programlarının bir varyasyonudur. Bu programın adı **NOFRI**.

```
// nofri.cpp
// asiri yuklenen + operatorune bir kisitlama
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance        //Ingiliz olculerini kullanan Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0)    //kurucu fonk. (argumansiz)
         { }                               //kurucu fonk. (tek argumanli)
      Distance(float fltfeet)              //float'u Distance'a cevir
```

```
        {                                   //feet, sayinin tamsayi kismi
        feet = static_cast<int>(fltfeet);
        inches = 12*(fltfeet-feet);         //inches, geriye kalan kisim
        }
      Distance(int ft, float in)            //kurucu fonk. (iki argumanli)
        { feet = ft; inches = in; }
      void showdist()                       //uzakligi goster
        { cout << feet << "\'-" << inches << '\"'; }
      Distance operator + (Distance);
   };
//--------------------------------------------------------------
                                                //bunu d2'ye ekle
Distance Distance::operator + (Distance d2)     //toplami dondur
   {
   int f = feet + d2.feet;                  //feet'leri topla
   float i = inches + d2.inches;            //inches'leri topla
   if(i >= 12.0)                            //eger toplam 12.0'den buyukse,
      { i -= 12.0; f++; }                   //inches'i 12 azalt, feet'i 1 artir
   return Distance(f, i);                   //yeni Distance'i toplamla birlikte dondur
   }
//////////////////////////////////////////////////////////////
int main()
   {
   Distance d1 = 2.5;                       //kurucu fonk. float tipindeki
   Distance d2 = 1.25;                      //feet'i Distance'a cevirir
   Distance d3;
   cout << "\nd1 = "; d1.showdist();
   cout << "\nd2 = "; d2.showdist();

   d3 = d1 + 10.0;                          //distance + float :tamam
   cout << "\nd3 = "; d3.showdist();
// d3 = 10.0 + d1;                          //float + Distance :HATA
// cout << "\nd3 = "; d3.showdist();
   cout << endl;
   return 0;
   }
```

Bu programda, **Distance** tipinde iki nesneyi toplamak için + operatörü aşırı yüklenir. Ayrıca, ayağı ve ayağın ondalık kesrini simgeleyen **float** tipinde bir değeri, **Distance** değerine dönüştürmek için tek argümanlı bir kurucu fonksiyon da mevcuttur. (Yani, 10.25' değeri 10'-3" e dönüştürülür.)

Bu tür bir kurucu fonksiyon mevcutsa, **main()**'de aşağıdaki gibi ifadeler kurabilirsiniz:

```
d3 = d1 + 10.0;
```

Aşırı yüklenmiş + operatörü hem sol hem de sağ tarafında **Distance** tipinde nesneler arar; fakat, sağdaki argüman eğer **float** tipinde ise, derleyici bu **float**'u **Distance** değerine dönüştürmek için tek argümanlı kurucu fonksiyonu kullanır, sonra toplama işlemine devam eder.

Bu ifade üzerindeki ince bir değişiklik şöyle görünür:

```
d3 = 10.0 + d1;
```


Peki, bu işe yarar mı? Hayır, çünkü aşırı yüklenmiş + operatörünün üyesi olduğu nesne operatörün sol tarafında olmalıdır. Sol tarafa farklı tipte bir değişken veya bir sabit koyduğumuz zaman derleyici o tipte toplama yapan + operatörünü kullanır (burada söz konusu olduğu gibi); **Distance** nesnelerini toplayan operatörü kullanmaz. Ne yazık ki, bu operatör **float**'tan **Distance**'a nasıl dönüşüm yapılacağını bilmez; bu nedenle, bu durumu ele alamaz. **NOFRI** programının çıktısı şöyledir:

```
d1 = 2'-6"
d2 = 1'-3"
d3 = 12'-6"
```

İkinci toplama derlenmez; bu nedenle, bu ifadeler açıklama yapılarak programdan çıkarılmıştır. **Distance** tipinde yeni bir nesne tanımlayarak bu problemden kurtulabiliriz:

```
d3 = Distance(10, 0) + d1;
```

Fakat bu, hem sezgisel değildir hem de kabadır. Operatörün solunda üye olmayan veri tipleri içeren doğal görünümlü ifadeler nasıl yazabiliriz? Tahmin etmiş olabileceğiniz gibi, bir **friend** sizi bu ikilemden kurtarmaya yardımcı olabilir. **FRENGL** programı nasıl olabileceğini gösterir.

```
// frengl.cpp
// asiri yuklenmis arkadas + operatoru
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////////
class Distance       //ingiliz olculerini kullanan Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      Distance()                            //kurucu fonk. (argumansiz)
         { feet = 0; inches = 0.0; }
      Distance(float fltfeet)               //kurucu fonk. (tek argumanli)
         {                                  //float'u Distance'a cevir
         feet = int(fltfeet);               //feet, sayinin tamsayi kismi
         inches = 12*(fltfeet-feet);        //inches, kalan kisim
         }
      Distance(int ft, float in)            //kurucu fonk. (iki argumanli)
         { feet = ft; inches = in; }
      void showdist()                       //uzakligi goster
         { cout << feet << "\'-" << inches << '\"'; }
      friend Distance operator + (Distance, Distance);   //arkadas
   };
//
Distance operator + (Distance d1, Distance d2)          //d1'i d2'ye ekle
   {
   int f = d1.feet + d2.feet;                           //feet'leri topla
   float i = d1.inches + d2.inches;                     //inches'leri topla
   if(i >= 12.0)                        //eger inches 12.0'den buyukse,
      { i -= 12.0; f++; }               //inches'i 12 azalt, feet'i 1 artir
   return Distance(f, i);               //yeni Distance'i toplamla birlikte dondur
   }
```

```
//-------------------------------------------------------------
int main()
   {
   Distance d1 = 2.5;                  //kurucu fonk. float tipindeki
   Distance d2 = 1.25;                 //feet'i Distance'a cevirir
   Distance d3;
   cout << "\nd1 = "; d1.showdist();
   cout << "\nd2 = "; d2.showdist();

   d3 = d1 + 10.0;                     //distance +float :tamam
   cout << "\nd3 = "; d3.showdist();
   d3 = 10.0 + d1;                     //float +Distance :tamam
   cout << "\nd3 = "; d3.showdist();
   cout << endl;
   return 0;
   }
```

Aşırı yüklenmiş + operatörü, **friend** haline getirilir:

```
friend Distance operator + (Distance, Distance);
```

Dikkat ederseniz, aşırı yüklenmiş + operatörü üye fonksiyon olarak tek argüman alırken, **friend** fonksiyonu olarak iki argüman alır. Üye fonksiyon içinde + operatörünün üzerinde işlem yaptığı nesnelerden biri kendisinin üyesi olduğu nesne, diğeri ise argümandır. **friend** içinde ise, her iki nesne de argüman olmak zorundadır.

Aşırı yüklenmiş + fonksiyonunun gövdesindeki tek değişiklik, nesnenin verilerine doğrudan erişim için **NOFRI**'de kullanılan **feet** ve **inches** değişkenlerinin, **FRENGL**'da **d1.feet** ve **d1.inches** ile değiştirilmesidir. Çünkü bu nesne **FRENGL**'da argüman olarak aktarılır.

Bir fonksiyonu **friend** yapmak için sadece, sınıfın içindeki fonksiyon deklarasyonunun önüne **friend** anahtar kelimesini yerleştirmek gerektiğini hatırınızda tutun. Sınıf tanımı ve fonksiyona yapılan çağrılar normalde olduğu gibi yazılır.

Fonksiyonel Notasyonda friend'ler

Bir fonksiyon çağrısında kimi zaman bir **friend**, bir üye fonksiyondan daha belirgin bir söz dizimine imkan verir. Diyelim ki, İngiliz ölçü sistemine dayanan **Distance** sınıfının bir nesnesinin karesini alacak (kendisi ile çarpacak) ve sonucu ayak kare cinsinden **float** tipinde döndürecek bir fonksiyon istiyoruz. **MISQ** örneği bunun bir üye fonksiyon ile nasıl yapılabileceğini gösterir.

```
// misq.cpp
// Distance icin square() uye fonskiyonu
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance          //Ingiliz olculerini kullanan Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0)          //kurucu fonksiyon (argumansiz)
         { }
      Distance(int ft, float in) : feet(ft), inches(in)
                                                 //kurucu fonksiyon (iki argumanli)
         { }
      void showdist()                            //uzakligi goster
         { cout << feet << "\'-" << inches << '\"'; }
      float square();                            //uye fonksiyon
```



```
   };
//-------------------------------------------------------------
float Distance::square()                     //Bu uzakligin
   {                                         //karesini bul
   float fltfeet = feet + inches/12;         //float'a cevir
   float feetsqrd = fltfeet * fltfeet;       //karesini al
   return feetsqrd;                          //ayak kareyi dondur
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist(3, 6.0);                    //iki argumanli kurucu fonksiyon (3'-6")
   float sqft;

   sqft = dist.square();                     //dist'in karesini dondur
                                             //uzakligi ve karesini goster
   cout << "\nDistance = "; dist.showdist();
   cout << "\nSquare = " << sqft << " square feet\n";
   return 0;
   }
```

Programın **main()** bölümü bir **Distance** değeri tanımlar, bunun karesini alır ve sonucu ekrana yazdırır. Çıktı, orijinal uzaklığı ve karesini gösterir:

```
Distance = 3'-6"
Square = 12.25 square feet
```

main()'de aşağıdaki ifade kullanılarak **dist**'in karesini bulunur ve **sqft**'ye atanır.

```
sqft = dist.square();
```

Bu ifade idare eder, fakat sıradan sayılarla kullandığımız sözdiziminin aynısını kullanarak **Distance** nesneleri ile çalışmak istiyorsak muhtemelen bir fonksiyonel notasyon tercih etmek durumunda kalacağız:

```
sqft=square(dist);
```

FRISQ programında gösterildiği gibi, **square()** fonksiyonunu **Distance** sınıfının bir **friend**'i yaparak bu etkiye ulaşabiliriz:

```
// frisq.cpp
// Distance icin arkadas square() fonksiyonu
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance          //Ingiliz olculerini kullanan Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0)    //kurucu fonksiyon (argumansiz)
         { }
                                           //kurucu fonksiyon (iki argumanli)
      Distance(int ft, float in) : feet(ft), inches(in)
```

```
      { }
      void showdist()                          //uzakligi goster
         { cout << feet << "\'-" << inches << '\"'; }
      friend float square(Distance);           //arkadas fonksiyon
   };
//--------------------------------------------------------------
float square(Distance d)                       //bu uzakligin
   {                                           //karesini dondur
   float fltfeet = d.feet + d.inches/12;       //float'a cevir
   float feetsqrd = fltfeet * fltfeet;         //karesini al
   return feetsqrd;                            //ayak kareyi dondur
   }
//////////////////////////////////////////////////////////////
int main()
   {
   Distance dist(3, 6.0);                  //iki argumanli kurucu fonksiyon (3'-6")
   float sqft;

   sqft = square(dist);                    //dist'in karesini al
                                           //uzakligi ve karesini dondur
   cout << "\nDistance = "; dist.showdist();
   cout << "\nSquare = " << sqft << " square feet\n";
   return 0;
   }
```

square() fonksiyonu **MISQ**'te bir üye fonksiyon olarak hiç argüman almazken **FRISQ**'te **friend** olarak bir argüman alır. Genellikle, bir fonksiyonun **friend** versiyonu, fonksiyonun üye olduğu duruma kıyasla bir fazla argüman gerektirir. **FRISQ**'teki **square()** fonksiyonu **MISQ**'tekinin aynısıdır, fakat bu kez, kaynak **Distance** nesnesindeki verilere **feet** ve **inches** olarak değil, **d.feet** ve **d.inches** olarak erişir.

friend Sınıflar

Bir sınıfın bütününü **friend** yaparken aynı zamanda bu sınıfın üye fonksiyonlarının tamamı da **friend** yapılabilir. **FRICLASS** programı bunun nasıl gerçekleştirilebileceğini gösteriyor.

```
// friclass.cpp
// arkadas siniflar
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////////
class alpha
   {
   private:
      int data1;
   public:
      alpha() : data1(99) { }          //kurucu fonk.
      friend class beta;               //beta, bir arkadas siniftir
   };
//////////////////////////////////////////////////////////////
class beta
   {                                   //tum uye fonksiyonlar
   public:                             //ozel alpha verilerine erisebilir
      void func1(alpha a) { cout << "\ndata1 = " << a.data1; }
      void func2(alpha a) { cout << "\ndata1 = " << a.data1; }
   };
```



```
//////////////////////////////////////////////////////////////
int main()
   {
   alpha a;
   beta b;

   b.func1(a);
   b.func2(a);
   cout << endl;
   return 0;
   }
```

alpha sınıfı içinde **beta** sınıfının bütünü **friend** olarak ilan edilir. Böylece **beta**'nın üye fonksiyonlarının tümü, **alpha**'nın private verilerine erişebilir (bu programda tek bir veri öğesi mevcuttur: **data1**). Dikkat ederseniz, **friend** deklarasyonunda, **beta**'nın bir sınıf olduğu, **class** anahtar kelimesi kullanılarak belirtilir:

```
friend class beta;
```

beta'nın deklarasyonu, önceki örneklerde olduğu gibi, **alpha**'nın sınıf tanımından önce de yapılabilirdi:

```
class beta;
```

Sonra da, **alpha**'nın içinde **beta**, **class** anahtar kelimesi olmadan kullanılabilirdi:

```
friend beta;
```

Statik Fonksiyonlar

"Nesneler ve Sınıflar" adlı Bölüm 6'daki **STATIC** örneğinde **static** veri üyelerini tanıtmıştık. Hatırlarsanız, bir statik veri üyesinin her nesne için bir kopyası çıkarılmaz; bunun yerine, tek bir veri öğesi bir sınıfın tüm nesneleri tarafından paylaşılır. **STATIC** örneğinde, kendisine ait nesnelerin sayısını takip eden bir sınıf gösterilmişti. Bu kavramı biraz daha genişletelim ve verilerin yanı sıra fonksiyonların da nasıl **static** yapılacağını gösterelim. Örneğimiz, statik fonksiyonları tanıtmanın yanı sıra nesnelerinin her birine bir kimlik numarası atayan bir sınıfı da modelleyecektir. Bu, bir nesnenin öteki durumlar içinde hangi nesne olduğunu ortaya çıkarmak amacıyla nesneyi sorgulamanıza imkan verir. Bu, bir programda hata ayıklaması sırasında kimi zaman işe yarayabilecek bir beceridir. Program ayrıca yok edici fonksiyonların işleyişine bir parça ışık tutar. **STATFUNC** programının listesi şöyledir:

```
// statfunc.cpp
// statik fonksiyonlar ve nesneler icin kimlik numaralari
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////////
class gamma
   {
   private:
      static int total;                //bu sinifin toplam nesne sayisi
                                       //(sadece deklarasyon)
      int id;                          //bu nesnenin kimlik numarasi
```

```
    public:
       gamma()                              //argumansiz kurucu fonk.
          {
          total++;                          //bir nesne daha ekle
          id = total;                       //id, su anki toplama esit
          }
       ~gamma()                             //yok edici fonk.
          {
          total--;
          cout << "Destroying ID number " << id << endl;
          }
       static void showtotal()              //statik fonksiyon
          {
          cout << "Total is " << total << endl;
          }
       void showid()                        //statik olmayan fonksiyon
          {
          cout << "ID number is " << id << endl;
          }
    };
//-------------------------------------------------------------
int gamma::total = 0;                       //total'in tanimi
////////////////////////////////////////////////////////////////
int main()
   {
   gamma g1;
   gamma::showtotal();

   gamma g2, g3;
   gamma::showtotal();

   g1.showid();
   g2.showid();
   g3.showid();
   cout << "----------end of program----------\n";
   return 0;
   }
```

static Fonksiyonlara Erişmek

Bu programda, **gamma** sınıfı içinde bir statik veri üyesi (**total**) tanımlanır. Bu veri, sınıfın kaç tane nesnesinin mevcut olduğunu takip eder. Kurucu fonksiyon tarafından artırılır; yok edici fonksiyon tarafından da eksiltilir.

Varsayalım ki, **total**'e sınıfın dışından erişmek istiyoruz. **total**'in değerini ekrana yazan **showtotal()** adında bir fonksiyon tanımlıyoruz. Peki, bu fonksiyona nasıl erişeceğiz?

Bir veri üyesi **static** olarak bildirildiğinde, bu sınıfa ait kaç tane nesne tanımlanmış olursa olsun sınıfın bütünü için bu türde sadece bir değer mevcuttur. Hatta gerçekte bu tür nesneler hiç mevcut olmayabilir bile, fakat yine de bu gerçeği öğrenebilmeyi istiyoruz. Bir üye fonksiyonu çağırma işleminde kullanılmak üzere, aşağıda gösterildiği gibi bir dublör nesne tanımlayabiliriz:

```
gamma dummyObj;
dummyObj.showtotal();          //fonksiyonu cagirabilmek icin bir nesne tanimla
                               //fonksiyonu cagir
```


Fakat bu estetikten oldukça uzaktır. Sınıfın bütününü ilgilendiren bir şey yaparken spesifik bir nesneye erişmemize gerek olmamalıdır. Kapsam çözünürlük operatörü ile birlikte sınıfın ismini kullanmak çok daha mantıklıdır.

```
gamma::showtotal();          //daha mantikli
```

Yine de, **showtotal()** eğer normal bir üye fonksiyon ise bu işe yaramayacaktır; bu tür durumlarda bir nesne ve nokta (üye-erişim) operatörü gereklidir. **showdata()**'ya yalnızca sınıf ismini kullanarak erişmek için bu fonksiyonu **static** üye fonksiyon olarak bildirmemiz gerekir. **STATFUNC** programında yer alan aşağıdaki deklarasyonla biz de bunu yapıyoruz:

```
static void showtotal()
```

Artık sadece sınıf ismi kullanılarak fonksiyona erişilebilir. Programın çıktısı şöyledir:

```
Total is 1
Total is 3
ID number is 1
ID number is 2
ID number is 3
----------end of program--------
Destroying ID number 3
Destroying ID number 2
Destroying ID number 1
```

g1 adında bir tane nesne tanımlanır ve **total**'in değeri (yani 1) ekrana yazdırılır. Sonra iki tane daha nesne tanımlanır (**g2** ve **g3**) ve yine **total**'ın değeri (bu kez, 3) ekrana yazdırılır.

Nesneleri Numaralandırmak

Üyelerin tek tek kimlik numaralarını yazdırmak amacıyla **gamma()**'nın içine bir başka fonksiyon yerleştirdik. Bu kimlik numarası nesne tanımlanırken **total**'e eşitlenir. Böylece, her nesne tek numaraya sahip olur. **showid()** fonksiyonu ait olduğu nesnenin kimlik numarasını ekrana yazdırır. Aşağıdaki ifadeler kullanılarak bu fonksiyon **main()**'de üç kez çağrılır:

```
g1.showid();
g2.showid();
g3.showid();
```

Çıktıdan da görüldüğü üzere her nesnenin kimlik numarası tektir. **g1** nesnesi 1; **g2**, 2 ve **g3**, 3 olarak numaralandırılır.

Yok Edicileri Araştırmak

Artık nesneleri numaralandırmayı öğrendiğimize göre yok edici fonksiyonlar hakkında ilginç bir gerçeği araştırabiliriz. **STATFUNC**, en son ifadesinde ekrana *end of program* mesajı yazar; fakat, çıktıda da görülebileceği gibi aslında bu noktada program henüz tamamlanmış değildir. Tanımlanan üç nesne, program sona ermeden önce yok edilmek zorundadır. Böylece, bellek erişilemez durumda bırakılmamış olur. Derleyici, yok edici fonksiyonu çağırarak bu işi ele alır.

Bu işin bu şekilde gerçekleştiğini görmek için yok edici fonksiyonun içine ekrana mesaj yazan bir ifade ekleyebiliriz. Nesneleri numaralandırdığımız için nesnelerin yok ediliş sırasını da

bulabiliriz. Çıktıda da görüldüğü gibi, en son tanımlanan nesne olan **g3**, ilk önce yok edilir. Bu olaydan, yığında saklanan yerel nesnelerle ilgili son giren ilk çıkar prensibini çıkarmak mümkündür.

Atama veya Kopyalama Yaparken İlk Kullanıma Hazırlamak

C++ derleyicisi, sizin zahmet etmenize gerek kalmadan işleri sizin adınıza yürüterek her zaman yanınızda yer alır. Eğer siz devreye girerseniz kararları size bırakır; aksi halde, işleri kendi bildiği şekilde gerçekleştirir. Bu işlemin iki önemli örneği atama operatörü ve kopyalama kurucu fonksiyonudur.

Atama operatörünü, muhtemelen hakkında çok fazla düşünmeden, pek çok kere kullandınız. Varsayalım, **a1** ve **a2** birer nesne olsun. Derleyiciye aksini söylemediğiniz müddetçe, şu ifade derleyicinin, **a1**'in verilerini üye üye **a2**'ye kopyalamasına neden olur:

```
a2 = a1;        //a2'ye a1'in degerini ver
```

Bu atama operatörünün (=) varsayılan davranış biçimidir.

Ayrıca, değişkenlere ilk değeri atama kavramına da aşinasınız. Aşağıdaki ifadede olduğu gibi, bir nesneyi bir başka nesne yardımıyla ilk kullanıma hazırlamak da benzer bir etkiye neden olur:

```
alpha a2(a1);      //a2'ye a1'in degerini ata
```

Derleyici **a2** adında yeni bir nesne tanımlar ve **a1**'in verisini üye üye **a2**'ye kopyalar. Bu, kurucu fonksiyonun varsayılan davranış biçimidir.

Varsayılan bu etkinliklerin her ikisi de derleyici tarafından, ücretsiz olarak, sunulur. Eğer istediğiniz üyelerin tek tek kopyalanması ise başka bir işlem yapmanıza gerek yoktur. Ancak, değer atama veya ilk kullanıma hazırlama işlemlerinde daha karmaşık bazı şeylerin yapılmasını istiyorsanız, fonksiyonların varsayılan işlevlerini dikkate almayabilirsiniz.

Atama operatörünü ve kopyalama kurucu fonksiyonunu aşırı yüklemek için uygulanan teknikleri ayrı ayrı ele alacağız; sonra, **String** sınıfına belleği daha etkili biçimde yönetme imkanı veren bir örnek üzerinde her ikisini birleştireceğiz. Ayrıca yeni bir UML özelliğini daha tanıtacağız: nesne şeması.

Atama Operatörünü Aşırı Yüklemek

Şimdi, atama operatörünü aşırı yüklemek için kullanılan tekniği gösteren kısa bir örneğe bakalım. **ASSIGN** programını listesi şöyledir:

```
// assign.cpp
// atama operatorunun (=) asiri yuklenmesi
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class alpha
   {
   private:
      int data;
   public:
      alpha()                          //argumansiz kurucu fonk.
         { }
      alpha(int d)                     //tek argumanli kurucu fonk.
         { data = d; }
      void display()                   //verileri goster
         { cout << data; }
      alpha operator = (alpha&a)       //asiri yuklenen = operatoru
         {
         data = a.data;                //otomatik olarak yapılmaz
         cout << "\nAssignment operator invoked";
         return alpha(data);           //bu alpha'nın bir kopyasını dondur
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   alpha a1(37);
   alpha a2;

   a2 = a1;                            //asiri yuklenen =
   cout << "\na2="; a2.display();       //a2'yı goruntule

   alpha a3 = a2;                      //= operatorunu cagirmaz
   cout << "\na3="; a3.display();       //a3'u goruntule
   cout << endl;
   return 0;
   }
```

alpha sınıfı çok basittir ve sadece bir tane veri üyesi içerir. Kurucu fonksiyonlar veriye başlangıç değeri atar ve bir üye fonksiyon verinin değerini ekrana yazar. **ASSIGN** programının yeni bir özelliği, = operatörünün aşırı yüklendiği **operator=()** fonksiyonudur.

main() içinde **a1** tanımlanır ve **a1**'e 37 değeri verilir. **a2** de tanımlanır ama buna bir değer verilmez. Sonra, **a2**'ye **a1**'in değerini vermek için atama operatörü kullanılır:

```
a2 = a1;          //atama ifadesi
```

Bu işlem, aşırı yüklenmiş **operator=()** fonksiyonun çağrılmasına neden olur. **ASSIGN** programının çıktısı şöyledir:

```
Assignment operator invoked
a2=37
a3=37
```

İlk Kullanıma Hazırlamak, Değer Atamak Değildir

ASSIGN programının son iki satırında **a3** nesnesi **a2**'nin değeriyle ilk kullanıma hazırlanır (ilk değeri verilir) ve bu, ekranda görüntülenir. Burada kullanılan söz dizimi aklınızı karıştırmasın. Aşağıdaki ifadedeki eşittir işareti

```
alpha a3 = a2;       //kopyalayarak deger vermek; deger atamak degil
```

atama (assignment) değildir; ilk kullanıma hazırlamaktır (initialization) ve aşağıdaki ifadeyle aynı etkiye sahiptir:

```
alpha a3(a2);          //kopyalayarak deger vermenin alternatif sekli
```

Atama operatörünün sadece bir kez çalıştırılmasının nedeni budur. **ASSIGN** programının çıktısındaki aşağıdaki satırın bir kez çağrılmış olmasından da bunu anlıyoruz:

```
Assignment operator invoked
```

Sorumluluğu Almak

= operatörünü aşırı yüklediğinizde, varsayılan atama operatörünün gerçekleştirdiği işlerin tümünün sorumluluğunu da üstlenirsiniz. Bu genellikle bir nesnenin veri üyelerini bir başka nesneye kopyalamayı içerir. **ASSIGN** programındaki **alpha** sınıfının sadece bir tane veri üyesi vardır: **data**. Bu nedenle, **operator=()** fonksiyonu aşağıdaki ifade ile bu veri üyesinin değerini kopyalar:

```
data = a.data;
```

Fonksiyon ayrıca **Assignment operator invoked** mesajını da ekrana yazar; böylece, fonksiyonun ne zaman çalıştırıldığını anlayabilirsiniz.

Referans Olarak Aktarmak

Dikkat ederseniz argüman, **operator=()** fonksiyonuna referans olarak aktarılır. Bunu yapmak tamamen gerekli değilse de, genellikle iyi bir fikirdir. Niçin? Bildiğiniz gibi, değer olarak aktarılan bir argüman, aktarıldığı fonksiyonda kendisinin bir kopyasını çıkarır. **operator=()** fonksiyonuna aktarılan argüman da bir istisna değildir. Bu tür nesneler eğer büyükse, kopyalama fazla miktarda bellek israf edebilir. Referans olarak aktarılan değerler kopya üretmez. Böylece, belleği de korumuş olurlar.

Ayrıca, nesnelerin sayısını takip etmek istediğiniz bazı durumlar olabilir (**STATFUNC** örneğinde olduğu gibi; bu örnekte nesnelere numara vermiştik). Atama operatörünü her kullanışınızda eğer derleyici fazladan nesneler üretiyorsa, kendinizi beklediğinizden çok daha fazla nesne ile çevrelenmiş bulabilirsiniz. Referans olarak aktarmak bu tür yapay nesne üretimini önlemeye yardımcı olur.

Bir Değer Döndürmek

Önceden gördüğümüz gibi, bir fonksiyon kendisini çağıran programa değer olarak veya referans yoluyla bilgi döndürebilir. Bir nesne değer olarak döndürüleceği zaman, yeni bir nesne oluşturulur ve fonksiyonu çağıran programa döndürülür. Fonksiyonu çağıran programda bu nesnenin değeri yeni bir nesneye atanabilir veya başka şekillerde kullanılabilir. Bir nesne referans yoluyla döndürüldüğünde yeni bir nesne oluşturulmaz. Fonksiyonu çağıran programa döndürülen sadece, fonksiyon içinden orijinal nesneye yapılan referanstan ibarettir.

ASSIGN programındaki **operator=()** fonksiyonu, geçici bir **alpha** nesnesi oluşturarak bir değer döndürür ve bunu, tek argümanlı kurucu fonksiyon kullanarak, aşağıdaki ifade ile, ilk kullanıma hazırlar:

```
return alpha(data);
```

Döndürülen değer, aşırı yüklenmiş = operatörünün üyesi olduğu nesnenin aynısı değil, bir kopyasıdır. Bir değer döndürülmesi, = operatörlerini zincirleme kullanmaya imkan verir:


```
a3 = a2 = a1;
```

Ancak, değer döndürmek, argümanı değer olarak aktarmakla aynı dezavantajlara sahiptir: Bellek harcayan fazladan bir kopya oluşturur ve bu, karışıklığa neden olabilir. Aşırı yüklenmiş = operatörü için gösterilen deklaratörü kullanarak bu değeri referans ile döndürebilir miyiz?

```
alpha& operator = (alpha& a)     //bu durumda kotu bir fikir
```

Ne yazık ki, bir fonksiyona yerel olan değişkenler üzerinden referans olarak dönüş gerçekleştiremezsiniz. Hatırlarsanız, yerel (**automatic**) değişkenler – yani, bir fonksiyon içinde tanımlananlar (ve **static** olarak belirtilmeyenler) – fonksiyon döndüğü zaman ortadan kaldırılırlar. Referans yoluyla yapılan bir dönüş, sadece döndürülen verinin adresini döndürür; yerel değişkenler için bu adres, fonksiyonun içindeki veriye işaret eder. Fonksiyon sona erdiğinde ve bu veri ortadan kaldırıldığında, bu işaretçi anlamsız bir değerle kalır. Derleyiciniz belki bu kullanımı bir uyarı ile haber verebilir. (Bu bölüm içinde "**this** İşaretçisi" başlığı altında bu problemi çözmenin bir yolunu göreceğiz.)

Kalıtım Yok

Operatörler içinde atama operatörü kalıtım uygulanamayan tek operatördür. Bir temel sınıf içinde atama operatörünü aşırı yüklerseniz, türetilmiş sınıfların herhangi birinde bu fonksiyonun aynını kullanamazsınız.

Kopyalama Kurucu Fonksiyonu

Önceden ele aldığımız gibi, iki tür ifade yardımıyla, bir nesneyi tanımlayabilir ve aynı anda bir başka nesnenin değerini bu nesneye atayabilirsiniz:

```
alpha a3(a2);      //kopyalayarak ilk kullanima hazirlamak
alpha a3 = a2;     //kopyalayarak ilk kullanima hazirlamak, alternatif soz dizimi
```

Tanımların her iki stili de bir kurucu fonksiyon çağırır: Yeni bir nesne tanımlayan ve argümanını bu nesneye kopyalayan bir kurucu fonksiyon. Derleyici tarafından her nesne için otomatik olarak sağlanan varsayılan kopyalama kurucu fonksiyonu, üyeleri tek tek kopyalayarak kopyalamayı gerçekleştirir. Bu atama operatörünün yaptığı ile aynıdır; aralarındaki fark, kopyalama kurucu fonksiyonunun ayrıca yeni bir nesne tanımlıyor olmasıdır. Atama operatörü gibi kopyalama kurucu fonksiyonunu da kullanıcı tarafından aşırı yüklenebilir. **XOFXREF** örneği bunun nasıl yapılabileceğini gösteriyor:

```
// xofxref.cpp
// kopyalama kurucu fonksiyonu:X(X&)
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class alpha
   {
   private:
      int data;
   public:
      alpha()                          //argumansiz kurucu fonk.
         { }
```

```
        alpha(int d)                    //tek argumanli kurucu fonk.
           { data = d; }
        alpha(alpha &a)                 //kopyalama kurucu fonk.
           {
           data = a.data;
           cout <<."\nCopy constructor invoked";
           }
        void display()                  //ekranda goster
           { cout << data; }
        void operator = (alpha&a)       //asiri yuklenen = operatoru
           {
           data = a.data;
           cout << "\nAssignment operator invoked";
           }
    };
////////////////////////////////////////////////////////////////
int main()
    {
    alpha a1(37);
    alpha a2;

    a2 = a1;                            //asiri yuklenen = operatoru cagir
    cout << "\na2="; a2.display();       //a2'yi goster

    alpha a3(a1);                       //kopyalama kurucu fonk. cagir
//  alpha a3 = a1;                      //a3'un esdeger tanimi
    cout << "\na3="; a3.display();       //a3'u goster
    cout << endl;
    return 0;
    }
```

Bu program hem atama operatörünü hem de kopyalama kurucu fonksiyonunu aşırı yükler. Aşırı yüklenmiş atama operatörü **ASSIGN** örneğindekiyle aynıdır. Kopyalama kurucu fonksiyonu tek argüman alır. Bu, **alpha** tipinde ve referans olarak aktarılan bir nesnedir. Fonksiyonun deklarasyonu şöyledir:

```
alpha(alpha&)
```

Bu deklarasyon, **X(X&)** ("X of X ref" - "X referansının X'i" olarak telaffuz edilir) formundadır. **XOFXREF** programının çıktısı şöyledir:

```
Assignment operator invoked
a2=37
Copy constructor invoked
a3=37
```

Şu ifade

```
a2 = a1;
```

atama operatörünü kullanırken; şu ifade

```
alpha a3(a1);
```


kopyalama kurucu fonksiyonunu çağırır. Kopyalama kurucu fonksiyonunu çağırmak için eşdeğer bir ifade de kullanılabilir:

```
alpha a3 = a1;
```

Kopyalama kurucu fonksiyonunun bir nesne tanımlandığı sırada çağrılabileceğini görmüştük. Ayrıca, bir argüman bir fonksiyona değer olarak aktarıldığında ve fonksiyondan bir değer döndürüldüğünde de kopyalama kurucu fonksiyon çağrılır. Şimdi, bu durumları kısaca gözden geçirelim.

Fonksiyon Argümanları

Bir nesne değer olarak bir fonksiyona aktarıldığında bir kopyalama kurucu fonksiyon çağrılır. Kopyalama kurucu fonksiyon, söz konusu fonksiyonun üzerinden işlem yapacağı kopyayı üretir. Yani, eğer **XOFXREF**'te aşağıdaki gibi bir deklarasyon olsaydı

```
void func(alpha);
```

ve bu fonksiyon şu ifade ile çağrılıyor olsaydı,

```
func(a1);
```

kopyalama kurucu fonksiyon, **func()** tarafından kullanılmak üzere **a1** nesnesinin bir kopyasını oluşturmak amacıyla çağrılmış olacaktı. (Elbette, argüman eğer referans yoluyla veya işaretçi üzerinden aktarılırsa kopyalama kurucu fonksiyon çağrılmaz. Bu gibi durumlarda kopya oluşturulmaz; fonksiyon orijinal değişen üzerinde işlem yapar.)

Fonksiyon Değer Döndürür

Kopyalama kurucu fonksiyon ayrıca, bir fonksiyondan bir değer döndürüldüğünde geçici bir nesne oluşturur. **XOFXREF**'te böyle bir fonksiyon olduğunu varsayalım:

```
alpha func();
```

Bu fonksiyonun şu ifade ile çağrıldığını varsayalım:

```
a2 = func();
```

func() tarafından döndürülen değerin bir kopyasını çıkarmak için kopyalama kurucu fonksiyon çağrılır ve bu değer, (atama operatörü kullanılarak) a2'ye atanır.

X(X) Kurucu Fonksiyonu Neden Olmasın?

Kopyalama kurucu fonksiyona argüman olarak bir referans kullanmamız gerekir mi? Bunun yerine, değer aktarabilir miyiz? Hayır, aşağıdaki ifadeyi derlemeye kalktığımızda derleyici, belleğin bu iş için yetersiz olduğunu (**out of memory**) söyler:

```
alpha(alpha a)
```

Çünkü bir argüman değer olarak aktarıldığında bunun bir kopyası oluşturulur. Peki, bu kopyayı kim hazırlar? Kopyalama kurucu fonksiyonu. Fakat kopyalama kurucu fonksiyonu zaten *bu*; yani, kendisini çağırıyor. Aslında, derleyici belleksiz kalana kadar fonksiyon kendisini üst üste çağırır. Bu nedenle, kopyalama kurucu fonksiyonda argüman, referans olarak aktarılmalıdır; bu yöntem hiç kopya oluşturmaz.

Yok Edicilere Dikkat

"Referans Olarak Aktarmak" ve "Bir Değer Döndürmek" başlıkları altında, argümanların bir fonksiyona değer olarak aktarılmasını ve fonksiyonun bir değer döndürmesini ele aldık. Bu gibi durumlar, yok edici fonksiyonun çağrılmasına da neden olur. Fonksiyon tarafından tanımlanan geçici nesnelerin ortadan kaldırılmasında ve fonksiyon sona erip döndüğünde yok edici fonksiyon çağrılır. Beklemediğiniz taktirde bu büyük ölçüde şaşkınlığa neden olabilir. Kısadan hisse: Üyelerin tek tek kopyalanmasından daha fazlasını gerektiren nesnelerle işlem yaparken, mümkün olan her durumda, argümanları değer olarak değil referans olarak aktarın ve referans olarak döndürün.

Hem Kopyalama Kurucu Fonksiyonunu Hem de Atama Operatörünü Tanımlayın

Atama operatörünü aşırı yüklediğinizde, neredeyse her seferinde kopyalama kurucu fonksiyonu da aşırı yüklemek istersiniz (ya da tam tersi). Bildik kopyalama rutinini bazı durumlarda, üyelerin tek tek kopyalanmasını diğer durumlarda kullanmayı tercih etmezsiniz. Bunlardan birini veya diğerini kullanacağınızı düşünmüyor olsanız bile, belli olmayan bazı durumlarda derleyiciyi bunları kullanırken bulabilirsiniz. Örneğin, bir argümanı bir değer olarak bir fonksiyona aktarırken veya bir fonksiyondan değer olarak dönüş yaparken. Aslında, bir sınıfın kurucu fonksiyonu eğer bellek veya disk dosyaları gibi sistem kaynaklarının kullanımını içeriyorsa, hemen hemen her zaman hem atama operatörünün hem de kurucu kopyalama fonksiyonunu aşırı yükleyebilirsiniz ve bunların istediğiniz gibi davranmalarını garanti edebilirsiniz.

Kopyalamayı Nasıl Önleyebilirsiniz?

Atama operatörünü ve kopyalama kurucu fonksiyonunu kullanarak nesneleri kopyalamayı duruma göre nasıl özelleştirdiğimizi ele aldık. Yine de, kimi zaman bir nesnenin kopyalanmasını bu işlemleri kullanarak engellemek isteyebilirsiniz? Söz gelişi, bir sınıfın üyelerinin her birinin, kurucu fonksiyona argüman olarak aktarılan belli bir üye için tek bir değer ile tanımlanması kaçınılmaz olabilir. Eğer bir nesne kopyalandıysa bu kopyaya da aynı değer verilir. Kopyalamayı önlemek için atama operatörünün ve kopyalama kurucu fonksiyonunu private üye olacak şekilde aşırı yükleyin.

```
class alpha
   {
   private:
      alpha& operator = (alpha&);   //private atama operatoru
      alpha(alpha&);                //private kopyalama kurucu fonksiyonu
   };
```

Aşağıdaki gibi bir kopyalama işlemine başladığınız anda derleyici, fonksiyonun erişilemez olduğunu size bildirecektir:

```
alpha a1, a2;
```



```
a1 = a2;        //atama
alpha a3(a1);   //kopyalama kurucu fonksiyonu
```

Fonksiyonlar hiçbir zaman çağrılmayacakları için bunları tanımlamanız gerekmez.

UML Nesne Şemaları

Sınıf şemalarının örneklerini önceki bölümlerde görmüştük. UML'in nesne şemalarını da destekliyor olduğunu öğrenmek muhtemelen sizi şaşırtmayacaktır. Nesne şemaları, spesifik nesneleri betimler (örneğin, `Professor` sınıfının `Mike_Gonzalez` nesnesi). Nesneler arası ilişkiler programın işleyişi sırasında değiştiği için (hatta, öyle ki nesneler tanımlanabilir ve yok edilebilirler) bir nesne şeması bir enstantane gibidir; zamanın belirli bir anındaki nesneleri simgeler. Nesne şemaları `statik` UML şemaları olarak bilinir.

Programınızın gerçekleştirdiği belirli bir işi modellemek için bir nesne şeması kullanabilirsiniz. Programı zaman içinde bir anlık dondurursunuz ve o anda ilgilendiğiniz davranış biçimini gösteren nesneleri ve bu nesneler arasındaki iletişimi incelersiniz.

Nesne şemalarında, nesneler tıpkı sınıf şemalarındaki sınıflar gibi dikdörtgenler ile temsil edilir. Nesnenin ismi, nitelikleri ve işlemleri de benzer şekilde gösterilir. Bununla birlikte, nesneler, sınıflardan altı çizili isimleri ile ayırt edilir. Nesne ve sınıf isimleri, iki nokta işareti ile ayrılarak, birlikte kullanılabilir:

```
anObj:aClass
```

Eğer nesnenin ismini bilmiyorsanız (örneğin, sadece işaretçi yardımıyla nesneyi tanıyorsanız), iki nokta işareti, sınıfı isminin önünde olması şartıyla sadece sınıf ismini kullanabilirsiniz:

```
:aClass
```

Nesneler arasındaki çizgiler *bağlantı* olarak adlandırılır ve başka bir nesne ile haberleşen nesneyi simgelerler. Şemada yönlendirilebilirlik (navigability) de gösterilebilir. Bir niteliğin değeri ise eşittir işareti kullanılarak gösterilebilir:

```
count = 0
```

En sonda noktalı virgül bulunmadığına dikkat edin; bu C++ değil, UML'dir.

Karşılaşacağımız bir başka UML özelliği ise *nottur*. Notlar, köpek kulağı gibi köşesi kıvrık dikdörtgenler ile simgelenir. Notlar, yorumları veya açıklamaları tutarlar. Noktalı bir çizgi bir notu şemadaki ilgili eleman ile birleştirir. Bağlantılar (associations) ve bağlardan (link) farklı olarak bir not, bir sınıf veya nesne dikdörtgeninin içindeki elemanlara erişebilir. Notlar her türlü UML şemasında kullanılabilir. Bu bölümün kalan kısmında nesne şemalarından birkaç tane daha göreceğiz.

Bellek Açısından Verimli String Sınıfı

`ASSIGN` ve `XOFXREF` örnekleri, aşırı yüklenmiş atama operatörlerine ve kopyalama kurucu fonksiyonlarına gerçekten sahip olmak zorunda değildirler. Bu örnekler, sadece bir veri öğesi olan anlaşılır sınıflar kullandıkları için varsayılan atama operatörü ve kopyalama kurucu fonksiyonu yeterli olacaktır. Şimdi, kullanıcının bu operatörleri aşırı yüklemek zorunda olduğu bir örneğe bakalım.

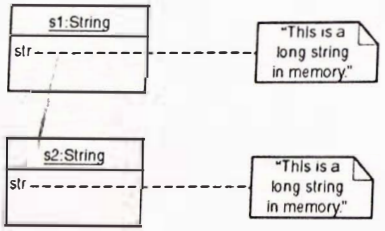
String Sınıfının Kusurları

"Ev yapımı" `String` sınıfımızın çeşitli versiyonlarını önceki bölümlerde görmüştük. Ancak, bu versiyonlar yeterince sofistike değildi. Bir `String` nesnesinin değerini bir diğerine aşağıdaki gibi atayabilmek amacıyla = operatörünü aşırı yüklemek hoş olurdu:

```
s2 = s1;
```

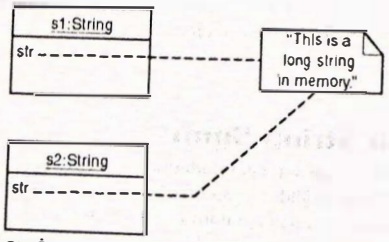
= operatörünü aşırı yüklersek, `String` sınıfının en temel veri öğesi olan esas karakter katarını (`char` tipinde dizi) nasıl ele alacağımız sorusu gündeme gelir.

Bir ihtimal, her `String` nesnesinin bir karakter katarı saklayabilecek kadar bir yere sahip olmasıdır. Bir `String` nesnesini bir diğerine atayacaksak (önceki ifadede `s1`'den `s2`'ye atanıyordu), sadece karakter katarı, kaynak nesneden hedef nesneye kopyalanır. Eğer belleği korumayı da düşünüyorsanız; bu yaklaşımdaki problem, aynı karakter katarının artık bellekte iki (veya daha fazla) ayrı yerde mevcut olmasıdır. Bu, özellikle karakter katarları uzunsa, çok da verimli değildir. Şekil 11.4'te bunun nasıl olduğunu görebilirsiniz.



ŞEKİL 11.4: UML nesne şeması: karakter katarları kopyalanıyor.

Her `String` nesnesinin kendi `char*` karakter katarını içermesini sağlamak yerine, her nesneyi sadece karakter katarına işaret eden bir işaretçi içerecek biçimde düzenleyebiliriz. Şimdi, bir `String` nesnesini diğerine atayacaksak, sadece işaretçiyi bir nesneden diğerine kopyalamamız yeterli olacaktır; her iki işaretçi de aynı karakter katarına işaret edecektir. Karakter katarının sadece tek kopyası bellekte saklanmak zorunda olduğu için, bu yaklaşım verimlidir. Şekil 11.5'te bu yaklaşım açıklanmıştır.



ŞEKİL 11.5: UML nesne şeması: karakter katarlarına işaret eden işaretçiler kopyalanıyor.

Ancak, eğer bu sistemi kullanıyorsak bir `String` nesnesini yok ederken dikkatli olmamız gerekir. Bir `String`'in yok edici fonksiyonu, karakter katarı tarafından işgal edilen belleği serbest bırakmak için `delete`'i kullanıyorsa ve karakter katarına işaret eden işaretçileri içeren birkaç tane nesne mevcutsa, bu nesneler, karakter katarının artık mevcut olmadığı belleğe işaret eden işaretçilerle kalakalacaklardır. Üstelik, işaretçilerin hala karakter katarına işaret ettiğini sanarak. Bu işaretçiler, sallanan (dangling) işaretçiler haline gelirler.

`String` nesnelerinde karakter katarlarına işaret eden işaretçiler kullanmak için, belirli bir karakter katarına kaç tane `String` nesnesinin işaret ettiğini takip etmemizi sağlayacak bir yöndar karakter katarı üzerinde `delete` operatörünü kullanmayı engelleyebiliriz. Sıradaki örneğimiz olan `STRIMEM`, tam olarak bunu yapar.

Bir String-Sayaç Sınıfı

Varsayalım, aynı karakter katarına işaret eden birkaç tane `String` nesnesine sahibiz ve bu karakter katarına işaret eden kaç tane `String` olduğunun kaydını tutmak istiyoruz. Bu toplamı nerede saklayacağız?

Her `String` nesnesi için, belirli bir karakter katarına işaret eden kaç tane benzer `String` olduğunun hesabını tutmak külfetli olabilir. Bu nedenle, bu toplam için `String`'in içinde bir üye değişken kullanmak istemiyoruz. Bir statik değişken kullanabilir miyiz? Bu bir ihtimal; bir statik dizi tanımlayabiliriz ve bunu, karakter katarı adreslerinin ve toplamlarının listesini saklamak için kullanabiliriz. Bu, yine de, önemli ölçüde bir yük getirir. Toplamı saklamak için yeni bir sınıf tanımlamak çok daha etkili olur. Bu sınıfın her nesnesi (bunu `strCount` olarak adlandıracağız) bir toplam ve karakter katarına işaret eden bir işaretçi içerir. Her `String` nesnesi ilgili `strCount` nesnesine işaret eden bir işaretçiye sahiptir. Şekil 11.6 bunun nasıl gerçeklendiğini gösteriyor.

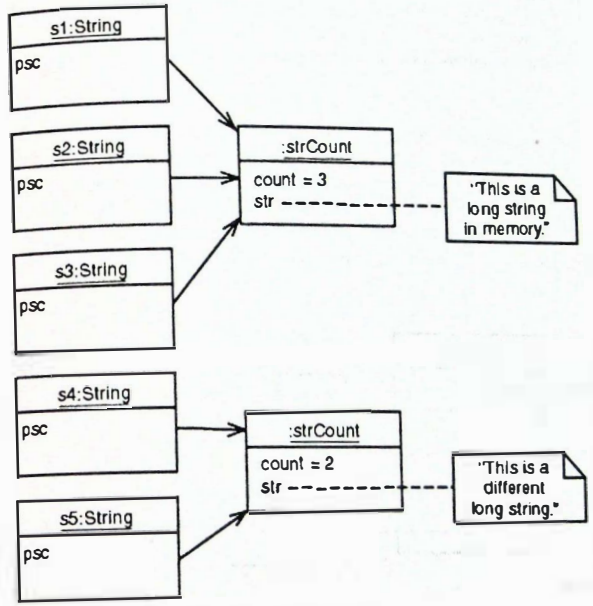
`String` nesnelerinin `strCount` nesnelerine erişimlerini garanti etmek için `String`'i, `strCount`'un bir `friend`'i yapıyoruz. Ayrıca, `strCount` sınıfının yalnızca `String` sınıfı tarafından kullanılmasını garanti etmek istiyoruz. `StrCount`'un fonksiyonlarının herhangi birine erişimi önlemek için `strCount`'un tüm üye fonksiyonlarını `private` olarak tanımlıyoruz. Bunun yanı sıra, `String` bir `friend` olduğu için `strCount`'un herhangi bir parçasına erişebilir. `STRIMEM` programının listesi şöyledir:

```
// strimem.cpp
// bellekten tasarruf saglayan String sinifi
// asiri yuklenen atama operatoru ve kopyalama kurucu fonksiyonu
#include <iostream>
#include <cstring>                          //strcpy() vs. icin
using namespace std;
////////////////////////////////////////////////////////////////
class strCount                              //benzerisiz karakter katarlarinin
   {                                        //sayisini takip eder
   private:
      int count;                            //orneklerin sayisi
      char* str;                            //karakter katarina isaret eden isaretci
      friend class String;                  //erisebilmemizi saglayalim
      //uye fonksiyonlar private
//--------------------------------------------------------------
      strCount(char* s)                     //tek argumanli kurucu fonksiyon
         {
         int length = strlen(s);            //karakter katari argumanin uzunlugu
```

```
            str = new char [length+1];    //karakter katari icin bellek al
            strcpy(str, s);               //argumani oraya kopyala
            count = 1;                    //toplami 1'den baslat
            }
   //-------------------------------------------------------------
         ~strCount()                      //yok edici fonksiyon
            { delete[] str; }             //karakter katarini ortadan kaldir
      };
   ////////////////////////////////////////////////////////////////
   class String                           //String sinifi
      {
      private:
         strCount* psc;                   //strCount'a isaret eden isaretci
      public:
         String()                         //argumansiz kurucu fonksiyon
            { psc = new strCount("NULL"); }
   //-------------------------------------------------------------
         String(char* s)                  //1 argumanli kurucu fonksiyon
            { psc = new strCount(s); }
   //-------------------------------------------------------------
         String(String& S)                //kopyalama kurucu fonksiyonu
            {
            psc = S.psc;
            (psc->count)++;
            }
   //-------------------------------------------------------------
         ~String()                        //Yok edici fonksiyon
            {
            if(psc->count==1)             //eger son kullanicisi isek,
               delete psc;                //strCount'umuzu yok et
            else                          //aksi halde,
               (psc->count)--;            //toplamini azalt
            }
   //-------------------------------------------------------------
         void display()                   //String'i goster
            {
            cout << psc->str;                          //karakter katarini yazdir
            cout << " (addr=" << psc << ")";           //adresi yazdir
            }
   //-------------------------------------------------------------
         void operator = (String&S)       //karakter katarina atama yap
            {
            if(psc->count==1)             //eger son kullanicisi isek,
               delete psc;                //strCount'umuzu yok et
            else                          //aksi halde,
               (psc->count)--;            //toplamini bir azalt
            psc = S.psc;                  //argumanin strCount'unu kullan
            (psc->count)++;               //toplamini bir artir
            }
      };
   ////////////////////////////////////////////////////////////////
   int main()
      {
      String s3 = "When the fox preaches, look to your geese.";
      cout << "\ns3="; s3.display();      //s3'u goster

      String s1;                          //String'i tanimla
      s1 = s3;                            //bir baska String'e ata
      cout << "\ns1="; s1.display();       //ekranda goster
```



```
      String s2(s3);                      //String ile ilk kullanima hazirla
      cout << "\ns2="; s2.display();       //ekranda goster
      cout << endl;
      return 0;
      }
```



ŞEKİL 11.6: **String** ve **strCount** nesneleri.

STRIMEM'in **main()** bölümünde, **s3** adında bir **String** nesnesi tanımlanır. **s3** şu atasözünü içerir: "When the fox preaches, look to your geese." Bir başka **String** daha tanımlanır (**s1**) ve **s3**'e eşitlenir. Sonra **s2**'yi tanımlanır ve **s2**'ye **s3**'ün değeri atanır. **s1**'i **s3**'e eşitlemek, aşırı yüklenmiş atama operatörünü; **s2**'ye **s3**'ün değerini atamak, aşırı yüklenmiş kopyalama kurucu fonksiyonunu çağırır. Üç karakter katarının dökümü alınır; ayrıca, bütün bu nesnelerin aynı olduğunu göstermek için, nesnelerin her birinin **psc** işaretçisi tarafından işaret edilen **strCount** nesnelerinin adresleri de listelenir. **STRIMEM** programının çıktısı şöyledir:

```
s3=When the fox preaches,look to your geese. (addr=0x8f510e00)
s1=When the fox preaches,look to your geese. (addr=0x8f510e00)
s2=When the fox preaches,look to your geese. (addr=0x8f510e00)
```

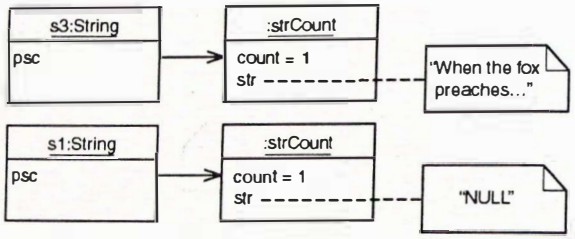
String sınıfının diğer görevleri **String** ve **strCount** sınıfları arasında paylaştırılır. Şimdi bunların yaptıklarına göz atalım.

strCount Sınıfı

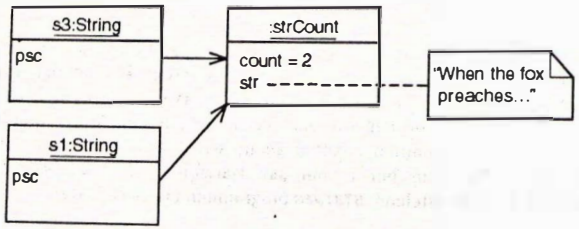
`strCount` sınıfı, esas karakter katarına işaret eden bir işaretçi içerir; ayrıca, bu karakter katarına işaret eden `String` sınıfı nesnelerinin sayısını da içerir. Sınıfta mevcut olan tek kurucu fonksiyon, bir karakter katarına işaret eden bir işaretçiyi argüman olarak alır ve bu karakter katarı için yeni bir bellek alanı oluşturur. Karakter katarını bu alana kopyalar ve karakter katarı tanımlandığında sadece bir tane `String` ona işaret ettiği için toplamı 1 olarak ayarlar. `strCount` içindeki yok edici fonksiyon, karakter katarı tarafından kullanılan belleği serbest bırakır. (Karakter katarı bir dizi olduğu için `delete`, köşeli parantezlerle kullanılır: `delete[]`.)

String Sınıfı

`String` sınıfı üç adet kurucu fonksiyon kullanır. Eğer yeni bir karakter katarı tanımlanıyorsa, sıfır argümanlı veya C karakter katarı argümanlı kurucu fonksiyonlarda olduğu gibi, karakter katarını tutması için yeni bir `strCount` nesnesi oluşturulur ve `psc` işaretçisi bu nesneyi işaret edecek şekilde ayarlanır. Eğer mevcut bir `String` nesnesi kopyalanmaktaysa, kopyalama kurucu fonksiyonunda ve aşırı yüklenmiş atama operatöründe olduğu gibi, `psc` işaretçisi eski `strCount` nesnesini işaret edecek şekilde ayarlanır ve bu nesnenin içindeki toplam (`count`) artırılır.



s1=s2; ifadesi çalıştırılmadan önce

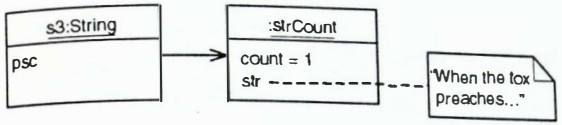


s1=s2; ifadesi çalıştırıldıktan sonra

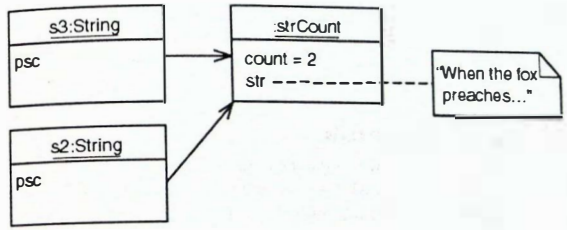
ŞEKİL 11.7: `STRIMEM` programındaki atama operatörü.

Eğer toplam 1 ise, aşırı yüklenmiş atama operatörünün ve de yok edici fonksiyonun, `psc` tarafından işaret edilen eski `strCount` nesnesini yok etmeleri gerekir. (Sadece tek bir

`strCount` nesnesini ortadan kaldırdığımız için `delete`'in yanında köşeli parantez kullanmamıza gerek yoktur.) Atama operatörü neden yok etme işleminden sorumlu olmalıdır? Hatırlarsanız, eşittir işaretinin sol tarafındaki `String` nesnesi (buna `s1` diyelim) atama işleminden önce bir `strCount` nesnesine (buna `oldStrCnt` diyelim) işaret ediyordu. Atama işleminden sonra `s1`, `String` nesnesi kalmadıysa, `oldStrCnt` yok edilmelidir. Eğer buna işaret eden başka nesneler mevcutsa, bu nesnenin toplamı azaltılmalıdır. Şekil 11.7 aşırı yüklenmiş atama operatörünün işleyişini; Şekil 11.8 ise kopyalama kurucu fonksiyonunu gösterir.



String s2 (s3); ifadesi çalıştırılmadan önce



String s2 (s3); ifadesi çalıştırıldıktan sonra

ŞEKİL 11.8: `STRIMEM` programındaki kopyalama kurucu fonksiyonu.

this İşaretçisi

Nesnelerin üye fonksiyonları, nesnenin kendisine işaret eden `this` adında bir tür sihirli bir işaretçiye erişim hakkına sahiptir. Herhangi bir üye fonksiyon, kendisinin üyesi olduğu nesnenin adresini böylelikle bulabilir. Aşağıda, bu mekanizmayı gösteren, kısa bir örnek olan `WHERE` görülüyor.

```
// where.cpp
// this isaretcisi
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class where
   {
   private:
      char charray[10];              //10 byte yer isgal eder
   public:
      void reveal()
         {cout << "\nMy object's address is " << this;}
   };
```

```
//////////////////////////////////////////////////////////////
int main()
   {
   where w1, w2, w3;                  //uc nesne tanimla
   w1.reveal();                       //nerede olduklarina bak
   w2.reveal();
   w3.reveal();
   cout << endl;
   return 0;
   }
```

Bu örnekteki `main()` program `where` tipinde üç nesne tanımlar. Program sonra, `reveal()` üye fonksiyonunu kullanarak nesnelerin her birinden adreslerini ekrana yazdırmalarını ister. Bu fonksiyon, `this` işaretçisinin değerini ekrana yazdırır. Programın çıktısı şöyledir:

```
My object's address is 0x8f4effec
My object's address is 0x8f4effe2
My object's address is 0x8f4effd8
```

Her nesnenin içindeki veriler 10 byte'lık birer dizi içerdiği için nesneler bellekte 10 byte ara ile yer alırlar. (`EC` eksi `E2`, ondalık tabanda 10'a karşılık gelir; `E2` eksi `D8` de aynı sonucu verir.) Bazı derleyiciler nesnelere fazladan byte'lar ilave edebilirler; böylece dizilerin 10 byte'tan biraz daha büyük olmasına neden olabilirler.

Üye Verilere this ile Erişmek

Bir üye fonksiyonu çağırdığınız zaman, fonksiyonu çağıran nesnenin adresinin `this`'e atanması sayesinde fonksiyon hayat bulur. `this` işaretçisi, bir nesneye işaret eden herhangi bir işaretçi gibi ele alınabilir; böylece, işaret ettiği nesnenin verilerine erişmek için kullanılabilir. `DOTHIS` programında olduğu gibi:

```
// dothis.cpp
// veriye isaret eden this isaretcisi
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////////
class what
   {
   private:
      int alpha;
   public:
      void tester()
      {
      this->alpha = 11;          //alpha = 11; ile ayni
      cout << this->alpha;       //cout << alpha; ile ayni
      }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   what w;
   w.tester();
   cout << endl;
   return 0;
   }
```


Bu program sadece 11 değerini ekrana yazar. Dikkat ederseniz, `tester()` üye fonksiyonu, `alpha` değişkenine şöyle erişir:

```
this->alpha
```

Bu, `alpha`'ya doğrudan erişmekle tamamen aynıdır. Bu söz dizimi hatasız çalışır, fakat `this`'in gerçekten bir nesneye işaret ettiğini göstermekten başka bunu kullanmanın hiçbir sebebi yoktur.

Değer Döndürmek için this Kullanmak

`this`'in çok daha pratik bir kullanımı, üye fonksiyonlardan ve aşırı yüklenmiş operatörlerden değer döndürürken karşımıza çıkıyor.

`ASSIGN` programında bir nesneyi referans olarak döndüremediğimizi hatırlayın; çünkü, nesne kendisini döndüren fonksiyonun yerel değişkeniydi ve bu nedenle, fonksiyon dönerken bu nesne yok edilmişti. Eğer referans olarak döndüreceksek daha kalıcı bir nesneye ihtiyacımız vardır. Bir fonksiyonun üyesi olduğu bir nesne, kendi üye fonksiyonlarından çok daha kalıcıdır. Bir nesnenin üye fonksiyonları, çağrıldıkları her an oluşturulabilir ve yok edilebilir, fakat nesnenin kendisi, dışarıdan bir etki gelip nesneyi yok edene kadar (örneğin, nesne üzerine `delete` operatörü uygulandığında) varlığını sürdürür. Bu nedenle, bir fonksiyonun üyesi olduğu bir nesneyi referans olarak döndürmek, bir üye fonksiyon içinde tanımlanan geçici bir nesneyi döndürmekten çok daha iyidir. `this` işaretçisi bu işlemi kolaylaştırır.

İşte `ASSIGN2`'nin listesi. Bu programda `operator=()` fonksiyonu, kendisini çağıran nesneyi referans olarak döndürür:

```
// assign2.cpp
// this isaretcisinin icerigini dondurur
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////////
class alpha
   {
   private:
      int data;
   public:
      alpha()                          //argumansiz kurucu fonk.
         { }
      alpha(int d)                     // tek argumanli kurucu fonk.
         { data = d; }
      void display()                   //verileri goster
         { cout << data; }
      alpha& operator = (alpha& a)     //asiri yuklenmis = operatoru
         {
         data = a.data;                //otomatik olarak yapilmaz
         cout << "\nAssignment operator invoked";
         return *this;                 //bu alpha'nin bir kopyasini dondur
         }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   alpha a1(37);
   alpha a2, a3;
```

```
a3 = a2 = a1;                    //asiri yuklenen = operatorunu 2 kez cagir
cout << "\na2="; a2.display();   //a2'yi ekranda goster
cout << "\na3="; a3.display();   //a3'u ekranda goster
cout << endl;
return 0;
}
```

Bu programda referans olarak dönen şu deklarasyonu kullanabiliriz:

```
alpha& operator = (alpha& a)
```

Aşağıda ise değer olarak dönen deklarasyon gösteriliyor.

```
alpha operator = (alpha& a)
```

Programda alttaki deklarasyonun yerine üstteki kullanılır. Bu fonksiyondaki en son ifade şudur:

```
return *this;
```

this, fonksiyonun üye olduğu nesneye işaret ettiği için ***this**, bu nesnenin kendisidir; yani, bu ifade nesnenin kendisini referans olarak döndürmektedir. **ASSIGN2**'nin çıktısı şöyledir:

```
Assignment operator invoked
Assignment operator invoked
a2=37
a3=37
```

Aşağıdaki ifadede, eşittir işaretine her rastlayışımızda aşırı yüklenmiş **operator=()** fonksiyonu çağrılır.

```
a3 = a2 = a1;
```

Bu fonksiyon mesajları ekrana yazar. Nesnelerin üçü de sonunda aynı değere sahip olmuş olur.

Fazladan nesne tanımlanmasını önlemek için genellikle aşırı yüklenmiş atama operatörlerinden, ***this** kullanarak, referans yoluyla değer döndürmek istersiniz.

Gözden Geçirilmiş STRIMEM Programı

this işaretçisini kullanarak **STRIMEM** programındaki **operator=()** fonksiyonunu, referans yoluyla bir değer döndürmesi için yeniden gözden geçireceğiz. Böylece, **String** nesneleri için birden fazla atama operatörünü bir arada kullanmak mümkün olabilecek. Örneğin,

```
s1 = s2 = s3;
```

Aynı zamanda, yapay nesnelerin (mesela, nesneler değer olarak döndürülürken tanımlananlar) oluşturulmasını da önleyebileceğiz. **STRIMEM2** programının listesi şöyledir:

```
// strimem2.cpp
// bellekten tasarruf saglayan String sinifi
```



```
// asiri yuklenen atama operatoru ile this isaretcisi
#include <iostream>
#include <cstring>                          //strcpy() vs. icin
using namespace std;
////////////////////////////////////////////////////////////////
class strCount                              //benzersiz karakter katarlarinin
   {                                        //sayisini takip eder
   private:
      int count;                            //ornek sayisi
      char* str;                            //karakter katarina isaret eden isaretci
      friend class String;                  //erisebilmemizi saglayalim
   //uye fonksiyonlar private
      strCount(char* s)                     //tek argumanli kurucu fonk.
         {
         int length = strlen(s);            //karakter katari argumaninin uzunlugu
         str = new char [length+1];         //karakter katari icin bellek al
         strcpy(str, s);                    //argumani oraya kopyala
         count = 1;                         //sayaci 1'den baslat
         }
//--------------------------------------------------------------
      ~strCount()                           //yok edici fonk.
         { delete[] str; }                  //karakter katarini yok et
   };
////////////////////////////////////////////////////////////////
class String                                //String sinifi
   {
   private:
      strCount* psc;                        //strCount'a isaret eden isaretci
   public:
      String()                              //argumansiz kurucu fonk.
         { psc = new strCount("NULL"); }
//--------------------------------------------------------------
      String(char* s)                       //1 argumanli kurucu fonk.
         { psc = new strCount(s); }
//--------------------------------------------------------------
      String(String& S)                     //kopyalama kurucu fonk.
         {
         cout << "\nCOPY CONSTRUCTOR";
         psc = S.psc;
            (psc->count)++;
         }
//--------------------------------------------------------------
      ~String()                             //yok edici fonk.
         {
         if(psc->count==1)                  //eger son kullanicisi isek,
            delete psc;                     //strCount'umuzu yok et
         else                               //aksi halde,
            (psc->count)--;                 //toplamini bir azalt
         }
//--------------------------------------------------------------
      void display()                        //String'i goster
         {
         cout << psc->str;                      //karakter katarini yazdir
         cout << " (addr=" << psc << ")";       //adresi yazdir
         }
//--------------------------------------------------------------
      String& operator = (String& S)        //karakter katarina deger ata
         {
         cout << "\nASSIGNMENT";
```

```
        if(psc->count==1)              //eger son kullanicisi isek,
           delete psc;                 //strCount'umuzu yok et
        else                           //aksi halde,
           (psc->count)--;             //toplamini bir azalt
        psc = S.psc;                   //argumanin strCount'unu kullan
        (psc->count)++;                //toplami bir artir
        return *this;                  //bu nesneyi dondur
        }
    };

int main()
   {
   String s3 = "When the fox preaches, look to your geese.";
   cout << "\ns3="; s3.display();      //s3'u ekranda goster

   String s1, s2;                      //String'leri tanimla
   s1 = s2 = s3;                       //String'lere deger ata
   cout << "\ns1="; s1.display();      //ekranda goster
   cout << "\ns2="; s2.display();      //ekranda goster
   cout << endl;                       //tusa basilmasini bekle
   return 0;
   }
```

= operatörünün deklaratörü artık şöyledir:

```
String& operator = (String& S)         //referans olarak dondur
```

Ayrıca, **ASSIGN2**'de olduğu gibi, bu fonksiyon `this`'e işaret eden bir işaretçi döndürür. Programın çıktısı şöyledir:

```
s3 = When the fox preaches, look to your geese. (addr=0x8f640d3a)
ASSIGNMENT
ASSIGNMENT
s1 = When the fox preaches, look to your geese. (addr=0x8f640d3a)
s2 = When the fox preaches, look to your geese. (addr=0x8f640d3a)
```

Program çıktısı, atama ifadelerinin peşinden `String` nesnelerinin üçünün de aynı `strCount` nesnesine işaret ettiklerini gösterir.

Statik üye fonksiyonları belirli bir nesne ile bağlantılı olmadıkları için `this` işaretçisinin statik üye fonksiyonlarında mevcut olmadığını not etmeliyiz.

Kendi Kendine Atamalara Dikkat Edin

Murphy'nin yasalarından biri; "bir şey mümkünse, biri eninde sonunda başarır" der. Bu, programlama için kesinlikle doğrudur. Bu nedenle, eğer = operatörünü aşırı yüklediyseniz, birisi bunu, bir nesneyi kendisine eşitlemek için kullanacaktır:

```
alpha = alpha;
```

Aşırı yüklenmiş operatörünüz, bu tür kendi kendine atamaları ele alabilecek şekilde hazırlanmalıdır. Aksi halde, kötü şeyler meydana gelebilir. Örneğin, **STRIMEM2** programının `main()` bölümünde, eğer bir `String` nesnesini kendisine eşitlerseniz, (aynı `strCount` nesnesini kullanan başka `String` nesneleri yoksa) program çökecektir. Buradaki problem şudur: Eğer

atama operatörü kendisini çağıran nesnenin `strCount`'u kullanan tek nesne olduğunu düşünürse `strCount` nesnesini yok eder. Hiçbir şeyin yok edilmesi gerekmiyor olsa bile, kendi kendine atama ifadesi operatörün bu şekilde inanmasına neden olacaktır.

Bu problemi çözmek için, aşırı yüklenmiş her türlü atama operatörünün en başında kendi kendine atama olup olmadığını kontrol etmelisiniz. Bu işlemi birçok durumda operatörü çağıran nesnenin adresini, operatörün argümanının adresiyle karşılaştırarak gerçekleştirebilirsiniz. Adresler aynıysa, nesneler türdeş demektir; fonksiyondan hemen dönmelisiniz. (Birini diğerine atamanıza gerek yoktur; her ikisi de neredeyse aynıdır.) Örneğin, **STRIMEM2**'de `operator=()` fonksiyonunun başına şu satırları ekleyebilirsiniz:

```
if(this == &S)
   return *this;
```

Bu satırlar problemi çözecektir.

Dinamik Tip Bilgisi

Bir nesnenin sınıfı hakkında bilgi toplamak mümkündür; hatta çalışma zamanında (runtime) nesnenin sınıfını değiştirmek bile mümkündür. Kısaca iki mekanizmaya göz atacağız: `dynamic_cast` operatörü ve `typeid` operatörü. Bunlar ileri düzeyde becerilerdir, fakat bir gün işinize yarayabilirler.

Bu beceriler genellikle bir temel sınıftan çeşitli değişik sınıflar (kimi zaman karmaşık yöntemlerle) türetilirken kullanılırlar. Dinamik tip atamalarının (dynamic casts) çalışması için temel sınıfın çok biçimli olması gerekir; yani, temel sınıfın en azından bir tane sanal fonksiyonu olmalıdır.

`dynamic_cast` ve `typeid`'nin her ikisinin de çalışması için derleyicinizin Run Time Type Information (RTTI - Çalışma Zamanı Tip Bilgisi) özelliğinin aktif olması gerekir. Borland C++ Builder bu beceriyi varsayılan durumda aktif konumda içerir, ancak Microsoft Visual C++'ta bunu açıkça sizin aktif konuma getirmeniz gerekir. Bunun nasıl yapılabileceği ile ilgili ayrıntılar için "Microsoft Visual C++" adlı Ek C'ye bakın. Ayrıca, `TYPEINFO` başlık dosyasını da programa dahil etmeniz gerekecek.

dynamic_cast Yardımıyla Bir Sınıfın Tipini Kontrol Etmek

Diyelim ki, başka bir program sizin programınıza bir nesne gönderiyor (işletim sistemi "callback" fonksiyonu ile bunu gerçekleştirebilir). Bu nesnenin belirli bir tipte olması beklenir, fakat siz yine de emin olmak için nesnenin tipini kontrol etmek isteyebilirsiniz. Bir nesne belirli bir tipte ise bunu nasıl bildirebilirsiniz? Nesnelerini kontrol etmek istediğiniz sınıfların aynı ortak sınıftan türetildikleri varsayımıyla `dynamic_cast` operatörü bunun için bir yöntem önerir. **DYNCAST1** programında bunun nasıl gerçekleştirilebileceği gösterilmiştir.

```
// dyncast1.cpp
// nesnenin tipini kontrol etmek icin dinamik tip ataması kullaniliyor
// derleyicinin RTTI ozelligi aktif konumda olmalidir
#include <iostream>
#include <typeinfo>                    //dynamic_cast icin
using namespace std;
```

```
//////////////////////////////////////////////////////////////
class Base
   {
   virtual void vertFunc()            //dinamik tip atamasi icin gerekli
      { }
   };
class Derv1 : public Base
   { };
class Derv2 : public Base
   { };
//////////////////////////////////////////////////////////////
//pUnknown'un bir Derv1 sinifina isaret edip etmedigini kontrol eder
bool isDerv1(Base* pUnknown)          //Base'in bilinmeyen bir alt sinifi
   {
   Derv1* pDerv1;
   if( pDerv1 = dynamic_cast<Derv1*>(pUnknown) )
      return true;
   else
      return false;
   }
//--------------------------------------------------------------
int main()
   {
   Derv1* d1 = new Derv1;
   Derv2* d2 = new Derv2;

   if( isDerv1(d1) )
      cout << "d1 is a member of the Derv1 class\n";
   else
      cout << "d1 is not a member of the Derv1 class\n";

   if( isDerv1(d2) )
      cout << "d2 is a member of the Derv1 class\n";
   else
      cout << "d2 is not a member of the Derv1 class\n";
   return 0;
   }
```

Burada, **Base** adında bir temel sınıfa; **Derv1** ve **Derv2** adında iki adet de türetilmiş sınıfa sahibiz. Bir de, **isDerv1()** adında bir fonksiyon mevcuttur. **isDerv1()** fonksiyonu, argüman olarak aldığı işaretçi **Derv1** sınıfına ait bir nesneye işaret ediyorsa **true** döndürür. Fonksiyonun argümanı **Base** sınıfı tipindedir. Bu nedenle, fonksiyona aktarılan nesne ya **Derv1** ya da **Derv2** olabilir. **dynamic_cast** operatörü, bu bilinmeyen **pUnknown** işaretçisini **Derv1** tipine dönüştürmeye çalışır. Eğer sonuç sıfır değilse, **pUnknown Derv1** nesnesine işaret eder hale geldi demektir. Eğer sonuç sıfır ise, **pUnknown** başka bir şeye ediyordur.

dynamic_cast Yardımıyla İşaretçi Tiplerini Değiştirmek

dynamic_cast operatörü, kalıtım ağacında yukarıya veya aşağıya doğru tip dönüşümü yapmanıza imkan verir. Yine de, bu tür bir dönüşüme sadece belirli sınırlar içinde müsaade eder. **DYNCAST2** programı bu tür dönüşümlerin örneklerini gösterir.

```
// dyncast2.cpp
// dinamik tip atamalarini test eder
```



```
// derleyicinin RTTI ozelligi aktif konumda olmalidir
#include <iostream>
#include <typeinfo>
using namespace std;                  //dynamic_cast icin
//////////////////////////////////////////////////////////////
class Base
   {
   protected:
      int ba;
   public:
      Base() : ba(0)
         { }
      Base(int b) : ba(b)
         { }
      virtual void vertFunc()         //dynamic_cast icin gerekli
         { }
      void show()
         { cout << "Base: ba=" << ba << endl;}
   };
//////////////////////////////////////////////////////////////
class Derv : public Base
   {
   private:
      int da;
   public:
      Derv(int b, int d) : da(d)
         { ba = b; }
      void show()
         { cout << "Derv: ba=" << ba << ", da=" << da << endl;}
   };
//////////////////////////////////////////////////////////////
int main()
   {
   Base* pBase = new Base(10);        //Base'e isaret eden isaretci
   Derv* pDerv = new Derv(21, 22);    //Derv'e isaret eden isaretci

   //turetilmisten temele: yukarı donusum-Derv'in Base alt nesnesine isaret eder
   pBase = dynamic_cast<Base*>(pDerv);
   pBase->show();                     //"Base: ba=21"

   pBase = new Derv(31, 32);          //normal
   //temelden turetilmise: asagi donusum-(pBase, bir Derv'e isaret etmelidir)
   pDerv = dynamic_cast<Derv*>(pBase);
   pDerv->show();                     //"Derv: ba=31, da=32"
   return 0;
   }
```

Burada bir temel sınıfa, bir de türetilmiş sınıfa sahibiz. Dinamik tip atamalarının etkilerini daha iyi göstermek için bu sınıfların her birine birer veri öğesi yerleştirdik.

Yukarı yönlü bir dönüşümde, bir türetilmiş sınıf nesnesi, bir temel sınıf nesnesine dönüştürülmeye çalışılır. Elde edilen, türetilmiş sınıf nesnesinin temel parçası olur. Örnekte, **Derv** sınıfına ait bir nesne tanımlanır. Bu nesnenin temel sınıf parçası **ba** üye verisini tutar. **ba**, 21 değerine sahiptir. Nesnenin türetilmiş parçası ise 22 değerini içeren **da** üye verisini tutar. Dönüşümden sonra, **pBase** bu **Derv** sınıfına ait nesnenin temel sınıf parçasına işaret eder. Bu nedenle, kendi değerini göstermek için çağrıldığında **Base: ba=21** çıktısını ekrana yazar. Eğer tüm istediğiniz bir nesnenin temel parçası ise yukarı yönlü dönüşümler işinize yarayacaktır.

Aşağı yönlü bir dönüşümde ise, bir temel sınıf işaretçisi tarafından işaret edilen bir türetilmiş sınıf nesnesi, bir türetilmiş sınıf işaretçisine yerleştirilir.

typeid Operatörü

Kimi zaman, bir nesnenin belirli bir sınıfa ait olduğunu basitçe doğrulamaktan ziyade, nesne hakkında daha fazla bilgiye sahip olmak istersiniz. Bilinmeyen bir nesnenin tipiyle ilgili bilgileri, mesela nesnenin ait olduğu sınıfın ismi gibi, `typeid` operatörünü kullanarak elde edebilirsiniz. `TYPEID` programı bunun nasıl gerçekleştirildiğini gösteriyor.

```
// typeid.cpp
// typeid()fonksiyonu
// derleyicinin RTTI ozelliginin aktif olmasi gerekir
#include <iostream>
#include <typeinfo>                    //typeid() icin
using namespace std;
//////////////////////////////////////////////////////////////
class Base
   {
   virtual void virtFunc()             //typeid icin gerekli
      { }
   };
class Derv1 : public Base
   { };
class Derv2 : public Base
   { };
//////////////////////////////////////////////////////////////
void displayName(Base* pB)
   {
   cout << "pointer to an object of "; //sinifin ismini goster
   cout << typeid(*pB).name() << endl; //pB tarafindan isaret edilir
   }
//-------------------------------------------------------------
int main()
   {
   Base* pBase = new Derv1;
   displayName(pBase);                 //"pointer to an object of class Derv1"

   pBase = new Derv2;
   displayName(pBase);                 //"pointer to an object of class Derv2"
   return 0;
   }
```

Bu örnekte, `displayName()` fonksiyonu, kendisine aktarılan nesnenin sınıfının ismini ekranda gösterir. Bunu gerçekleştirmek için fonksiyon, `typeid` operatörüyle birlikte `type_info` sınıfının `name` üyesini kullanır. `main()`'de, bu fonksiyona sırayla, `Derv1` ve `Derv2` sınıflarına ait iki nesne aktarılır. Programın çıktısı şöyle görünür:

```
pointer to an object of class Derv1
pointer to an object of class Derv2
```

`typeid` kullanarak, bir sınıfın isminin yanı sıra, sınıfla ilgili diğer bilgiler de elde edilebilir. Örneğin, aşırı yüklenmiş `==` operatörünü kullanarak sınıfların eşitliğini kontrol edebilirsiniz. Bunun bir örneğini "Akışlar ve Dosyalar" adlı Bölüm 12'de `EMPL_IO` programında göstereceğiz. Bu

bölümdeki örneklerde işaretçiler kullanılmış olsa da, `dynamic_cast` ve `typeid`, referanslarla da aynı derecede iyi çalışırlar.

Özet

Sanal fonksiyonlar, bir programın çalışırken hangi fonksiyonu çağıracağına karar vermesini sağlayan yöntemlerden biridir. Genellikle bu tür kararlar derleme sırasında verilir. Sanal fonksiyonlar, farklı tür nesneler üzerinde aynı tür faaliyetlerde bulunurken daha büyük bir esneklik sağlarlar. Sanal fonksiyonlar özellikle, çeşitli türetilmiş sınıflara işaret eden işaretçileri tutan bir (temel sınıfa işaret eden işaretçi tipinde) bir dizi tarafından çağrılan fonksiyonların kullanımına imkan verirler. Bu, çok biçimliliğe bir örnektir. Çoğunlukla bir fonksiyon temel sınıf içinde sanal olarak bildirilir; aynı isimli diğer sınıflar ise türetilmiş sınıflar içinde bildirilirler.

Bir sınıfın içinde bir veya daha fazla sanal fonksiyon kullanılarak sınıf *soyut* hale getirilir. Sınıfın soyut olması, bu sınıftan hiç nesne örneklenmeyecek demektir.

Bir `friend` fonksiyon, bir sınıfın üye fonksiyonu olmasa bile bu sınıfın private verilerine erişebilir. Bu özellik, bir fonksiyonun, birbiriyle alakasız iki veya daha fazla sınıfa erişmesi gerektiğinde yararlıdır. Ayrıca, aşırı yüklenmiş bir operatör, sol tarafında, kendisinin üyesi olduğu sınıfın haricinde, bir sınıfın değerini kullanmak zorundaysa yine bu özellik işe yarar. `friend`'ler fonksiyonel notasyonu kolaylaştırmak için de kullanılır.

Bir `static` fonksiyon, bir sınıfın nesneleri yerine, genel olarak sınıf üzerinde işlem yapan bir fonksiyondur. Bir `static` fonksiyon özellikle statik değişkenler üzerinde işlem yapar. Sınıf ismi ve kapsam çözünürlük operatörü ile birlikte kullanılır.

Atama operatörü (=) aşırı yüklenebilir. Bu, atama operatörü bir nesnenin içeriğini bir başka nesneye kopyalamaktan başka, daha fazla işler yapmak zorunda olduğunda kaçınılmazdır. Kopyalama kurucu fonksiyonu da aşırı yüklenebilir. Kopyalama kurucu fonksiyonu, nesneleri ilk kullanıma hazırlarken, ayrıca argümanlar değer olarak aktarılırken ve döndürülürken nesnelerin kopyalarını çıkarır. Bir fonksiyon, sadece bir nesnenin kopyasını çıkarmaktan daha fazlasını yapmak zorunda ise, kopyalama kurucu fonksiyonunun aşırı yüklenmesi gereklidir.

`this` işaretçisi, bir üye fonksiyon içinde önceden tanımlıdır ve bu üye fonksiyonun ait olduğu nesneye işaret eder. Fonksiyon, üyesi olduğu nesneyi döndürürken `this` işaretçisi işe yarar.

`dynamic_cast` operatörü birkaç görev üstlenir. `dynamic_cast` operatörü, bir işaretçinin işaret ettiği nesnenin hangi tipte olduğunu belirlemek için kullanılabilir. Ayrıca, bazı durumlarda işaretçinin tipini de değiştirebilir. `typeid` operatörü bir nesnenin sınıfı hakkında, mesela sınıfın ismi, belli bilgileri ortaya çıkarabilir.

UML nesne şemaları, bir programın işleyişi esnasında belirli bir anda, bir grup nesne arasındaki ilişkileri gösterir.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Sanal fonksiyonlar aşağıdakilerden hangisine (hangilerine) imkan verir?
   a. Türetilmiş sınıflara işaret eden işaretçileri tutan, temel sınıfa işaret eden işaretçi tipinde bir dizi oluşturmaya.
   b. Hiçbir zaman erişilemeyecek fonksiyonlar tanımlamaya.
   c. Farklı sınıfların nesnelerini, aynı fonksiyon koduyla erişilebilecek şekilde gruplamaya.

d. Farklı sınıflara ait nesnelerin üye fonksiyonlarını çalıştırmak için aynı fonksiyon çağrısını kullanmaya.

2. Doğru/Yanlış: Temel sınıfa işaret eden bir işaretçi, bir türetilmiş sınıfın nesnelerine işaret edebilir.
3. Bir temel sınıfın nesnelerini işaret eden bir `p` işaretçisi varsa; bu işaretçi, bir türetilmiş sınıfın bir nesnesinin adresini içeriyorsa ve her iki sınıf, sanal olmayan `ding()` isminde bir üye fonksiyona sahipse, `p->ding();` ifadesi, `ding()` fonksiyonunun _____ sınıftaki versiyonunun çalışmasına neden olur.
4. `void` tipini döndüren ve `int` tipinde bir argüman alan `dang()` isminde bir sanal fonksiyon için bir deklaratör yazın.
5. Belirli bir fonksiyon çağrısı ifadesi ile hangi fonksiyonların çalıştırılacağına program çalışmaya başladıktan sonra karar vermek _____ olarak adlandırılır.
6. Bir temel sınıfın nesnelerini işaret eden bir `p` işaretçisi varsa; bu işaretçi, bir türetilmiş sınıfın bir nesnesinin adresini içeriyorsa ve her iki sınıf, `ding()` isminde sanal bir üye fonksiyona sahipse, `p->ding();` ifadesi, `ding()` fonksiyonunun _____ sınıftaki versiyonunun çalışmasına neden olur.
7. Bir değer döndürmeyen ve hiç argüman alamayan `aragorn` isminde bir saf sanal fonksiyon için deklarasyon yazın.
8. Bir saf sanal fonksiyon, aşağıdaki özelliklerden hangisine (hangilerine) sahip bir sanal fonksiyondur?
   a. Ait olduğu sınıfın soyut olmasına neden olur.
   b. Hiçbir şey döndürmez.
   c. Bir temel sınıf içinde kullanılır.
   d. Hiç argüman almaz.
9. `dong` sınıfının nesnelerine işaret eden işaretçilerden 10 adet içeren `parr` adında bir dizi için bir tanım yazın.
10. Bir soyut sınıf ne zaman işe yarar?
    a. Bu sınıftan hiç sınıf türetilmediğinde.
    b. Bir türetilmiş sınıftan diğerine birden fazla erişim söz konusu olduğunda.
    c. Bu sınıftan hiç nesne örneklenmediğinde.
    d. Sınıfın deklarasyonunu ertelemek istediğinizde.
11. Doğru/Yanlış: Bir `friend` fonksiyon, bir sınıfın private verilerine, sınıfın üyesi olmadan erişebilir.
12. Bir `friend` fonksiyon hangi amaçla kullanılabilir?
    a. Sınıflar arası argüman alışverişi sağlamak için.
    b. Kaynak kodu mevcut olmayan sınıflara erişim imkan vermek için.
    c. Alakasız bir sınıfa erişim imkanı vermek için.
    d. Aşırı yüklenmiş bir operatörün çeşitliliğini artırmak için.
13. `void` tipini döndüren ve `george` sınıfına ait bir argüman alan `harry()` adında bir `friend` fonksiyon için deklarasyon yazın.
14. `friend` anahtar sözcüğü aşağıdakilerin hangisinde yer alır?
    a. Bir başka sınıfa erişim izni veren bir sınıf içinde.
    b. Bir başka sınıfa erişmek isteyen bir sınıf içinde.
    c. Bir sınıfın `private` bölümünde.
    d. Bir sınıfın `public` bölümünde.

15. Bulunduğu sınıf içinde, `harry` sınıfının üyelerinin tümünü bir `friend` fonksiyon yapan bir deklarasyon yazın.
16. Bir statik fonksiyon
    a. bir nesne ortadan kaldırılacağı zaman çağrılmalıdır.
    b. bir sınıfın tek bir nesnesi ile yakından ilgilidir.
    c. sınıf ismi ve fonksiyon ismi kullanılarak çağrılabilir.
    d. yapay bir nesne oluşturulması gerektiğinde kullanılır.
17. Varsayılan atama operatörü (=) nesnelere uygulanırsa neler yapar, açıklayın.
18. `zeta` sınıfı içinde, aşırı yüklenmiş bir atama operatörü için bir deklarasyon yazın.
19. Bir atama operatörü neden aşırı yüklenir?
    a. Türdeş nesnelerin sayısını takip etmeye yardımcı olmak için.
    b. Her nesneye ayrı bir kimlik numarası atamak için.
    c. Tüm üye verilerin tam olarak kopyalandığını garanti etmek için.
    d. Atama gerçekleştiğinde haber vermek için.
20. Doğru/Yanlış: Kullanıcı, kopyalama kurucu fonksiyonun işleyişini daima tanımlamak zorundadır.
21. Atama operatörünün işleyişi ile kopyalama kurucu fonksiyonunun işleyişi,
    a. aynıdır; bir farkla, kopyalama kurucu fonksiyonu yeni bir nesne tanımlar.
    b. aynıdır; bir farkla, atama operatörü üye verileri kopyalar.
    c. her ikisinin de yeni bir nesne tanımlıyor olması haricinde farklıdır.
    d. her ikisinin de üye verileri kopyalamaları haricinde farklıdır.
22. `Bertha` adında bir sınıfın kopyalama kurucu fonksiyonu için deklarasyon yazın.
23. Doğru/Yanlış: Bir kopyalama kurucu fonksiyonu, bir nesnenin verilerinin sadece bir bölümünü kopyalamak için tanımlanabilir.
24. Bir değişkenin ömrü için aşağıdakilerden hangisi doğrudur?
    a. Değişken bir üye fonksiyona yerel ise, fonksiyonun ömrü ile örtüşür.
    b. Değişken global ise, sınıfın ömrü ile örtüşür.
    c. Değişken bir nesnenin statik olmayan bir verisi ise, nesnenin ömrü ile örtüşür.
    d. Değişken bir üye fonksiyon içinde statik ise, fonksiyonun ömrü ile örtüşür.
25. Doğru/Yanlış: Bir üye fonksiyon, içinde yerel olarak tanımlanmış bir değişkenin değerini döndürürken, değer olarak döndürüldüğü sürece bir problem olmaz.
26. Aşağıdaki iki ifade arasındaki işleyiş farkını açıklayın:

```
person p1(p0);
person p1 = p0;
```

27. Bir kopyalama kurucu fonksiyonu ne zaman çağrılır?
    a. Bir fonksiyon değer olarak döndüğünde.
    b. Bir argüman değer olarak aktarıldığında.
    c. Bir fonksiyon referans olarak döndüğünde.
    d. Bir argüman referans olarak aktarıldığında.
28. `this` işaretçisi neye işaret eder?
29. Eğer, bir sınıf içinde, `da` bir üye değişken ise, `this.da = 37;` ifadesi `da`'ya 37 değerini atayacak mıdır?

30. Bir üye fonksiyonun, üyesi olduğu nesnenin tümünü döndürmek amacıyla, geçici nesneler oluşturmadan kullanabileceği bir ifade yazın.
31. Bir nesne şemasındaki nesne dikdörtgeni neyi simgeler?
    a. Genel bir nesne grubunu.
    b. Bir sınıfı.
    c. Bir sınıfın bir örneğini.
    d. Bir sınıfın nesnelerinin tümünü.
32. Bir UML nesne şemasında, nesneler arasındaki çizgilere _____ denir.
33. Doğru/Yanlış: `A` nesnesi zaman içinde belli bir anda `B` nesnesi ile ilişkili olabilir, başka bir anda olmayabilir.
34. Nesne şemaları neyi gösterir?
    a. Zaman içinde bir anda hangi nesnelerin mevcut olduğunu.
    b. Zaman içinde bir anda hangi nesnelerin haberleştiğini.
    c. Programın belirli bir davranışı içinde hangi nesnelerin yer aldığını.
    d. Diğer sınıfların nesnelerini çağıran işlemlere (üye fonksiyonlara) hangi nesnelerin sahip olduğunu.

Alıştırmalar

Yıldızla işaretli soruların cevaplarını Ek G'de bulabilirsiniz.

1. Bölüm 9'da 1. alıştırmadaki yayın şirketinin aynısını hayal edin. Yayın şirketi, çalışmalarını hem kitap hem de audio kaset olarak pazarlıyordu. O alıştırmadaki gibi, bir yayının başlığını (bir karakter katarı) ve fiyatını (`float` tipinde) saklayan `publication` adında bir sınıf tanımlayın. Bu sınıftan iki sınıf türetin: `book`, ilaveten sayfa sayısı (`int` tipinde) içersin ve `tape`, ilaveten dakika cinsinden kasetin oynatma süresini (`float` tipinde) içersin. Üç sınıfın her birinin, kullanıcının klavyeden girdiği verileri almak için bir `getdata()` fonksiyonuna ve verileri ekranda göstermek için bir de `putdata()` fonksiyonuna ihtiyacı vardır.*
   Bir `main()` program yazın. `main()`, `publication`'a işaret eden işaretçilerden oluşan bir dizi tanımlasın. Bu, bu bölümdeki `VIRTPERS` örneğine benzer. Bir döngü içinde, kullanıcının belirli bir kitap veya kaset hakkında veri girmesini isteyin ve bu veriyi tutmak için `book` veya `tape` tipinde bir nesne oluşturmak için `new` operatörünü kullanın. Bu nesneye işaret eden işaretçiyi diziye yerleştirin. Kullanıcı tüm kitap ve kasetlerle ilgili veri girme işlemini tamamladığında, tüm kitap ve kasetler için elde edilen verileri ekranda gösterin. Bunun için bir `for` döngüsü kullanarak dizideki tüm nesnelerin verilerini ekranda gösterecek olan aşağıdaki gibi tek bir ifade yazın:

```
pubarr[j]->putdata();
```

2. Bu bölümdeki `FRENGL` ve `FRISQ` örneklerinde olduğu gibi bir `Distance` sınıfı tanımlayın. İki mesafe ölçüsünün çarpımını yapacak şekilde aşırı yüklenmiş bir `*` operatörü kullanın. Aşağıdaki gibi deyimlerin kullanılabilmesi için bunu bir `friend` fonksiyonu olarak yazın:*

```
Wdist1 = 7.5 * dist2;
```


   Kayan noktalı değerleri `Distance` değerlerine çevirmek için tek argümanlı bir kurucuya ihtiyacınız olacak. Operatörü çeşitli şekillerde test etmek için bir de `main()` program yazın.
3. Daha önce gördüğümüz üzere, bir dizi gibi hareket eden bir sınıf oluşturmak mümkündür. Aşağıda tam bir program şeklinde sunulan `CLARRAY` örneğinde bir dizi sınıfını oluşturmanın bir yolu gösterilmektedir:*

```
// clarray.cpp
// dizi sinifi olusturur
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Array                     //normal bir C++ dizisini modeller
   {
   private:
      int* ptr;                 //dizi icerigine isaretci
      int size;                 //dizinin boyutu
   public:
      Array(int s)              //tek argumanli kurucu fonk.
         {
         size = s;              //arguman, dizinin boyutudur
         ptr = new int[s];      //dizi icin yer ayir
         }
      ~Array()                  //yok edici fonk.
         { delete[] ptr; }
      int& operator [] (int j)  //asiri yuklenmis indeks operatoru
         { return *(ptr+j); }
   };
////////////////////////////////////////////////////////////
int main()
   {
   const int ASIZE = 10;        //dizinin buyuklugu
   Array arr(ASIZE);            //diziyi olustur

   for(int j=0; j<ASIZE; j++)   //diziyi sayilarin kareleri ile doldur
      arr[j] = j*j;

   for(j=0; j<ASIZE; j++)       //dizinin icerigini goster
      cout << arr[j] << ' ';
   cout << endl;
   return 0;
   }
```

   Bu programın çıktısı şöyledir:

```
0 1 4 9 16 25 36 49 64 81
```

   `CLARRAY` ile başlayarak, `Array` sınıfına aşırı yüklenmiş bir atama operatörü ve aşırı yüklenmiş bir kopyalama kurucu fonksiyonu ekleyin. Sonra aşırı yüklenmiş bu operatörlerin çalışıp çalışmadığını kontrol etmek için `main()` programa şu tür ifadeler ekleyin:

```
Array arr2(arr1);
```

   ve

```
arr3 = arr1;
```

Kopyalama kurucu fonksiyonu, dizi elemanlarını saklamak için kendi belleği olan bütünüyle yeni bir **Array** nesnesi tanımlamalıdır. Hem kopyalama kurucu fonksiyonu hem de atama operatörü eski **Array** nesnesinin içeriğini yeni nesneye kopyalamalıdır. Bir boyuttaki bir **Array** dizisini bir başka boyuttaki bir **Array** dizisine atarsanız ne olur?

4. Bu bölümdeki 1. alıştırma ile başlayın; **book** ve **tape** sınıflarına **bool** tipinde **isOversize()** isminde bir üye fonksiyon ekleyin. Diyelim ki, 800 sayfadan daha kalın bir kitap ile 90 dakikadan daha uzun bir oynatma süresine sahip bir kaset "oversize" (büyük boyutlu) olarak ele alınsın. Bu fonksiyona **main()**'den ulaşabilirsiniz ve büyük boyutlu kitap ve kasetler için bunların diğer verilerini ekranda gösterirken "Oversize" karakter katarını da ekrana yazdırabilirsiniz. Eğer **book** ve **tape**, **publication** tipindeki dizinin içinde saklanan ve kendilerini işaret eden işaretçiler ile erişilecekse, **publication** temel sınıfına ne eklemek zorundasınız? Bu temel sınıfın üyelerini örnekleyebilir misiniz?
5. Para karakter katarları için beş aritmetik operatörün aşırı yüklendiği, Bölüm 8'deki 8. alıştırma ile başlayın. O alıştırmada aşırı yüklenmeyen iki operatör daha ekleyin. Bu işlemler

```
long double * bMoney     //sayi carpi para
long double / bMoney     //sayi bolu para
```

friend fonksiyonlarını gerektirir; çünkü, operatörün sol tarafında nümerik bir sabit görünürken sağ tarafında bir nesne yer alır. **main()** programın, kullanıcının iki para karakter katarı ve bir de kayan noktalı bir sayı girmesine imkan verdiğinden emin olun. Programın sonra, bu değer çiftlerinin uygun olanlarını kullanarak yedi adet aritmetik işleminin tümünü gerçekleştirmesini sağlayın.

6. Önceki alıştırmadaki gibi, Bölüm 9'da 8. alıştırmadaki program ile başlayın. Bu kez, bir **bMoney** değerini en yakın dolar değerine yuvarlayan bir fonksiyon ilave edin. Bu fonksiyon şu şekilde kullanılmalıdır:

```
mo2=round(mo1);
```

Bildiğiniz gibi, 0.49 dolar ve bundan daha az olan miktarlar aşağıya yuvarlanır; 0.50 dolar ve bundan daha fazla olan miktarlar yukarıya yuvarlanır. **modfl()** isimli kütüphane fonksiyonu burada kullanışlı olur. Bu fonksiyon, **long double** tipinde bir değişkeni ondalık kısmına ve tamsayı kısmına ayırır. Eğer ondalık kısım 0.50'den küçükse, tamsayı kısmı olduğu gibi döndürün; aksi halde, 1.0 ekleyin. **main()**'de, 0.49 dolardan küçük ve 0.50 dolardan büyük çeşitli **bMoney** miktarlarını fonksiyona göndererek fonksiyonu test edin.

7. Bölüm 10'daki **PARSE** programını hatırlıyor musunuz? Bu programı, tek basamaklı sayılara yerine, diyelim ki **float** tipi gibi, reel sayılar içeren deyimleri hesaplayabilecek şekilde geliştirmek hoş olurdu. Örneğin,

```
3.14159 /2.0 + 75.25 * 3.333 + 6.02
```

Bu hedefe doğru ile adım olarak, hem operatörleri (**char** tipinde) hem de sayıları (**float** tipinde) tutabilen bir yığın geliştirmeniz gerekecek. Fakat, esasen bir dizi olan bir yığında ikifarklı tipi nasıl saklayabilirsiniz? Üstelik, **char** tipi ve **float** tipi aynı boyutta bile değiller. Farklı tiplere işaret eden işaretçileri saklayabilir misiniz? Bunlar aynı

boyuttadır ama, derleyici **char*** tipini ve **float*** tipini aynı dizi içinde saklamaya yine de müsaade etmeyecektir. İki farklı tipteki işaretçiyi aynı dizi içinde saklayabilmenin tek yolu, bu işaretçilerin aynı temel sınıftan türetilmiş olmalarıdır. Böylece, **char**'ı bir sınıfın içine **float**'ı ise bir başka sınıfın içine paketleyebiliriz (encapsulation). Sonra, her iki tipteki işaretçiyi, temel sınıfa işaret eden bir işaretçi dizisinin içinde saklayabiliriz. Temel sınıfın kendisine ait verileri olmasına gerek yok; temel sınıf, kendisinden hiç nesne örneklenmeyen bir soyut sınıf olabilir.

Kurucu fonksiyonlar, her zamanki gibi, değerleri türetilmiş sınıf içinde saklayabilirler. Ancak, değerleri tekrar geri alabilmek için saf sanal fonksiyonlar kullanmanız gerekecektir. Muhtemel bir senaryo şöyle olabilir:

```
class Token              //soyut temel sinif
   {
   public:
      virtual float getNumber()=0;     // saf sanal fonksiyonlar
      virtual char getOperator()=0;
   };
class Operator : public Token
   {
   private:
      char oper;                       //operatorler +, -, *, /
   public:
      Operator(char);                  //kurucu fonksiyon deger atiyor
      char getOperator();              //degeri aliyor
      float getNumber();               //dublor fonksiyon
   };
class Number : public Token
   {
   private:
      float fnum;                      //sayi
   public:
      Number(float);                   //kurucu fonksiyon deger atiyor
      float getNumber();               //degeri aliyor
      char getOperator();              //dublor fonksiyon
   };

   Token* atoken[100];                 //Operator* ve Number* tiplerini tutar
```

Temel sınıf sanal fonksiyonlarının türetilmiş sınıfların tümünde örneklenmeleri gerekir. Bu nedenle **Operand** sınıfının, bir sayı saklayacak olmasa bile **getNumber()** fonksiyonunu içermesi; **Number** sınıfının da bir operand saklamayacak olmasına rağmen **getOperand()** fonksiyonunu içermesi gerekir.

Bu iskeleti, **Token** nesnelerini tutan bir **Stack** sınıfı ekleyerek ve yığına çeşitli operatörleri (mesela, + ve *) ve kayan noktalı sayıları (1.123 gibi) ekleyip çıkaran bir **main()** ilave ederek genişletin; çalışan bir program şekline dönüştürün.

8. Bölüm 10'daki **HORSE** programına ekstra rekabet eden bir at sınıfı ekleyerek, programda küçük bir değişiklik yapalım. Yarış pistinin yarısını geçmiş olan bir atın neredeyse yenilemez olduğunu varsayacağız. At sınıfından **comhorse** (rekabet eden at anlamında) adında bir sınıf türetin. Bu sınıfın içindeki **horse_tick()** fonksiyonunu aşırı yükleyin. Öyle ki, her at kendisinin önde olup olmadığını ve kendisini yakından takip eden (diyelim ki, 0.1 furlong gerisinde) bir at olup olmadığını kontrol edebilsin. Eğer takip eden

varsa, biraz daha hızlanmalıdır. Belki her seferinde kazanmak amacıyla olmayabilir, fakat net bir avantaj sağlamaya yetecek kadar olmalıdır.
Atların her biri, diğer atların nerede olduğunu nasıl bilecek? Söz konusu at, diğer atları tutan belleğe erişebilmeli. Bu bellek, **HORSE** programında **hArray** dizisidir. Her şeye rağmen, dikkatli olun. Rekabet eden **comhorse**'ları tanımlamak istiyorsunuz, atları (**horse**) değil. **comhorse**'ları oluşturmak için yeni bir yarış pisti (**comtrack**) türetmeniz gerekebilir.
Atınızın, aksi halde en önde olacak olan atın önünde olup olmadığını sürekli kontrol edebilirsiniz. Eğer küçük bir farkla öndeyse atınızı biraz hızlandırabilirsiniz.

9. Bölüm 10'daki 4. alıştırma **linklist** sınıfına aşırı yüklenmiş bir yok edici fonksiyon eklemeyi içeriyordu. Varsayalım, yok edici fonksiyonla işlevsel olarak geliştirilmiş böyle bir sınıfın bir nesnesini verilerle doldurduyoruz. Sonra diyelim ki, varsayılan atama operatörünü kullanarak sınıfın tamamına şu şekilde atama yapıyoruz:

```
list2 = list1;
```

Şimdi varsayalım, bu **list1** nesnesini bir süre sonra yok ediyoruz. Aynı verilere erişmek için **list2**'yi halen kullanabilir miyiz? Hayır, çünkü **list1** yok edildiği zaman, onu yok eden yok edici fonksiyon tüm bağlantılarını da ortadan kaldırır. Bir **linklist** nesnesinde yer alan tek veri, aslında ilk bağlantıyı işaret eden işaretçidir. Bağlantılar gittikten sonra **list2** içindeki işaretçi geçersiz olur ve listeye erişmeye çalışmak bizi anlamsız değerlere götürür veya sistemin çökmesine neden olur.
Bunu önlemenin bir yolu, tüm veri bağlantılarını, ilaveten **linklist** nesnesinin kendisini kopyalayacak şekilde atama operatörünü aşırı yüklemektir. Zinciri takip ederek sırayla her bağlantıyı kopyalamanız gerekir. Önceden belirttiğimiz gibi, kopyalama kurucu fonksiyonunun da işlevini artırmalısınız. **linklist** nesnelerinin **main()** içinde yok edilmesinin mümkün olmasını sağlamak için bu nesneleri işaretçiler ve **new** kullanarak oluşturmak isteyebilirsiniz. Bu, yeni rutinleri test etmeyi kolaylaştırır. Kopyalama işlemi verilerin sırasını tersine çevirirse endişe etmeyin.
Dikkat ederseniz, verilerinin tümünü kopyalamak bellek kullanımı açısından çok verimli değildir. Bu yöntemi Bölüm 10'daki **STRIMEM** örneğinde kullanılan yöntemle karşılaştırın. **STRIMEM**, tüm nesneler için sadece tek bir veri seti kullanıyordu ve bu verilere işaret eden nesnelerin sayısını takip ediyordu.

10. Yedinci alıştırmada anlatılan değişiklikleri Bölüm 10'daki **PARSE** programına uygulayın. Yani, kayan noktalı sayıları içeren deyimlerin de "parse" edilmesini mümkün kılın. 7. alıştırmadaki sınıfları **PARSE** algoritması ile birleştirin. Karakterler yerine jetonlara (token) işaret eden işaretçiler üzerinde işlem yapmanız gerekecektir. Yani, programınız şu tür ifadeleri içermelidir:

```
Number* ptrN = new Number(ans);
s.push(ptrN);
```

ve

```
Operator* ptrO = new Operator(ch);
s.push(ptrO);
```


ONİKİNCİ BÖLÜM

12

AKIŞLAR VE DOSYALAR

Akış Sınıfları

Akış Hataları

Akışlarla Disk Dosyası Giriş/Çıkışları

Dosya İşaretçileri

Dosya Giriş/Çıkışlarında Hataların Ele Alınması

Çıkarma ve Ekleme Operatörlerini aşın Yüklemek

Bir Akış Nesnesi Olarak Bellek

Komut Satın Argümanları

Yazıcı Çıktısı

Bu bölüm C++ akış sınıfları üzerine odaklanmıştır. Bu sınıfların düzenlenmesini sağlayan bir hiyerarşiyi inceleyerek başlayacağız; bunların önemli özelliklerini özetleyeceğiz. Bu bölümün büyük bir kısmı, C++ akışlarını kullanarak dosya bağlantılı faaliyetleri nasıl gerçekleştirdiğimizi göstermeye ayrıldı. Verileri çeşitli yollardan dosyalara ne şekilde yazıp, dosyalardan ne şekilde okuduğumuzu; hataları nasıl ele aldığımızı ve dosyalarla nesne yönelimli programlamanın nasıl bir ilişkisi olduğunu göstereceğiz. Bölüm içinde daha sonra C++'ın dosyalarla ilgili diğer birkaç özelliğini daha inceleyeceğiz. Bellek üzerinde metin biçimlendirme, komut satırı argümanları, ekleme ve çıkarma operatörlerinin aşırı yüklenmesi ve verileri yazıcıya göndermek ele alacağımız özellikler arasında yer alıyor.

Akış Sınıfları

Bir *akış* (*stream*), veri akışına verilen genel bir addır. C++'ta bir akış, belirli bir sınıfın nesnesi ile simgelenir. Şu ana kadar, `cin` ve `cout` akış nesneleri kullandık. Farklı türde veri akışlarını simgelemek için farklı akışlar kullanılır. Örneğin, `ifstream` sınıfı, girdi disk dosyalarından gelen veri akışını simgeler.

Akışların Avantajları

C programcıları, I/O için `printf()` ve `scanf()` gibi ve –dosyalar için – `fprintf()`, `fscanf()` vs gibi geleneksel C fonksiyonları kullanmak yerine akış sınıfları kullanmanın ne gibi bir avantajı olduğunu merak edebilirler.

Bunun bir nedeni, basitliktir. Şimdiye dek, bir `printf()` içinde `%f` kullanmanız gerekiyorken `%d` biçimlendirme karakterini kullanmışsanız, bunu takdir edersiniz. Her nesne kendisini nasıl görüntüleyeceğini bildiği için bu tür biçimlendirme karakterleri akışlarda mevcut değildir. Böylece, en temel hata kaynağı ortadan kalmış olur.

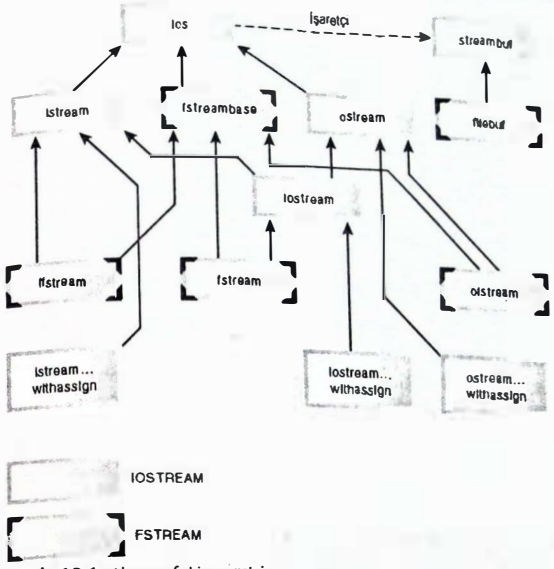
Bir başka neden ise, bu sayede, kendi tanımladığınız sınıflarla çalışacak şekilde mevcut operatörlerin ve fonksiyonların, mesela ekleme (<<) ve çıkarma (>>) operatörlerini aşırı yükleyebilmenizdir. Bu, kendi sınıflarınızın standart tiplerle aynı şekilde çalışmalarını sağlar; böylece, programlamayı bir kez daha kolaylaştırır ve hatalardan kurtarır (estetik olarak çok daha tatmin edici olduğunu söylememize gerek yok).

Ekranda direkt metin çıktısı kullanılmayan Windows gibi grafik kullanıcı arayüzüne sahip bir ortamda programlama yapmayı planlıyorsanız, akış I/O'nın önemli olup olmadığını merak ediyor olabilirsiniz. C++ akışları hakkında yine de bir şeyler bilmeniz gerekir mi? Evet, çünkü verileri dosyaya yazmak için ve metin bazlı giriş/çıkış pencerelerinde ve diğer GUI elemanlarında daha sonra kullanmak üzere verileri bellekte biçimlendirmek için akışlar en iyi yoldur.

Akış Sınıfı Hiyerarşisi

Akış sınıfları oldukça karmaşık bir hiyerarşi içinde düzenlenirler. Şekil 12.1 bu sınıfların en önemlilerinin düzenlenişini gösteriyor.

Bazı akış sınıflarından zaten kapsamlı biçimde yararlanırız. Çıkarma operatörü (>>), `istream` sınıfının bir üyesi; ekleme operatörü (<<) de `ostream` sınıfının bir üyesidir. Bu sınıfların her ikisi de `ios` sınıfından türetilmiştir. Standart çıktı akışını simgeleyen ve genellikle çıktıyı video monitöre yönlendiren `cout` nesnesi, `ostream` sınıfından türetilen `ostream_withassign` sınıfının öntanımlı bir nesnesidir. Benzer şekilde, `cin` de `istream`'den türetilen `istream_withassign` sınıfının bir nesnesidir.



ŞEKİL 12.1: Akış sınıfı hiyerarşisi.

Video monitöre ve klavyeye giriş ve çıkışta kullanılan sınıflar `IOSTREAM` başlık dosyasında bildirilir. `IOSTREAM` başlık dosyasını önceki bölümlerdeki örneklerimize rutin olarak dahil etmiştik. Disk dosyalarının giriş/çıkışlarında kullanılan sınıflar `FSTREAM` dosyasında bildirilir. Şekil 12.1 iki başlık dosyasında hangi sınıfların olduğunu gösteriyor. (Ayrıca, `IOMANIP` içinde bazı manipülatörlerin deklarasyonları; STRSTREAM içinde de bellek içi sınıfların deklarasyonları yer alır.) Bu başlık dosyalarının dökümünü almak ve çeşitli sınıflar arasındaki ilişkileri takip etmek sizin için öğretici olabilir. Bunlar derleyicinizin `INCLUDE` alt dizinindedir. Akışlar hakkında bir çok soru akışların sınıf ve sabit deklarasyonları incelenerek cevaplanabilir.

Şekil 12.1'den görebileceğiniz gibi, `ios` sınıfı hiyerarşinin temel sınıfıdır. `ios` sınıfı, her tür giriş ve çıkış işlemlerinde ortak olan birçok sabit ve üye fonksiyon içerir. Bunlardan bazılarını, mesela `showpoint` ve `fixed` biçimlendirme işaretlerini (formatting flags) önceden görmüştük. `ios` sınıfı ayrıca `streambuf` sınıfına işaret eden bir işaretçi ve verileri idare etmek için alt düzey rutinler de içerir. `streambuf`, verilerinin okunduğu ve yazıldığı gerçek bellek tamponuna sahiptir. Diğer sınıflar tarafından otomatik olarak başvurulan `streambuf` sınıfı hakkında genel olarak endişe etmenize gerek yoktur.

`istream` ve `ostream`, `ios` sınıfından türetilmiş ve sırasıyla giriş ve çıkışa adanmıştır. `istream` sınıfı `get()`, `getline()`, `read()` gibi fonksiyonları ve aşırı yüklenmiş çıkarma operatörünü (>>) içerir. `ostream` ise `put()` ve `write()` fonksiyonlarını ve aşırı yüklenmiş ekleme operatörünü (<<) içerir.

`iostream` sınıfı çoklu kalıtım yoluyla hem `istream`'den hem de `ostream`'den türetilir. Bu sınıftan türetilen sınıflar, disk dosyaları gibi aynı anda hem girdi hem çıktı için açılabilen aygıt-

larla kullanılabilir. Şu üç sınıf `istream_withassign`, `ostream_withassign` ve `iostream_withassign` sırasıyla `istream`, `ostream` ve `iostream`'den kalıtım yoluyla türetilir. Bunlar, bu sınıflara atama operatörünü ilave ederler.

Akış sınıflarının aşağıda yer alan özeti oldukça soyut görünebilir. Bunu şimdi şöyle bir gözden geçirmek ve daha sonra akış bağlantılı belirli bir faaliyetin nasıl gerçekleştirildiğini öğrenmeniz gerektiğinde buraya geri dönmek isteyebilirsiniz.

ios Sınıfı

`ios` sınıfı, tüm akış sınıflarının büyük babasıdır ve C++ akışlarını çalıştırmanız için gerekli olan özelliklerin büyük bölümünü içerir. Biçimlendirme işaretleri, hata durum işaretleri ve dosya işleme modu en büyük öneme sahip üç özelliktir. Bir sonraki bölümde biçimlendirme işaretlerine ve hata durum işaretlerine göz atacağız. Dosya işleme modunu daha sonraya saklayacağız; onu disk dosyalarından bahsederken ele alacağız.

Biçimlendirme İşaretleri

Biçimlendirme işaretleri, `ios` içinde bir sıra `enum` tanımından ibarettir. Biçimlendirme işaretleri, giriş ve çıkış biçimlerinin ve işlemlerinin çeşitli özellikleri için seçenekler sunan birer açık/kapalı anahtar gibi davranırlar. Her işaretle ilgili ayrıntılı bir açıklama sunmayacağız; çünkü, bunların bir kısmının kullanımını zaten görmüştük, diğerleri de az çok kendi kendilerini açıklar. Bazılarını da bu bölümünde daha sonra ele alacağız. Biçimlendirme işaretlerinin komple listesi Tablo 12.1'de veriliyor.

TABLO 12.1: ios Biçimlendirme İşaretleri

İşaret	Anlamı
`skipws`	Girdideki boşlukları atla (dikkate alma)
`left`	Çıktıyı sola hizala `[12.34 ]`
`right`	Çıktıyı sağa hizala `[ 12.34]`
`internal`	İşaret veya taban simgesi ile sayı arasına boşluk doldur `[+ 12.34]`
`dec`	Onluk tabana çevir
`oct`	Sekizlik tabana çevir
`hex`	Onaltılık tabana çevir
`boolalpha`	Bool'u "true" veya "false" karakter katarına çevir
`showbase`	Çıktıda tabanı simgesini kullan (sekizlik için 0, onaltılık için 0x)
`showpoint`	Çıktıda ondalık noktayı kullan
`uppercase`	Büyük harf `X`, `E` ve onaltılık çıktı harflerini kullan (`ABCDEF`) – varsayılan durumda küçük harfler kullanılır
`showpos`	Pozitif tamsayılardan önce + işaretini yaz
`scientific`	Kayan nokta içeren çıktıda üstel biçim kullan `[9.1234E2]`
`fixed`	Kayan nokta içeren çıktıda sabit biçim kullan `[912.34]`
`unitbuf`	Ekleme işleminden sonra tüm akışları temizle (sıfırla)
`stdio`	Ekleme işleminden sonra `stdout` ve `stderror`'u temizle

Biçimlendirme işaretlerini ayarlamanın birkaç değişik yolu vardır; farklı işaretler farklı şekilde ayarlanırlar. Biçimlendirme işaretleri `ios` sınıfının üyesi oldukları için genellikle bunların önünde `ios` ismini ve kapsam çözünürlük operatörünü kullanmalısınız (örneğin, `ios::skipws`). İşaretlerin tümü `setf()` ve `unsetf()` adlı `ios` üye fonksiyonları kullanılarak ayarlanabilir. Aşağıdaki örneğe bir bakın:

```
cout.setf(ios::left);          //cikti metnini sola hizala
cout >> "This text is left-justified";
cout.unsetf(ios::left);        //varsayilani dondur (saga hizali)
```

Biçimlendirme işaretlerinin bir çoğu manipülatör kullanılarak ayarlanabilir. Öyleyse, şimdi manipülatörlere göz atalım.

Manipülatörler

Manipülatörler, bir akış içine doğrudan eklenen biçimlendirme komutlarıdır. Daha önce örneklerini görmüştük. Örneğin, `endl` manipülatörü akışa yeni satır komutunu gönderiyor ve akışı temizliyor:

```
cout <<"To each his own." <<endl;
```

Ayrıca `setiosflags()` manipülatörünü de kullanmıştık (Bölüm 7'de "Diziler ve Karakter Dizileri" bahsinde `SALEMON` programına bakın):

```
cout << setiosflags(ios::fixed)     //sabit ondalik nokta kullan
     << setiosflags(ios::showpoint)    //ondalik noktayi daima goster
     << var;
```

Bu örneklerden de görüldüğü gibi, manipülatörler iki şekilde bulunur: Argüman alanlar ve almayanlar. Tablo 12.2 argüman almayan önemli manipülatörleri özetliyor.

TABLO 12.2: Argüman Almayan ios Manipülatörleri

Manipülatör	Amaç
`ws`	Girdideki görünmeyen karakterleri atlama işlemini aktif hale getir
`dec`	Ondalık tabana çevir
`oct`	Sekizlik tabana çevir
`hex`	Onaltılık tabana çevir
`endl`	Yeni bir satır ekle ve çıktı akışını temizle
`ends`	Çıktı karakter katarını sonlandırmak için null karakteri ekle
`flush`	Çıktı akışını temizle
`lock`	Handle'ı kilitle
`unlock`	Handle'ın kilidini aç

Bu manipülatörler doğrudan akışın içine eklenir. Örneğin, `var`'ın çıkışını onaltılık biçimde almak için şöyle yazabilirsiniz:

```
cout << hex << var;
```

Dikkat ederseniz, manipülatörler sadece akış içinde kendilerini izleyen verileri etkilerler. Kendilerinden önce gelen verileri etkilemezler. Tablo 12.3, argüman alan önemli manipülatörleri özetliyor. Bu fonksiyonlar için `IOMANIP` başlık dosyasına ihtiyacınız vardır.

TABLO 12.3: Argüman Alan ios Manipülatörleri

Manipülatör	Argüman	Amaç
`setw()`	alan genişliği (`int`)	Çıktı için alan genişliğini ayarla
`setfill()`	dolgu karakteri (`int`)	Çıktı için dolgu karakterini ayarla (boşluk karakteri, varsayılan karakterdir)
`setprecision()`	basamak sayısı (`int`)	Gösterilecek basamak sayısını ayarla
`setiosflags()`	biçimlendirme işaretleri (`long`)	Belirtilen işaretleri ayarla
`resetiosflags()`	biçimlendirme işaretleri (`long`)	Belirtilen işaretleri temizle

Fonksiyonlar

`ios` sınıfı, biçimlendirme işaretlerini ayarlamak ve diğer görevleri gerçekleştirmek için kullanabileceğiniz birkaç tane fonksiyon içerir. Bu fonksiyonların bir çoğu Tablo 12.4'le gösterilmiştir. Yalnızca, hataları ele alanlar tabloya eklenmemiştir. Bu fonksiyonları ayrıca inceleyeceğiz.

TABLO 12.4: Ios Fonksiyonları

Fonksiyon	Amaç
`ch = fill();`	Dolgu karakterini döndür (çıktı alanının kullanılmayan bölümünü doldurur; varsayılan karakter boşluk karakteridir)
`fill(ch);`	Dolgu karakterini ayarlar
`p = precision();`	Basamak sayısını al (Kayan noktalı sayılar için ekranda gösterilen basamak sayısı)
`precision(p);`	Basamak sayısını ayarla
`w = width();`	Şu anki alan genişliğini al (karakter olarak)
`width(w);`	Şu anki alan genişliğini ayarla
`setf(flags);`	Belirtilen biçimlendirme işaretlerini ayarla (örneğin, `ios::left`)
`unsetf(flags);`	Belirtilen biçimlendirme işaretlerini ayarlama önceki değerlerine döndür
`setf(flags,field);`	Önce alanı temizle, sonra işaretleri ayarla

Bu fonksiyonlar, normal nokta operatörü kullanılarak belirli akış nesneleri için çağrılırlar. Örneğin, alan genişliğini 12'ye ayarlamak için, şöyle yazabilirsiniz:

```
cout.width(14);
```

Aşağıdaki ifade, dolgu karakterini asteriks olarak ayarlar (baskıyı kontrol etmek amacıyla):

```
cout.fill('*');
```

`ios` biçimlendirme işaretleri üzerinde doğrudan değişiklik yapmak için birkaç tane fonksiyon kullanabilirsiniz. Örneğin, sola hizalama için, şunu kullanın:


```
cout.setf(ios::left);
```

Tekrar sağa hizalı biçime dönmek için, şunu kullanın:

```
cout.unsetf(ios::left);
```

`setf()` fonksiyonunun iki argümanlı versiyonu, ikinci argümanı belirli bir tipteki işaretlerin tümünü veya *alanı* temizlemek amacıyla kullanır. Sonra, birinci argümanda belirtilen işaret ayarlanır. Bu, yeni bir işareti ayarlamadan önce ilgili işaretleri sıfırlamayı kolaylaştırır. Tablo 12.5 bu fonksiyonla ilgili düzenlemeyi gösterir.

```
cout.setf(ios::left, ios::adjustfield);
```

Yukarıdaki ifade, metin hizalamayla ilgili olan işaretlerin tümünü temizler. Sonra, sola hizalı bir çıktı için `left` işaretini ayarlar.

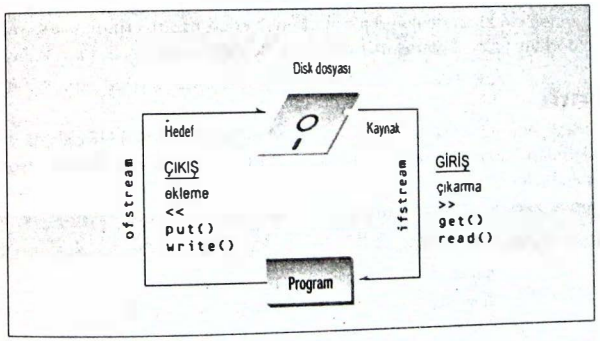
TABLO 12.5: tetf() Fonksiyonunun İki Argümanlı Versiyonu

Birinci Argüman: Ayarlanacak İşaretler	İkinci Argüman: Temizlenecek Alan
`dec, oct, hex`	`basefield`
`left, right, internal`	`adjustfield`
`scientific, fixed`	`floatfield`

Biçimlendirme işaretleriyle ilgili burada gösterilen teknikleri kullanarak giriş/çıkışları biçimlendirmek için genellikle bir yol bulabilirsiniz. Burada, sadece klavye ve monitör giriş/çıkışlarının biçimlendirilmesinden bahsetmiyoruz; ayrıca, bu bölümde daha sonra göreceğimiz dosya giriş/çıkışlarının biçimlendirilmesini da kast ediyoruz.

istream Sınıfı

`ios`'tan türetilen `istream` sınıfı, giriş tabanlı faaliyetleri veya çıkarma işlemini gerçekleştirir. Çıkarma ile çıkış bağlantılı bir faaliyet olan ekleme kolaylıkla birbirine karıştırılabilir. Şekil 12.2 ikisi arasındaki farkı açıklar.



ŞEKİL 12.2: Dosya giriş ve çıkışı.

Tablo 12.6 `istream` sınıfından en çok sıklıkta kullanacağınız fonksiyonları gösteriyor.

TABLO 12.6: istream Fonksiyonları

Fonksiyon	Amaç
`>>`	Tüm temel tipler (ve aşırı yüklenmiş tipler) için biçimlendirilmiş çıkarma işlemi
`get(ch);`	Bir karakter al ve `ch`'e yerleştir
`get(str)`	'`\n`' karakterine kadar karakterleri al, `str` dizisine yerleştir
`get(str, MAX)`	En fazla `MAX` karakter al ve diziye yerleştir
`get(str, DELIM)`	Belirtilen sınırlayıcı karaktere (genellikle '`\n`') kadar karakterleri al, `str` dizisine yerleştir. Sınırlayıcı karakteri akış içinde bırak.
`get(str, MAX, DELIM)`	`MAX` karaktere veya sınırlayıcı karaktere kadar karakterleri al, `str` dizisine yerleştir. Sınırlayıcı karakteri akış içinde bırak.
`getline(str, MAX, DELIM)`	`MAX` karaktere veya sınırlayıcı karaktere kadar karakterleri al, `str` dizisine yerleştir. Sınırlayıcı karakteri de al.
`putback(ch)`	Okunan son karakteri girdi akışına geri koy
`ignore(MAX, DELIM)`	Sınırlayıcı karaktere (genellikle '`\n`') kadar (sınırlayıcı dahil) en fazla `MAX` karakter al ve at
`peek(ch)`	Bir karakter oku ve bunu akışta bırak
`count = gcount()`	`get()`, `getline()` veya `read()` fonksiyonları çağrılarak okunan karakterlerin sayısını döndür
`read(str, MAX)`	Dosyalar için, `EOF`'a kadar en fazla `MAX` karakter al ve `str`'a yerleştir
`seekg()`	Dosyanın başından itibaren dosya işaretçisinin uzaklığını(byte cinsinden) ayarla
`seekg(pos, seek_dir)`	Dosyanın belirtilen yerinden itibaren dosya işaretçisinin uzaklığını (byte cinsinden) ayarla. `seek_dir` şunlardan biri olabilir: `ios::beg`, `ios::cur`, `ios::end`
`pos = tellg(pos)`	Dosyanın başından itibaren işaretçisinin pozisyonunu (byte cinsinden) döndür

Bu fonksiyonlardan bazılarını, mesela `get()` fonksiyonunu önceden görmüştünüz. Bu fonksiyonların çoğu, genellikle klavyeden veri akışını temsil eden `cin` üzerinde işlem yaparlar. Bununla birlikte, son dört tanesi özel olarak disk dosyaları ile ilgilenirler.

ostream Sınıfı

`ostream` sınıfı, çıkış faaliyetlerini veya ekleme işlemini idare eder. Bu sınıfın en çok sıklıkta kullanılan üye fonksiyonları Tablo 12.7'de gösterilmiştir. Bu tablodaki son dört fonksiyon özel olarak disk dosyaları ile ilgilenir.

TABLO 12.7: ostream Fonksiyonları

Fonksiyon	Amaç
`<<`	Tüm temel tipler (ve aşırı yüklenmiş tipler) için biçimlendirilmiş ekleme işlemi
`put(ch)`	Akışa `ch` karakterini ekler
`flush()`	Tamponun içeriğini temizler ve yeni bir satır ekler

TABLO 12.7: ostream Fonksiyonları

Fonksiyon	Amaç
`write(str, SIZE)`	`str` dizisinden `SIZE` tane karakteri dosyaya ekler
`seekp(position)`	Dosyanın başından itibaren dosya işaretçisinin uzaklığını byte cinsinden ayarla
`seekp(position, seek_dir)`	Dosyanın belirtilen yerinden itibaren dosya işaretçisinin uzaklığını byte cinsinden ayarla. `seek_dir` şunlardan biri olabilir: `ios::beg`, `ios::cur`, `ios::end`
`pos = tellp()`	Dosyanın başından itibaren işaretçisinin pozisyonunu (byte cinsinden) döndür

iostream ve _withassign Sınıfları

`istream` ve `ostream` sınıflarının her ikisinden türetilen `iostream` sınıfı sadece, diğer sınıfların özellikle `iostream_withassign` sınıfının, kendisinden türetilebileceği bir temel sınıf görevi görür. Kendisine ait (kurucu ve yok edici fonksiyonlar haricinde) bir fonksiyonu yoktur. `iostream`'den türetilen sınıflar hem giriş hem çıkış işlemlerini gerçekleştirebilir.

Üç adet `_withassign` sınıfı mevcuttur:

- `istream_withassign` sınıfı, `istream`'den türetilmiştir
- `ostream_withassign` sınıfı, `ostream`'den türetilmiştir
- `iostream_withassign` sınıfı, `iostream`'den türetilmiştir

`_withassign` sınıfları, türetildikleri sınıflara çok benzerler; farklı olarak sadece aşırı yüklenmiş atama operatörleri içerirler, bu sayede nesneleri kopyalanabilir.

Niçin kopyalanabilen ve kopyalanamayan ayrı akış sınıflarına ihtiyaç duyuyoruz? Genel olarak, akış sınıf nesnelerini kopyalamak iyi bir fikir değildir. Bunun nedeni, bu tür nesnelerin her birinin, nesnenin gerçek verilerini tutmak için bellekte bir yer içeren belirli bir `streambuf` nesnesi ile ilişkili olmasıdır. Eğer akış nesnesini kopyalıyorsanız, üstelik bir de `streambuf` nesnesini kopyalıyorsanız bu karışıklığa neden olur. Yine de, bir akışı kopyalayabilmek bir-iki durumda önemlidir.

Bu nedenle, `istream`, `ostream` ve `iostream` sınıfları kopyalanamayacak şekilde (aşırı yüklenmiş kopyalama kurucu fonksiyonunu ve atama operatörlerini private yaparak) tanımlanır. Ancak, bu sınıflardan türetilen sınıflar kopyalanabilir.

Öntanımlı Akış Nesneleri

`_withassign` sınıflarından türetilen öntanımlı iki akış nesnesinden şimdiden derinlemesine faydalandık: `cin` ve `cout`. Bunlar normalde sırasıyla klavye ve monitörle bağlantılıdır. Diğer iki öntanımlı nesne ise `cerr` ve `clog`'dur.

- `cin` nesnesi, `istream_withassign` sınıfının bir nesnesidir. Normal olarak klavyeden alınan girdiler için kullanılır.
- `cout` nesnesi, `ostream_withassign` sınıfının bir nesnesidir. Normal olarak monitör görüntüleri için kullanılır.
- `cerr` nesnesi, `ostream_withassign` sınıfının bir nesnesidir, hata mesajları için kullanılır.
- `clog` nesnesi, `ostream_withassign` sınıfının bir nesnesidir, günlük mesajları için kullanılır.

`cerr` nesnesi, genellikle hata mesajları ve program teşhisleri için kullanılır. `cerr`'e gönderilen çıktı, `cout`'ta olduğu gibi tampona yerleştirilmez, anında ekranda gösterilir. Ayrıca, yönlendirilemez (bu konuyla ilgili ileride daha fazla bilgi vereceğiz). Bu nedenlerden ötürü, eğer programınız vaktinden önce sona ererse, en son çıktı mesajını `cerr` kullanarak daha fazla görme şansına sahipsiniz. Bir başka nesne olan `clog`, yönlendirilemiyor olmasından ötürü `cerr`'e benzer. Ancak, `cerr`'in çıktısı tampona yerleştirilmezken `clog`'un çıktısı yerleştirilir.

Akış Hataları

Bu kitapta şimdiye kadar, giriş ve çıkışlar için aşağıdaki ifadeleri kullanarak çoğu kez oldukça açık bir yaklaşım kullandık:

```
cout << "Good morning";
```

ve

```
cin >> var;
```

Bununla birlikte, fark etmiş olabileceğiniz gibi, bu yaklaşım I/O işlemi sırasında hiçbir şeyin yanlış gitmeyeceğini varsayar. Bu, özellikle girdide, her zaman söz konusu değildir. Kullanıcı tamsayı 9 yerine karakter katarı olarak "dokuz" girerse veya hiçbir şey girmeden Enter tuşuna basarsa ne olur? Ya da donanım hatası söz konusu olursa ne olur? Bu bölümde bu tür problemleri ele alacağız. Burada göreceğimiz tekniklerin bir çoğu dosya giriş/çıkışlarına da uygulanabilir.

Hata Durum Bitleri

Akış hata durum işaretleri, bir giriş ve çıkış işleminde ortaya çıkan hataları rapor eden bir `ios` `enum` üyesinden oluşur. Bu işaretler Tablo 12.8'de özetlenmiştir. Şekil 12.3 bu işaretlerin ne şekilde göründüğünü gösterir. Tablo 12.9'da gösterildiği gibi, bu hata işaretlerini okumak (hatta ayarlamak) için çeşitli `ios` fonksiyonları kullanılabilir.

TABLO 12.8: Hata-Durum İşaretleri

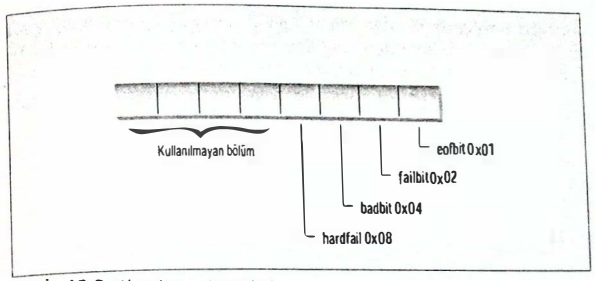
İsim	Anlam
`goodbit`	Hata yok (işaretlerin hiç biri ayarlanmaz, değerleri = 0)
`eofbit`	Dosyanın sonuna ulaşıldı
`failbit`	İşlem başarısız oldu (kullanıcı hatası, vakitsiz `EOF`)
`badbit`	Geçersiz işlem (ilgili bir `streambuf` yok)
`hardfail`	Düzeltilemez hata

TABLO 12.9: Hata İşaretleri İçin Fonksiyonlar

Fonksiyon	Amaç
`int = eof();`	`EOF` işareti ayarlanmışsa, true döndürür
`int= fail();`	Eğer `failbit`, `badbit` veya `hardfail` işareti ayarlanmışsa, true döndürür

TABLO 12.9: Hata İşaretleri İçin Fonksiyonlar

Fonksiyon	Amaç
`int = bad();`	Eğer `badbit` veya `hardfail` işareti ayarlanmışsa, true döndürür
`int = good();`	Herşey tamamsa, true döndürür; işaretlerin hiç biri ayarlanmaz
`clear(int=0);`	Argümansız olduğunda tüm hata bitlerini temizler (sıfırlar); aksi halde, `clear(ios::failbit)`'te olduğu gibi belirtilen işaretleri ayarlar



ŞEKİL 12.3: Akış durum işaretleri.

Sayıları Girmek

Şimdi, sayıları girerken hataları nasıl ele alabileceğimize göz atalım. Bu yöntem, klavyeden veya dosyadan (bunu daha sonra göreceğiz) okunan sayılara uygulanır. Amaç `goodbit`'in değerini kontrol etmek, eğer `true` ise hata sinyali vermek ve doğru girdiyi girmesi için kullanıcıya bir şans daha vermektir.

```
while(true)          //girdi tamamlanana kadar don
   {
   cout << "\nEnter an integer: ";
   cin >> i;
   if( cin.good() )                  //hata yoksa
      {
      cin.ignore(10, '\n');          //newline'i kaldir
      break;                         //donguden cik
      }
   cin.clear();                      //hata bitlerini sifirla
   cout << "Incorrect input";
   cin.ignore(10, '\n');             //newline'i kaldir
   }
cout << "integer is " << i;          //hatasiz tamsayi
```

Bu yöntemin yakaladığı en yaygın hata, klavyeden gelen girdiyi okurken kullanıcının rakam harici veriler girmesidir (örneğin, "9" yerine "dokuz" girilmesi). Bu, `failbit`'in ayarlanmasına neden olur. Bununla birlikte, bu yöntem, disk dosyalarında daha sık görülen sistem bağlantılı hataları da tespit eder.

Kayan noktalı sayılara (`float`, `double` ve `long double`), tamsayılarla aynı şekilde hata analizi yapılabilir.

Çok Fazla Karakter

Çok fazla karakter, sanki film yönetmelerinin tecrübe ettiği bir zorluk gibi gelebilir; fakat, fazladan karakterler, girdi akışlarını okurken de probleme yol açabilirler. Özellikle, girdi tamamlandı sanılırken girdi akışında kalan fazladan karakter bu problemin kaynağı olabilir. Bu karakterler daha sonra, bir sonraki girdi işlemine aktarılmış olurlar; halbuki, bu hiç planlanmamıştır. Geride kalan genellikle newline (yeni satır) karakteridir; fakat, kimi zaman diğer karakterler de kalmış olabilir. Bu alakasız karakterlerden kurtulmak için `istream`'in `ignore(MAX, DELIM)` üye fonksiyonu kullanılır. Bu fonksiyon, belirtilen sınırlandırıcı karakter de dahil olmak üzere en fazla `MAX` sayıda karakteri okur ve atar (dikkate almaz). Örneğimizdeki,

```
cin.ignore(10, '\n');
```

ifadesi, `cin`'in `'\n'` karakteri de dahil olmak üzere en fazla 10 karakter okumasına ve sonra bunları girdiden çıkarmasına neden olur.

Girdisiz Girdi

Boşluklar, mesela sekme ve `'\n'`, sayı girdilerinde normal olarak dikkate alınmazlar (atlanırlar). Fakat bu, istenmeyen yan etkilere neden olabilir. Örneğin, bir sayı girmesi istenen kullanıcı, hiç rakam tuşlamadan sadece Enter tuşuna basabilir. (Belki Enter'a basınca 0 girileceğini düşünüyorlardır ya da belki de sadece zihinler karışmıştır.) Yukarıda gösterilen kodda olduğu gibi, aşağıdaki şu basit ifadede de Enter'a basmak, akış bir sayı girilmesini beklerken, imlecin bir alt satıra kaymasına neden olur:

```
cin >>i;
```

İmlecin alt satıra geçmesinde ne gibi bir problem olabilir? Birincisi, deneyimsiz kullanıcılar Enter'a basınca bir onaylama görmedikleri için bilgisayarın bozulduğunu düşünebilirler. İkincisi, sürekli Enter'a basmak normalde bütün ekran yukarı kaymaya başlayana kadar imlecin devamlı aşağıya inmesine neden olur. Bu, program ve kullanıcının sadece tuşlarla haberleştiği otomatik yazı makinesi stilindeki etkileşimlerde uygun olabilir. Ancak, metin tabanlı grafik programlarında (mesela, Bölüm 13'te "Birden Fazla Dosya Kullanan Programlar" bahsindeki `ELEV` programı) ekranın kayması görüntüyü bozar ve halta en sonunda görüntüyü tamamen ortadan kaldırır.

Bu nedenle, giriş akışının boşlukları dikkate *almasını* akışa bildirebilmek önemlidir. Bu, `skipws` işaretinin silinmesiyle gerçekleştirilir:

```
cout << "\nEnter an integer: ";
cin.unsetf(ios::skipws);      //gorunmeyen karakterleri dikkate al
cin >> i;
if(cin.good())
   {
   //hata yok
   }
//hatali
```


Artık kullanıcı hiç rakam girmeden Enter'a basarsa, `failbit` ayarlanır ve bir hata mesajı verilir. Bundan sonra program kullanıcıya ne yapması gerektiğini bildirebilir ya da ekranın kaymaması için imlecin konumunu değiştirebilir.

Karakter Katarı ve Karakter Girişleri

Kullanıcının karakter katarları ve karakterler girerken gerçek anlamda ciddi bir hata yapması zordur, çünkü tüm girdi, hatta sayılar bile, karakter katarı olarak yorumlanabilir. Yine de, eğer girdi bir disk dosyasından alınıyorsa, karakterler ve karakter katarları, `EOF` veya bundan daha beter bir durumla karşılaşılmaması için hata kontrolünden geçirilmelidir. Sayılardaki durumdan farklı olarak, karakter katarları ve karakterler girerken boşlukları genellikle dikkate *almamanız* gerekebilir.

Hatasız Mesafeler

Şimdi, İngiliz uzaklık ölçülerini kullanan `Distance` sınıfının kullanıcı verilerinin hata kontrolünü yapan bir programa göz atalım. Bu program basitçe, `Distance` değerlerini ayak ve inç cinsinden kullanıcıdan alır ve bu değerleri ekranda gösterir. Bununla birlikte, eğer kullanıcı bir girdi hatası yaparsa, program kullanıcıya uygun bir açıklama yaparak girdiyi reddeder ve kullanıcının yeni veri girmesini ister.

Program çok basittir. Yalnızca, `getdist()` üye fonksiyonu hataları ele alacak şekilde genişletilmiştir. Bu yeni kodu oluşturan bölümler, yukarıdaki parçada gösterilen yöntemi izler. Bununla birlikte, kullanıcının ayak değeri için kayan noktalı bir sayı girmemesini garanti etmek amacıyla birkaç ifade daha eklenmiştir. Bu önemlidir, çünkü ayak değeri bir tamsayı iken inç değeri bir kayan noktalı sayı olduğu için kullanıcı kolaylıkla şaşırabilir.

Her zamanki gibi, çıkarma operatörü (`>>`) bir ondalık nokta gördüğünde hata uyarısında bulunmadan işlemi sona erdirir. Halbuki biz bu tür bir hatadan haberdar olmak istiyoruz; bu nedenle, ayak değeri bir `int` olarak değil, bir karakter katarı olarak okunur. Karakter katarı daha sonra kendi tanımladığımız `isFeet()` fonksiyonu ile incelenir. Eğer karakter katarının ayak için geçerli bir değer taşıdığı doğrulanırsa, `isFeet()` fonksiyonu `true` değerini döndürür. Ayak testini geçmesi için karakter katarının sadece rakamlardan oluşmuş olması ve bu rakamların ifade ettiği değerin –999 ile 999 arasındaki bir sayıya karşılık gelmesi gerekir. (`Distance` sınıfının daha büyük ayak değerleri gerektiren ölçümlerde asla kullanılmayacağını varsayıyoruz.) Eğer karakter katarı ayak testini geçerse, `atoi()` kütüphane fonksiyonunu kullanılarak karakter katarı gerçek `int` değerine çevrilir.

İnç değeri ise bir kayan noktalı sayıdır. İnçin tanımlı olduğu aralığı kontrol etmek istiyoruz. Bu, 0 olabilir veya 0'dan büyük olabilir ama 12.0'den küçük olmalıdır. İnç değeri ayrıca `ios` hata işaretleri açısından da kontrol edilir. Genellikle `failbit` ayarlanır çünkü kullanıcı sayı yerine rakamdan farklı karakterler tuşlamıştır. `ENGLERR` programının listesi işte şöyledir:

```
// englerr.cpp
// Ingiliz Distance sinifi ile giris kontrolu
#include <iostream>
#include <string>
#include <cstdlib>                    //atoi(), atof() icin
using namespace std;
int isFeet(string);                   //deklarasyon
////////////////////////////////////////////////////////////////
class Distance          //English Distance sinifi
   {
```

```
   private:
      int feet;
      float inches;
   public:
      Distance()                          //kurucu (argumansiz)
         { feet = 0; inches = 0.0;}
      Distance(int ft, float in)          //kurucu (iki argumanli)
         { feet = ft; inches = in;}
      void showdist()                     //uzakligi goster
         { cout << feet << "\'-" << inches << '\"'; }
      void getdist();                     //uzakligi kullanicidan al
   };
//-----------------------------------------------------------
void Distance::getdist()                  //uzakligi kullanicidan al
   {
   string instr;                          //giris karakter katari icin

   while(true)                           //ayak dogru girilene kadar donguye devam et
      {
      cout << "\n\nEnter feet: ";
      cin.unsetf(ios::skipws);            //bosluklari atlama
      cin >> instr;                       //ayagi karakter katari olarak al
      if( isFeet(instr) )                 //ayak degeri dogru mu girilmis?
         {                                //evet
         cin.ignore(10, '\n');            //newline dahil karakterleri yok et
         feet = atoi( instr.c_str() );    //tam sayiya cevir
         break;                           //'while'dan cik
         }                                //tam sayi degil
   cin.ignore(10, '\n');                  //newline dahil karakterleri yok et
   cout << "Feet must be an integer less than 1000\n";
   }     //ayak while dongusunun sonu
   while(true)                            //inc degerleri dogru girilene kadar don
      {
      cout << "Enter inches: ";
      cin.unsetf(ios::skipws);            //bosluklari atlama
      cin >> inches;                      //inc'leri oku (float tipi)
      if(inches>=12.0 || inches<0.0)
         {
         cout << "Inches must be between 0.0 and 11.99\n";
         cin.clear(ios::failbit);         //failbit'i yapay olarak ayarla
         }
      if( cin.good() )                    //cin hatasi olup olmadigina bak
         {                          //(cogunlukla rakam girilmemesi seklinde olur)
         cin.ignore(10, '\n');            //newline'i yok et
         break;                           //dogru giris yapilmis, 'while'dan cik
         }
      cin.clear();                        //hata; hata durumunu sifirla
      cin.ignore(10, '\n');               //newline dahil karakterleri yok et
      cout << "Incorrect inches input\n";  //yeniden basla
      }     //inc while dongusunun sonu
   }
//-----------------------------------------------------------
int isFeet(string str)                    //karakter katari dogru (nizami) bir ayak
   {                                      //degeri tasiyorsa true dondur
   int slen = str.size();                 //uzunlugu al
   if(slen==0 || slen >5)                 //giris yoksa ya da cok uzunsa
      return 0;                           //int degil
   for(int j=0; j<slen; j++)              //tek tek tum karakterleri kontrol et
                                          //rakam degilse ya da sifirdan kucukse
```



```
      if( (str[j] < '0' || str[j] > '9') && str[j] !='-') )
         return 0;                        //karakter katari nizami bir ayak degil
   double n = atof( str.c_str() );        //double'a cevir
   if( n<-999.0 || n>999.0 )              //limitlerin disinda mi?
      return 0;                           //nizami bir ayak degil
   return 1;                              //ayak ifadesi nizami
   }
//////////////////////////////////////////////////////////////
int main()
   {
   Distance d;                            //Distance nesnesi olustur
   char ans;
   do
      {
      d.getdist();                        //degerini kullanicidan al
      cout << "\nDistance = ";
      d.showdist();                       //goster
      cout << "\nDo another (y/n)? ";
      cin >> ans;
      cin.ignore(10, '\n');               //newline dahil karakterleri yok et
      } while(ans != 'n');                //'n' girilene kadar donguye devam et
   return 0;
   }
```

Burada başka bir yöntem kullandık: Hata durumunu bildiren bir işareti elle ayarladık. Bu şekilde davrandık; çünkü, inç değerinin 0'dan büyük fakat 12.0'den küçük olmasını garanti etmek istiyoruz. Eğer değilse, **failbit** işareti aşağıdaki ifade ile aktif hale getirilir:

```
cin.clear(ios::failbit);          //failbit'i ayarla
```

Program **cin.good()** ile hataları kontrol ederken **failbit**'in ayarlanmış olduğunu fark edecek ve girdinin hatalı olduğunu işaret edecektir.

Akışlarla Disk Dosyası Giriş/Çıkışları

Programların bir çoğu verileri disk dosyalarına kaydetmek ve sonra geri okumak zorundadır. Disk dosyalarıyla çalışmak bir başka grup sınıfı kullanmayı gerektiriyor: girdi için **ifstream**; hem girdi hem çıktı için **fstream** ve çıktı için **ostream**. Bu sınıfların nesneleri disk dosyaları ile ilişkilendirilebilirler; ayrıca, dosyaları okumak ve yazmak için bu sınıfların üye fonksiyonlarını kullanabiliriz.

Şekil 12.1'e geri dönersek, **ifstream**'in **istream**'den türetildiğini, **fstream**'in **iostream**'den türetildiğini ve **ofstream**'in **ostream**'den türetildiğini görebilirsiniz. Bu ata sınıflar da sırayla **ios** sınıfından türetilmiştir. Dolayısıyla, dosya işlemlerine yönelik sınıflar, üye fonksiyonlarının birçoğunu daha genel sınıflardan türetmişlerdir. Dosya işlemlerine yönelik sınıflar, ayrıca, çoklu kalıtım yoluyla **fstreambase** sınıfından da türetilmiştir. **fstreambase** sınıfı, dosya işlemlerine yönelik bir tampon olan **filebuf** sınıfının bir nesnesini ve daha genel bir sınıf olan **streambuf**'tan türetilen ilgili üye fonksiyonları içerir. Genellikle bu tampon sınıfları dert etmenize gerek yoktur.

ifstream, **ostream** ve **fstream** sınıflarının deklarasyonları **FSTREAM** dosyasında mevcuttur.

C programcıları, C++'ta kullanılan disk I/O yaklaşımının C'dekinden oldukça farklı olduğuna dikkat edeceklerdir. Eski C fonksiyonları, mesela **fread()** ve **fwrite()**, C++'ta halen çalışır, fakat bunlar nesne yönelimli bir ortam için çok uygun değildir. Yeni C++ yaklaşımı,

uygulanabilirlik açısından çok daha net ve kolaydır. (Bu arada, eski C fonksiyonlarını C++ akışları ile karıştırmamaya dikkat edin. Bu fonksiyonların birlikte kullanımına imkan veren yöntemler olmasına rağmen, bunlar birlikte her zaman düzgün çalışmazlar.)

Biçimlendirilmiş Disk Giriş/Çıkışları

Biçimlendirilmiş giriş/çıkışlarda sayılar, disk üzerinde bir karakter serisi olarak saklanır. Yani, 6.02 sayısı 4 byte'lık bir `float` olarak saklanmak yerine `'6'`, `'.'`, `'0'` ve `'2'` karakterleri olarak saklanır. Bu, basamak sayısı çok olan sayılar için kullanışsız olabilir, fakat bir çok durum için uygundur ve kolaylıkla uygulanabilir. Karakterler ve karakter katarları da az çok normal şekilde saklanırlar.

Verileri Yazmak

Aşağıdaki program, bir disk dosyasına bir karakter, bir tamsayı, `double` tipinde bir sayı ve iki `string` nesnesi yazar. Program ekrana çıktı göndermez. `FORMATO` programının listesi şöyledir:

```
// formato.cpp
// << kullanarak, bicimlendirilmis ciktiyi bir dosyaya yazar
#include <fstream>      //dosya I/O icin
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   char ch = 'x';
   int j = 77;
   double d = 6.02;
   string str1 = "Kafka ";              //icinde bosluk icermeyen
   string str2 = "Proust ";             //string'ler

   ofstream outfile("fdata.txt");       //ofstream nesnesini olustur

   outfile <<ch                         //verileri ekle (yaz)
           <<j
           <<' '                        //sayilar arasina bosluk gerekli
           <<d
           <<str1
           <<' '                        //string'ler arasina bosluk gerekli
           <<str2;
   cout << "File written\n";
   return 0;
   }
```

Bu programda, `ofstream` sınıfının bir üyesi olarak `outfile` adında bir nesne tanımlanır. Aynı zamanda bu nesne, `FDATA.TXT` dosyasıyla ilk kullanıma hazırlanır. Bu ataima, dosyanın çeşitli kaynaklarını bir kenara bırakır ve disk üzerindeki bu isme sahip dosyaya erişir; başka deyişle, dosyayı *açar*. Eğer dosya mevcut değilse, oluşturulur. Eğer mevcutsa, dosya kesilir ve yeni veriler eskilerinin yerini alır. `outfile` nesnesi, önceki programlardaki `cout`'unkine benzer bir davranış gösterir. Bu nedenle, temel veri tipindeki herhangi bir değişkenin çıktısını dosyaya göndermek için ekleme operatörünü (<<) kullanabiliriz. Bu yöntem işe yarar, çünkü `ofstream`'in türetildiği `ostream` sınıfı içinde ekleme operatörü uygun biçimde aşırı yüklenir.

Program sona erdiğinde, `outfile` nesnesi kapsamın dışına çıkıyor ve dosyayı kapatacak olan kendi yok edici fonksiyonunu çağırıyor; böylece, dosyayı açıkça bizim kapatmamıza gerek kalmıyor.

Burada, problem çıkarma potansiyeline sahip birkaç tane biçimlendirme pürüzü mevcut. Birincisi, sayılarla (mesela, 77 ve 6.02) nümerik olmayan karakterleri birbirinden ayırmalısınız. Sayılar sabit uzunluktaki alanlar içinde saklanmak yerine bir karakter serisi olarak saklandıkları için, çıkarma operatörü verileri dosyadan geri okurken bir sayının nerede bitip, bir sonrakinin nerede başladığını ancak bu şekilde anlayabilir. İkincisi, karakter katarları da aynı sebepten ötürü boşluklarla birbirinden ayrılmalıdır. Bu, karakter katarlarının içinde gömülü boşluklar olamayacağını bildirir. Bu örnekte, her iki tipteki sınırlayıcı olarak boşluk karakteri (`' '`) kullanılır. Karakterler sabit uzunlukta olduklarından dolayı karakterler için bir sınırlayıcı karaktere gerek kalmaz.

`FORMATO`'nun verileri gerçekten dosyaya yazdığını doğrulayabilirsiniz. Bunun için `FDATA.TXT` dosyasını Windows WORDPAD uygulaması ile açıp veya DOS komutlarından `TYPE` komutunu kullanıp dosyayı incelemeniz yeterlidir.

Verileri Okumak

`FORMATO` tarafından üretilen dosyayı, dosyanın ismini taşıyan bir `ifstream` nesnesi kullanarak okuyabiliriz. Nesne oluşturulduğunda dosya otomatik olarak açılır. Çıkarma operatörünü (>>) kullanarak dosyadan okuyabiliriz.

`FDATA.TXT` dosyasının içindeki verileri okuyan `FORMATI` programının listesi şöyledir:

```
// formati.cpp
// >> kullanarak, bir dosyadan bicimlendirilmis cikisi okur
#include <fstream>          //dosya giris cikisi icin
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   char ch;
   int j;
   double d;
   string str1;
   string str2;

   ifstream infile("fdata.txt");        //ifstream nesnesi olustur
                                        //nesneden veri oku
   infile >> ch >> j >> d >> str1 >> str2;

   cout << ch << endl                   //veriyi goster
        << j << endl
        << d << endl
        << str1 << endl
        << str2 << endl;
   return 0;
   }
```

Burada, `infile` ismini verdiğimiz `ifstream` nesnesi, önceki programlardaki `cin` ile neredeyse aynı şekilde davranır. Dosyaya verileri eklerken verileri doğru olarak biçimlendirmemiz

şartıyla, verileri dosyadan geri alma, uygun değişkenlerde saklamak ve değerlerini görüntüleme problem olmaz. Programın çıktısı şu şekilde görünür:

```
x
77
6.02
Kafka
Proust
```

Sayılar programda saklanmak için elbette tekrar ikili (binary) gösterimlerine çevrilir. Yani 77, j değişkeninde iki karakter olarak değil, **int** tipinde saklanır; 6.02 ise **double** olarak saklanır.

Boşluk İçeren Karakter Katarları

Son örneklerimizde kullanılan teknikler, içinde boşluk içeren **char*** karakter katarları ile çalışmaz. Bu tür karakter katarlarını ele almak için her karakter katarından sonra belirli bir sınırlayıcı karakter yazmanız gerekir. Bu karakter katarlarını okumak için ise çıkarma operatörü yerine **getline()** fonksiyonunu kullanmalısınız. Sıradaki örneğimiz olan **OLINE**, içinde gömülü boşluklar olan bazı karakter katarlarının çıktısını alır.

```
// oline.cpp
// karakter katarlariyla dosya cikisi
#include <fstream>              //dosya giris/cikisi icin
using namespace std;

int main()
   {
   ofstream outfile("TEST.TXT");    //cikis icin dosya olustur
                                    //dosyaya metin gonder
   outfile << "I fear thee,ancient Mariner!\n";
   outfile << "I fear thy skinny hand \n";
   outfile << "And thou art long, and lank, and brown,\n";
   outfile << "As is the ribbed sea sand.\n";
   return 0;
   }
```

Programı çalıştırdığınızda, metnin satırları (Samuel Taylor Coleridge'in *The Rime of the Ancient Mariner* adlı şiirinden alınma) bir dosyaya yazılır. Her satır, newline (yeni satır) karakteri (**'\n'**) ile sona erer. Dikkat ederseniz, bunlar birer **char*** karakter katarıdır; **string** sınıfının nesneleri değil. Akış işlemlerinin birçoğu **char*** karakter katarlarıyla çok daha kolay çalışırlar.

Karakter katarlarını dosyadan çıkarmak için bir **ifstream** tanımlıyoruz ve **istream**'in bir üyesi olan **getline()** fonksiyonunu kullanarak satır satır **ifstream**'den okuyoruz. Bu fonksiyon, **'\n'** karakterine ulaşıncaya dek, boşlukları da dahil ederek karakterleri okur ve elde edilen karakter katarını argüman olarak sağlanan tampona yerleştirir. Tamponun maksimum büyüklüğü ikinci argümanda verilir. Her satırdan sonra tamponun içeriği görüntülenir.

```
// iline.cpp
// karakter katarlariyla dosya girisi
#include <fstream>                      //dosya fonksiyonlari icin
#include <iostream>
using namespace std;
```



```
int main()
   {
   const int MAX = 80;                  //tamponun boyutu
   char buffer[MAX];                    //karakter tamponu
   ifstream infile("TEST.TXT");         //giric icin dosya olustur
   while( !infile.eof() )               //dosya sonuna ulasilana kadar
      {
      infile.getline(buffer, MAX);      //bir metin satiri oku
      cout << buffer << endl;           //goster
      }
   return 0;
   }
```

ILINE'ın ekrandaki çıktısı, **OLINE** tarafından **TEST.TXT** dosyasına yazılan verilerle aynıdır: Coleridge'den alınma bir dörtlük. Program, dosya içinde kaç tane karakter katarı olduğunu önceden bilemez; bu nedenle, dosyanın sonu ile karşılaşana kadar her seferinde bir karakter katarı okumaya devam eder. Bu arada, bu programı rasgele bir dosyayı okumak için kullanmayın. Program, metin satırlarının her birinin **'\n'** karakteri ile sona ermesini gerektirir. Eğer böyle bir durumun söz konusu olmadığı bir dosya ile karşılaşırsanız, program asılı kalacaktır.

Dosya Sonunu Tespit Etmek

Önceden gördüğümüz gibi, **ios**'tan türetilen nesneler, işlemlerin sonuçlarını belirlemek için kontrol edilebilen hata durum işaretleri içerirler. Bir dosyayı küçük küçük okuyorsak, burada yaptığımız gibi, eninden sonunda dosya sonu (**EOF**) koşulu ile karşı karşıya geliriz. **EOF**, okunacak veri kalmadığında işletim sistemi tarafından programa gönderilen işarettir. **ILINE**'da bu durumu aşağıdaki satır ile kontrol edebilirdik:

```
while( !infile.eof() )          //eof ile karsilasana kadar
```

Bununla birlikte, spesifik olarak bir **eofbit**'i kontrol etmek diğer hata işaretlerini, mesela **failbit** ve **badbit**'i kontrol etmeyeceğiz anlamına gelir. Bu tür hatalar daha ender olsa da yine de ortaya çıkabilirler. Bunu gerçekleştirmek için döngü koşulumuzu değiştirebiliriz:

```
while( infile.good() )          //herhangi bir hata ile karsilasana kadar
```

Ayrıca, akışı da doğrudan test edebilirsiniz. Herhangi bir akış nesnesi, mesela **infile**, olağan hata koşulları için, **EOF** da dahil olmak üzere, test edilebilen bir değere sahiptir. Eğer bu tür bir koşul sağlanıyorsa, nesne sıfır değeri döndürür. Eğer her şey yolundaysa nesne sıfırdan farklı bir değer döndürür. Bu değer aslında bir işaretçidir, fakat döndürülen "adres" sıfır veya sıfırdan farklı bir değer olarak test edilmenin dışında hiç bir öneme sahip değildir. Dolayısıyla, **while** döngümüzü yeniden yazabiliriz:

```
while( infile )                 // herhangi bir hata ile karsilasana kadar
```

Bu kesinlikle basit bir ifadedir. Fakat, konuya yabancı olanlar için bu ifadenin ne yaptığı pek de açık olmayabilir.

Karakter I/O

`ostream` ve `istream`'in üyesi olan `put()` ve `get()` fonksiyonları sırasıyla, karakterlerin tek tek çıktısını almak ve karakterleri girdi olarak almak için kullanılabilir. İşte, bir karakter katarının tek tek karakterlerle çıktısını alan `OCHAR` programı:

```
// ochar.cpp
// karakterlerle dosya cikisi          //dosya fonksiyonlari icin
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string str = "Time is a great teacher, but unfortunately "
                "it kills all its pupils.Berlioz ";

   ofstream outfile("TEST.TXT");       //cikis icin dosya olustur
   for(int j=0; j<str.size(); j++)     //her bir karakteri
      outfile.put( str[j] );           //dosyaya yaz
   cout << "File written\n";
   return 0;
   }
```

Bu programda, `OLINE` programında olduğu gibi bir `ostream` nesnesi tanımlanır. Bir `string` nesnesi olan `str`'ın uzunluğu `size()` üye fonksiyonu kullanılarak bulunur ve karakterlerin çıktısı bir `for` döngüsü içinde `put()` fonksiyonu kullanılarak alınır. `TEST.TXT` dosyasına Hector Berlioz'un (19. yüzyıl opera ve program müzikleri bestecisi) bir vecizesi yazılır. `ICHAR` programını kullanarak bu dosyayı geri okuyabilir ve içindekileri görüntüleyebilirsiz.

```
// ichar.cpp
// karakterlerle dosya girisi
#include <fstream>                     //dosya fonksiyonlari icin
#include <iostream>
using namespace std;

int main()
   {
   char ch;                            //okunacak karakter
   ifstream infile("TEST.TXT");        //giris icin dosya olustur
   while(infile)                       //EOF gelene ya da hata olana kadar oku
      {
      infile.get(ch);                  //karakter oku
      cout << ch;                      //goster
      }
   cout << endl;
   return 0;
   }
```

Bu program `get()` fonksiyonunu kullanır ve `EOF`'a ulaşılana (ya da bir hata ile karşılaşılana) kadar okumaya devam eder. Dosyadan okunan karakterler `cout` kullanılarak tek tek ekrana yazılır. Böylece vecizenin tamamı ekrana getirilir.

Dosyadan karakter okumanın bir başka yolu da `ios` sınıfının bir üyesi olan `rdbuf()` fonksiyonunu kullanmaktır. Bu fonksiyon `stream` nesnesi ile ilişkili `streambuf` (ya da `filebuf`) fonksiyonuna bir işaretçi döndürür. Söz konusu nesne, akıştan okunan karakterleri tutan bir tampon içerir. Dolayısıyla, bu nesneye işaret eden işaretçiyi kendi başına bir veri nesnesi olarak kullanabilirsiniz. `ICHAR2`'nin kaynak kodu aşağıdadır:


```
// ichar2.cpp
// dosyadan karakter okumak
#include <fstream>                     //dosya fonksiyonlari icin
#include <iostream>
using namespace std;

int main()
   {
   ifstream infile("TEST.TXT");        //girdi icin dosya olustur

   cout << infile.rdbuf();             //tamponu cout'a gonder
   cout << endl;
   return 0;
   }
```

Bu programın ürettiği sonuç, `ICHAR` ile aynıdır. `ICHAR2` ayrıca, dosya kullanan en kısa program ödülünü de hak eder! Dikkat ederseniz, `rdbuf()`, bir `EOF` ile karşılaştığında geri dönmesi gerektiğini bilir.

İkili Giriş/Çıkış

Biçimlendirilmiş giriş/çıkış kullanarak diske birkaç sayı yazabilirsiniz. Ancak büyük miktarda nümerik veriyi dosyada saklamak durumundaysanız, ikili giriş/çıkış kullanmak daha randımanlı olur. Çünkü, ikili giriş/çıkışta sayılar tıpkı bilgisayarın RAM'ında tutuldukları biçimdedirler. Biçimlendirilmiş olarak yazılıp okunduklarında ise karakter katarı halindedirler. İkili giriş/çıkışta bir `int` 4 byte'ta saklanır. Aynı sayının metin versiyonu ise "12345" olabilir ve bu durumda 5 byte'lık yer gerektirebilir. Benzer şekilde, bir `float` da her zaman 4 byte'ta muhafaza edilir. Aynı sayının biçimlendirilmiş versiyonu örneğin "6.02314e13" olabilir ve bu da tam 10 byte tutabilir.

Şimdi göreceğimiz örnek, bir tamsayı dizisinin ikili biçimde diske yazılıp diskten geri okunmasını işler. Burada iki yeni fonksiyon kullanılır: `ofstream`'in üyesi `write()` ve `ifstream`'in üyesi `read()`. Bu fonksiyonlarda veriler `byte` (`char` tipi) cinsindendir. Fonksiyonlarımız, verinin ne biçimde olduğuyla hiç ilgilenmezler; tampondaki byte'ları diske yazarlar ve gerektiğinde diskten geri okurlar. `write()` ve `read()` fonksiyonlarının parametreleri; veri tamponunun adresi ile uzunluğundan ibarettir. Adres, `reinterpret_cast` kullanılarak `char*` tipine dönüştürülmelidir. Uzunluk da, tampondaki veri unsurlarının sayısı değil, byte (karakter) cinsinden uzunluktur. Şimdi de `BINIO`'nun kaynak kodunu inceleyelim:

```
// binio.cpp
// tamsayilarla ikili giris/cikis
#include <fstream>                     //dosya akisları icin
#include <iostream>
using namespace std;
const int MAX = 100;                   //tamponun boyutu
int buff[MAX];                         //tamsayilar icin tampon

int main()
   {
```

```
   for(int j=0; j<MAX; j++)                  //tamponu veri ile doldur
      buff[j] = j;                           //(0,1,2,...)
                                             //cikis akisini olustur
   ofstream os("edata.dat", ios::binary);
                                             //cikis akisina yaz
   os.write( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
   os.close();                               //kapatmak gerek

   for(j=0; j<MAX; j++)                      //tamponu sil
      buff[j] = 0;
                                             //giris akisi olustur
   ifstream is("edata.dat", ios::binary);
                                             //giris akisindan oku
   is.read( reinterpret_cast<char*>(buff), MAX*sizeof(int) );

   for(j=0; j<MAX; j++)                      //veriyi kontrol et
      if( buff[j] != j )
         { cerr << "Data is incorrect\n"; return 1; }
   cout << "Data is correct\n";
   return 0;
   }
```

İkili veri ile çalışırken **write()** ve **read()**'in ikinci parametresinde **ios::binary** argümanını kullanmalısınız. Çünkü, bunu belirtmediğiniz zaman otomatik olarak kullanılan metin modu, veriyi kendine göre biçimlendirebilir. Örneğin, metin modunda '**\n**' karakteri iki byte'a çevrilir – bir carriage return (satır sonu) ve bir de linefeed (paragraf başı) karakteri olmak üzere. Bu sayede biçimlendirilmiş metin dosyası **TYPE** gibi DOS tabanlı programlar tarafından daha kolay okunabilir. Ama aynı uygulama ikili veride karışıklığa sebep olur; ASCII değeri 10 olan her karakter iki byte'a dönüştürülür. Burada gördüğümüz **ios::binary** argümanı *mod bit*'inin bir örneğidir. Bu konuya birazdan **open()** fonksiyonunu ele alırken dönüp ayrıntısına bakacağız.

reinterpret_cast Operatörü

BINIO programında (ve takip edecek pek çok programda), **int** tipinden bir tamponun **read()** ve **write()** fonksiyonlarına **char** tipinden bir tampon olarak görünmesini sağlamak için **reinterpret_cast** operatörü kullanılır.

```
is.read( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
```

reinterpret_cast operatörü derleyiciye şu mesajı verdiğiniz anlamına gelir: "Bundan hoşlanmadığının farkındayım ama yine de böyle yapmak istiyorum." Bu operatör, anlamlı olduğuna bakıp bakmadan, belleğin bir kısmında tip değişikliği yapar. Size düşen, itinayla kullanmaktır.

reinterpret_cast operatörünü, işaretçi değerlerini tamsayıya ve tamsayıları işaretçi değerlerine dönüştürmek için de kullanabilirsiniz. Bu çok tehlikeli bir uygulamadır ama bazen de gereklidir.

Dosyaları Kapatmak

Şu ana kadar örnek programlarımızda akışları açıkça kapatmamıza gerek yoktu; çünkü, akışlar geçerli oldukları alanının dışına çıktıklarında otomatik olarak kapatılıyorlar. Kapsamın dışına çıkmış olmaları akışların yok edici fonksiyonlarının çağrılmasına neden oluyor ve ilgili dosyayı

kapatıyor. Buna rağmen, **BINIO** programında hem çıktı akışı (**os**) hem de girdi akışı (**is**) aynı dosya (**EDATA.DAT**) ile ilişkili olduğu için ikinci akış açılmadan önce birincisinin kapatılması gerekir. Bunun için **close()** üye fonksiyonu kullanılır.

Bir dosyayı her kapatışınızda akışın yok edici fonksiyonuna güvenmeksizin açıkça bir **close()** fonksiyonu kullanmak isteyebilirsiniz. Bu, potansiyel olarak daha güvenilirdir ve listeyi kesinlikle daha okunur kılar.

Nesne Giriş/Çıkışları

C++ nesne yönelimli bir dil olduğu için nesnelerin diske nasıl yazıldığını ve diskten nasıl okunduğunu merak etmek mantıklıdır. Sıradaki örnekler bu işlemleri gösterir. Önceki birkaç örnekte kullanılan **person** sınıfı (örneğin, Bölüm 11'de "Sanal Fonksiyonlar" bahsindeki **VIRTPERS** programı) nesneleri tedarik eder.

Bir Nesneyi Diske Yazmak

Bir nesneyi yazarken genellikle ikili mod kullanmak isteriz. Bu, bellekte saklanmış olan bit konfigürasyonunun aynını diske yazar ve nesne içindeki nümerik verilerin düzgün biçimde ele alınmasını garanti eder. Kullanıcının **person** sınıfının bir nesnesi hakkında bilgi girmesini isteyen ve bu nesneyi **PERSON.DAT** disk dosyasına yazan **OPERS** programın listesi şöyledir:

```
// opers.cpp
// person nesnesini diske yazar
#include <fstream>                           //dosya akisları icin
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person                                 //kisi sinifi
   {
   protected:
      char name[80];                         //ismi
      short age;                             //yasi
   public:
      void getData()                         //kisinin verilerini al
         {
         cout << "Enter name: "; cin >> name;
         cout << "Enter age: "; cin >> age;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   person pers;                              //kisi kaydi olustur
   pers.getData();                           //kisinin verilerini al
                                             //ofstream nesnesi olustur
   ofstream outfile("PERSON.DAT ", ios::binary);
                                             //nesneye yaz
   outfile.write(reinterpret_cast<char*>(&pers), sizeof(pers));
   return 0;
   }
```

person'ın **getData()** üye fonksiyonu, kullanıcıdan bilgileri girmesini istemek için çağrılır. Fonksiyon, alınan bilgileri **pers** nesnesine yerleştirir. Örnek bir etkileşim şu şekilde olabilir:

```
Enter name: Coleridge
Enter age: 62
```

`pers` nesnesinin içeriği daha sonra `write()` fonksiyonu kullanılarak diske yazılır. `pers` nesnesinin boyutunu bulmak için `sizeof` operatörü kullanılır.

Diskten Bir Nesne Okumak

`PERSON.DAT` dosyasından bir nesne okumak `read()` üye fonksiyonunu kullanmayı gerektirir. `IPER` programının listesi şöyledir:

```
// ipers.cpp
// person nesnesini diskten okur
#include <fstream>                      //dosya akisları icin
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class person                            //kisi sinifi
   {
   protected:
      char name [80];                   //ismi
      short age;                        //yasi
   public:
      void showOata()                   //kisinin verilerini goster
         {
         cout << "Name: " << name << endl;
         cout << "Age: " << age << endl;
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   person pers;                                     //kisi degiskeni olustur
   ifstream infile("PERSON.DAT", ios::binary);      //akis olustur
                                                    //akisi oku
   infile.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
   pers.showData();                                 //kisiyi goster
   return 0;
   }
```

`IPERS`'in çıktısı, `OPERS` programının `PERSON.DAT` dosyasına yerleştirdiği verileri yansıtır:

```
Name: Coleridge
Age: 62
```

Uyumlu Veri Yapıları

Dosyalara nesne yazıp dosyalardan nesne okuyan `OPERS` ve `IPERS` gibi programların düzgün çalışmaları için aynı sınıfın nesnelerinden bahsediyor olmaları gerekiyor. Bu programlardaki `person` sınıfının nesneleri tam olarak 82 byte uzunluğundadır: İlk 80 byte kişinin ismini simgeleyen karakter katarı tarafından kullanılır ve son iki byte ise, kişinin yaşını simgeleyen `short` tipinde bir tamsayı içerir. Eğer programların ikisi de, mesela isim alanının farklı uzunlukta olduğunu düşünselerdi, bu programlardan hiç biri bir diğerinin ürettiği dosyayı doğru olarak okuyamazdı.

Bununla birlikte, `OPERS` ve `IPERS` programlarındaki `person` sınıflarının aynı verileri içermelerinin yanı sıra farklı üye fonksiyonlarına sahip olduklarına dikkat edin. İlki `getData()` adında tek bir fonksiyon içerirken ikincisi sadece `showData()` fonksiyonuna sahiptir. Üye fonksiyonlar, nesne verileriyle birlikte diske yazılmadıkları için hangi üye fonksiyonları kullandığınız önemli değildir. Veriler aynı biçimde olmak zorundadır, fakat üye fonksiyonlar arasındaki çelişkilerin bir etkisi yoktur. Yine de bu, sanal fonksiyon kullanmayan basit sınıflar için geçerlidir.

Eğer türetilmiş sınıfların nesnelerini bir dosyadan okuyup, dosyaya yazıyorsanız, daha dikkatli olmalısınız. Türetilmiş sınıfların nesneleri, nesnenin bellekte yer alan verilerinin hemen önüne yerleştirilmiş gizemli bir sayı içerirler. Bu sayı, sanal fonksiyonlar kullanıldığında nesnenin sınıfını belirlemeye yardımcı olur. Bir nesneyi diske yazdığınızda nesnenin diğer verileriyle birlikte bu sayı da yazılır. Bir sınıfın üye fonksiyonlarını değiştirirseniz bu sayı da değişir. Eğer bir sınıfa ait bir nesneyi bir dosyaya yazıyorsanız, sonra bunu, aynı verileri içeren fakat farklı bir üye fonksiyona sahip bir sınıfa ait bir nesnenin içine okuyorsanız, bir de nesne üzerinde sanal fonksiyonları kullanmaya kalkışmışsanız başınız büyük belada demektir. Kısaca hisse: Bir nesneyi okumak için kullanılan bir sınıfın, nesneyi yazmak için kullanılan sınıf ile *aynı* olduğundan emin olun.

Ayrıca, veri üyesi olarak işaretçileri içeren nesneler ile disk giriş/çıkışlarına girişmemelisiniz. Tahmin edebileceğiniz gibi, nesne bellekte farklı bir yerden yeniden okunduğu zaman işaretçi değerleri doğru olamayacaktır.

Birden Fazla Nesne ile I/O

`OPERS` ve `IPERS` programları bir kerede sadece bir nesne yazıp okuyorlardı. Sıradaki örneğimiz bir dosya açar ve kullanıcının istediği kadar sayıda nesneyi dosyaya yazar. Sonra dosyanın bütün içeriğini okur ve görüntüler. `DISKFUN` programının çıktısı şöyledir:

```
// diskfun.cpp
// cesitli nesneleri diske yazar ve diskten okur
#include <fstream>                      //dosya akisları icin
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class person                            //kisi sinifi
   {
   protected:
      char name[80];                    //adi
      int age;                          //yasi
   public:
      void getData()                    //kisinin verilerini al
         {
         cout << "\n   Enter name: "; cin >> name;
         cout << "   Enter age :"; cin >> age;
         }
      void showData()                   //kisinin verilerini goster
         {
         cout << "\n   Name: " << name;
         cout << "\n   Age: " << age;
         }
   };
////////////////////////////////////////////////////////////
int main()
```

```
    {
    char ch;                          //kisi nesnesi olustur
    person pers;                      //giris/cikis dosyasi olustur
    fstream file;                     //eklemek uzere ac
    file.open("GROUP.DAT", ios::app | ios::out |
                                      ios::in | ios::binary );
                                      //kullanici verileri dosyaya gider
    do
       {
       cout << "\nEnter person 's data:";
       pers.getData();                //bir kisinin verilerini al
                                      //dosyaya yaz
       file.write( reinterpret_cast<char*>(&pers), sizeof(pers) );
       cout << "Enter another person (y/n)? ";
       cin >> ch;
       }
    while(ch=='y');                   //'n' girilene kadar devam et
    file.seekg(0);                    //dosyanin basina don
                                      //ilk kisiyi oku
    file.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
    while( !file.eof() )              //dosya sonunu gorene kadar devam et
       {
       cout << "\nPerson:";          //kisiyi goster
       pers.showData();               //baska bir kisiyi oku
       file.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
       }
    cout << endl;
    return 0;
    }
```

İşte, **DISKFUN** ile bazı örnek etkileşimler. Burada gösterilen çıktı, programın önceden çalıştırılmış olduğunu ve iki **person** nesnesinin dosyaya önceden yazılmış olduğunu varsayar.

```
Enter person 's data:
   Enter name: McKinley
   Enter age: 22
Enter another person (y/n)? n
Person:
   Name: Whitney
   Age: 20
Person:
   Name: Rainier
   Age: 21
Person:
   Name: McKinley
   Age: 22
```

Burada dosyaya bir nesne daha eklenir ve üç nesneyi birden içeren bütün dosya içeriği görüntülenir.

fstream Sınıfı

Bu bölümde şimdiye kadar tanımladığımız dosya nesneleri ya girdi ya da çıktı içindi. **DISKFUN**'da hem girdi hem de çıktı için kullanılabilecek bir dosya tanımlamak istedik. Bu, **iostream**'den türetilen bir **fstream** nesnesi gerektirir; **iostream** ise hem **istream**'den hem de **ostream**'den türetilir. Böylece, dosya hem girdi hem çıktıyı idare edebilir.

open() Fonksiyonu

Önceki örneklerde bir dosya nesnesi tanımladık ve bunu aynı ifade içinde ilk kullanıma hazırladık:

```
ofstream outfile("TEST.TXT");
```

DISKFUN'da ise farklı bir yaklaşım kullandık: Dosyayı bir ifade içinde tanımlıyoruz ve **fstream** sınıfının bir üyesi olan **open()** fonksiyonunu kullanarak bir başka ifade içinde açtık. Bu, dosya açma işleminin başarısız olabileceği durumlar için kullanışlı bir yöntemdir. Bir akış nesnesini bir kez tanımlayabilirsiniz. Daha sonra, her seferinde yeni bir akış tanımlama yükünü taşımadan dosyayı tekrar tekrar açmayı deneyebilirsiniz.

Mod Bitleri

ios::binary mod bitini önceden görmüştük. **open()** fonksiyonuna birkaç yeni mod biti dahil ediyoruz. **ios** içinde tanımlı olan mod bitleri, bir akış nesnesinin nasıl açılacağını ilgilendiren çeşitli özellikleri belirtir. İhtimaller Tablo 12.10'da gösterilmiştir.

TABLO 12.10: open() Fonksiyonu İçin Mod Bitleri

Mod Biti	Sonuç
in	Okuma için aç (**ifstream** için varsayılan değer)
out	Yazma için aç (**ofstream** için varsayılan değer)
ate	Dosyanın sonundan okumaya veya yazmaya başla (AT End)
app	Dosyanın sonundan yazmaya başla (APPend)
trunc	Eğer dosya mevcutsa dosyayı sıfır uzunluğuna gelecek şekilde kes (TRUNCate)
nocreate	Mevcut olmayan dosyayı açmaya çalışmaktan kaynaklanan hata
noreplace	Dosya mevcutsa fakat **ate** veya **app** ayarlanmamışsa, dosyayı çıktı için açmaktan kaynaklanan hata
binary	Dosyayı ikili modda (metin modunda değil) aç

DISKFUN'da **ios::app** kullandık, çünkü dosyada önceden mevcut ne varsa korumak istedik. Yani, dosyaya yazabiliriz, programı sona erdirebiliriz ve programı yeniden çalıştırabiliriz. Bu durumda, dosyaya yazdıklarımız dosyanın mevcut içeriğinin peşine eklenecektir. **ios::in** ve **ios::out** kullandık, çünkü dosya üzerinde hem girdi hem de çıktı işlemlerini gerçekleştirmek istedik. Ayrıca, **ios::binary** kullandık, çünkü ikili nesneleri yazmak istedik. İşaretler arasındaki düşey çubuklar, bu işaretleri simgeleyen bitlerin mantıksal olarak tek bir tamsayı içinde birleştirilmesini sağlar; böylece, birkaç tane işaret aynı anda uygulanabilir.

write() fonksiyonu kullanılarak dosyaya bir kerede bir **person** nesnesi yazılır. Yazma işlemi bittiğinde, dosyanın tümünü okumak istiyoruz. Bunu yapmadan önce dosyanın şimdiki pozisyonunu sıfırlamalıyız. Bunu **seekg()** fonksiyonu ile gerçekleştiririz. **seekg()** fonksiyonunu bir sonraki bölümde inceleyeceğiz. Bu fonksiyon, dosyanın en başından okumaya başlamamızı garanti eder. Sonra, bir **while** döngüsü içinde, dosyadan tekrar tekrar bir **person** nesnesi okunur ve ekranda gösterilir.

Bu işlem, tüm **person** nesnelerini okuyana kadar sürer. Tüm **person** nesnelerinin okunduğu, **ios::eofbit**'in durumunu döndüren **eof()** fonksiyonu kullanılarak anlaşılır.

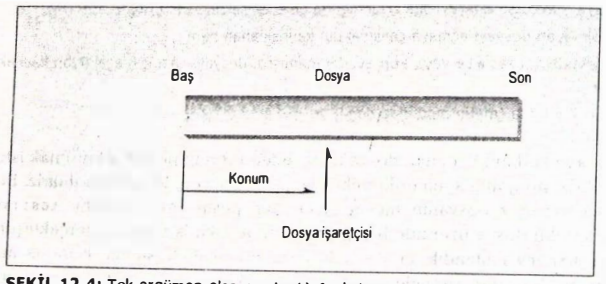
Dosya İşaretçileri

Her dosya nesnesi, *get işaretçisi* (*get pointer*) ve *put işaretçisi* (*put pointer*) denilen iki tamsayı değer ile ilişkilendirilmiştir. Bunlara ayrıca *mevcut alma konumu* (*current get position*) ve *mevcut koyma konumu* (*current put position*) – veya hangisinin kast edildiği belli ise – sadece *mevcut konum* (*current position*) da denir. Bu değerler, dosya içinde yazma veya okuma işleminin gerçekleştirileceği byte sayısını belirtir. (Bu bağlamda kullanılan *işaretçi* terimini adres değişkenleri olarak kullanılan normal C++ işaretçileri ile karıştırmayın.)

Genellikle mevcut bir dosyanın başından okumaya başlayıp sonuna kadar devam etmek istersiniz. Yazarken ise, dosyanın başından, mevcut içeriği silerek başlamak isteyebilirsiniz. Ya da sonundan başlamak isteyebilirsiniz; bu durumda, dosyayı **ios::app** mod belirteci ile açabilirsiniz. Bunlar varsayılan davranışlardır, yani dosya işaretçilerinin herhangi bir şekilde ayarlanması şart değildir. Buna rağmen, dosya işaretçilerinin kontrolünü kendi elinize almanız gereken durumlar olabilir. Böylece, dosya içinde herhangi bir yerden okuyabilir ve herhangi bir yere yazabilirsiniz. **seekg()** ve **tellg()** fonksiyonları, get işaretçisini ayarlamanıza ve incelemenize imkan verir; **seekp()** ve **tellp()** fonksiyonları da aynı faaliyetleri put işaretçisi üzerinde gerçekleştirir.

Pozisyonu Belirtmek

DISKFUN programında get işaretçisini konumlandırma ile ilgili bir örnek gördük. **seekg()** fonksiyonu, işaretçiyi dosyanın başını gösterecek şekilde ayarlıyordu, böylece okuma buradan başlayabilecekti. **seekg()**'nin bu şekli, dosyanın içindeki mutlak pozisyonu simgeleyen tek argüman alır. Dosyanın başı byte 0'dır; zaten **DISKFUN**'da da kullandığımız buydu. Şekil 12.4'te bu fonksiyon açıklanmıştır.



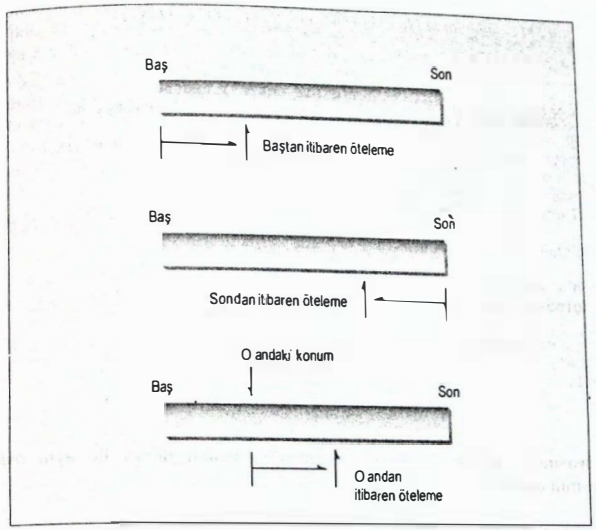
ŞEKİL 12.4: Tek argüman alan **seekg()** fonksiyonu.

Ötelemeyi Belirtmek

seekg() fonksiyonu iki şekilde kullanılabilir: Birincisini gördük; fonksiyonun tek argümanı dosyanın başından itibaren pozisyonu simgeler. Bu fonksiyonu iki argüman ile de kullanabilirsiniz. Bu durumda, birinci argüman, dosyanın içinde belirli konumdan itibaren bir ötelemeyi (offset) simgeler; ikinci argüman ise ötelemenin ölçüldüğü başlangıç konumunu belirtir. İkinci argüman için üç ihtimal vardır: **beg**, dosyanın başı; **cur**, işaretçinin şu anki pozisyonu ve **end**, dosyanın sonu. Örneğin şu ifade, put işaretçisini dosyanın en sonundan 10 byte öncesine ayarlayacaktır:


```
seekp(-10, ios::end);
```

Şekil 12.5'te bunun nasıl gerçekleştirildiği gösterilmiştir.



ŞEKİL 12.5: İki argümanlı **seekg()** fonksiyonu.

Şimdi, **GROUP.DAT** dosyasındaki belirli bir **person** nesnesini bulmak için **seekg()** fonksiyonunun iki argümanlı versiyonunu kullanan ve bu şahsın verilerini görüntüleyen bir örnek verelim. **SEEKG** programının listesi şöyledir:

```
// seekg.cpp
// dosyada belli bir kisiyi arar
#include <fstream>                      //dosya akisları icin
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person                            //kisi sinifi
   {
   protected:
      char name[80];                    //adi
      int age;                          //yasi
   public:
      void getData()                    //kisinin verilerini al
        {
```

```
        cout << "\n   Enter name: "; cin >> name;
        cout << "   Enter age: "; cin >> age;
        }
    void showData(void)                    //kisinin verilerini goster
        {
        cout << "\n   Name: " << name;
        cout << "\n   Age: " << age;
        }
};
////////////////////////////////////////////////////////////////
int main()
    {
    person pers;                           //kisi sinifi olustur
    ifstream infile;                       //giris dosyasi olustur
    infile.open("GROUP.DAT", ios::in | ios::binary);       //dosyayi ac

    infile.seekg(0, ios::end);             //sondan itibaren 0 byte git
    int endposition = infile.tellg();      //yerimizi bul
    int n = endposition / sizeof(person);  //kisilerin sayisi
    cout << "\nThere are " << n << " persons in file";

    cout << "\nEnter person number: ";
    cin >> n;
    int position = (n-1) * sizeof(person);  //sayi carpi boyut
    infile.seekg(position);                 //bastan kacinci byte?
                                            //bir kisi oku
    infile.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
    pers.showData();                        //kisiyi goster
    cout << endl;
    return 0;
    }
```

GROUP.DAT dosyasının, az önce **DISKFUN** örneğinde erişilen dosya ile aynı olduğunu varsayarsak, programın çıktısı şöyle görünür:

```
There are 3 persons in file
Enter person number:2
   Name: Rainier
   Age: 21
```

Program, dosyanın öğelerini 0'dan başlayarak numaralandırıyor olsa da, biz kullanıcı açısından bunları 1'den başlayarak numaralandırdık. Bu nedenle, 2. kişi, üç kişilik dosyadaki ikinci kişidir.

tellg() Fonksiyonu

Programın ilk yaptığı iş, dosyada kaç kişi bulunduğunu tespit etmektir. Program bu işi, get işaretçisini dosyanın sonuna konumlandırarak gerçekleştirir. Bunun için şu ifadeyi kullanır:

```
infile.seekg(0, ios::end);
```

tellg() fonksiyonu, get işaretçisinin mevcut konumunu döndürür. Program bu fonksiyonu, işaretçi dosyanın sonunda iken işaretçinin konumunu döndürmek için kullanır. Bu, dosyanın byte cinsinden uzunluğudur. Program sonra, dosyada kaç kişi olduğunu dosya uzunluğunu bir **person**'ın boyutuna bölerek hesaplar ve sonucu ekranda gösterir.

Yukarıda gösterilen çıktıda kullanıcı dosyadaki ikinci nesneyi seçer ve program **seekg()**'yi kullanarak bunun dosyada kaç byte ileride olduğunu hesaplıyor. Program sonra, bir kişinin (**person**) bu noktadan başlayan verilerini okumak için **read()**'i kullanır. Son olarak da, verileri **showData()** ile ekranda gösterir.

Dosya Giriş/Çıkışlarında Hataların Ele Alınması

Şimdiye kadar ele aldığımız dosya bağlantılı örneklerde hata durumlarıyla ilgilenmedik. Özellikle, okumak için açtığımız dosyaların zaten mevcut olduğunu ve yazmak için açtıklarımızın da oluşturulabileceğini veya eklenebileceğini varsaydık. Ayrıca, okuma veya yazma sırasında başarısız bir durum olmayacağını farz ettik. Halbuki gerçek bir programda bu tür varsayımları doğrulamak ve bu varsayımların yanlış olduğu ortaya çıkarsa uygun bir davranışta bulunmak önemlidir. Mevcut olduğunu düşündüğünüz bir dosya mevcut olmayabilir veya yeni bir dosya için kullanabileceğinizi zannettiğiniz bir dosya ismi zaten mevcut bir dosyada kullanılıyor olabilir. Ya da diskte hiç boş yer kalmamış olabilir, ya da sürücüde disket bulunmuyor olabilir vs.

Hatalara Karşı Tavır Almak

Sıradaki örneğimiz bu tür hataların en uygun biçimde nasıl ele alındığını gösterir. Disk işlemlerinin tümü işlem gerçekleştirildikten sonra kontrol edilir. Eğer bir hata ortaya çıkmışsa, ekrana bir mesaj yazdırılır ve program sona erer. Bu programda, daha önce bahsedilen, nesnenin hata durumunu belirlemek için nesnenin kendisinin döndürdüğü değeri kontrol etmeye dayanan bir teknik kullandık. Program bir çıkış akış nesnesi açar, tek bir **write()** çağrısı ile bu nesneye bir tamsayılar dizisinin tamamını yazar ve nesneyi kapatır. Program daha sonra bir giriş akış nesnesi açar ve bir **read()** çağrısı ile tamsayı dizisini okur.

```
// rewerr.cpp
// giris ve cikistaki hatalari yonetir
#include <fstream>                     //dosya akisları icin
#include <iostream>
using namespace std;
#include <process.h>                   //exit() icin

const int MAX = 1000;
int buff[MAX];

int main()
   {
   for(int j=0; j<MAX; j++)            //tamponu veri ile doldur
      buff[j] = j;
   ofstream os;                        //cikis akisi olustur
                                       //akisi ac
   os.open("a:edata.dat", ios::trunc | ios::binary);
   if(!os)
      { cerr << "Could not open output file\n"; exit(1); }

   cout << "Writing...\n ";            //tamponu akisa yaz
   os.write( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
   if(!os)
      { cerr << "Could not write to file\n"; exit(1); }
   os.close();                         //kapatmak gerek

   for(j=0; j<MAX; j++)                //tamponu temizle
```

```
    buff[j] = 0;

 ifstream is;                          //giris akisi olustur
 is.open("a:edata.dat", ios::binary);
 if(!is)
    { cerr << "Could not open input file\n"; exit(1); }

 cout << "Reading...\n";               //read file
 is.read( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
 if(!is)
    { cerr << "Could not read from file\n"; exit(1); }

 for(j=0; j<MAX; j++)                  //veriyi kontrol et
    if( buff[j] !=j )
       { cerr << "\nData is incorrect\n"; exit(1); }
 cout << "Data is correct\n";
 return 0;
 }
```

Hata Analizi

REWERR programında, bir akış nesnesinin tümünün döndürdüğü değeri inceleyerek bir I/O işleminde hata olup olmadığını belirledik.

```
if (!is)
   //hata ortaya cikti
```

Burada, her şey yolunda giderse **is** bir işaretçi değeri, aksi halde 0 döndürür. Bu hatalara balyozla ile yaklaşmaya benzer: Hata ne olursa olsun, hatalar aynı şekilde tespit edilir ve hatalardan kurtulmak için aynı davranışta bulunulur. Bununla birlikte, **ios** hata işaretlerini kullanarak bir dosya I/O hatası hakkında daha spesifik bilgi elde etmek mümkündür. Bu durum işaretlerinden bazılarının ekran ve klavye giriş/çıkışlarındaki kullanımlarını önceden görmüştük. Sıradaki örneğimiz olan **FERRORS**, bu işaretlerin dosya giriş/çıkışlarında nasıl kullanılabileceğini gösteriyor.

```
// ferrors.cpp
// dosya acarken olusan hatalari kontrol eder
#include <fstream>                     //dosya fonksiyonlari icin
#include <iostream>
using namespace std;

int main()
   {
   ifstream file;
   file.open("a:test.dat");

   if( !file )
      cout << "\nCan't open GROUP.DAT";
   else
      cout << "\nFile opened successfully.";
   cout << "\nfile = " << file;
   cout << "\nError state = " << file.rdstate();
   cout << "\ngood() = " << file.good();
   cout << "\neof() = " << file.eof();
   cout << "\nfail() = " << file.fail();
```



```
   cout << "\nbad() = " << file.bad() << endl;
   file.close();
   return 0;
   }
```

Bu program önce nesne dosyasının değerini kontrol eder. Eğer bu değer sıfır ise, dosya muhtemelen açılamayacaktır çünkü mevcut değildir. Bu durum söz konusu olduğunda **FERRORS** programının çıktı şöyle olur:

```
Can't open GROUP.DAT
file = 0x1c730000
Error state = 4
good() = 0
eof() = 0
fail() = 4
bad() = 4
```

rdstate() ile döndürülen hata durumu 4'tür. Bu, dosyanın mevcut olmadığını gösteren bit'tir; bu bit 1 değerini alır. Diğer bit'lerin tümüne 0 değeri verilir. **good()** fonksiyonu sadece bitlerin hiç biri ayarlanmamışsa (tümü 0 ise) 1 (**true**) döndürür. Bu nedenle, **good()** fonksiyonu bu programda 0 (**false**) döndürür. Dosyanın sonunda olmadığımız için **eof()** 0 döndürür. Hata oluştuğu için **fail()** ve **bad()** fonksiyonları sıfırdan farklı bir değer döndürürler.

Ciddi bir programda, işlerin beklenildiği gibi gitmesini garanti etmek için her I/O işleminden sonra bu fonksiyonların tümü veya bir kısmı kullanılmalıdır.

Üye Fonksiyonlarla Dosya Giriş/Çıkışı

Şu ana kadar dosya giriş/çıkışının ayrıntılarıyla uğraşma işini **main()** fonksiyona bıraktık. Daha sofistike sınıflar kullandığımızda, dosya giriş/çıkış işlemlerini sınıfın üye fonksiyonu yapmak gayet doğaldır. Bu bölümde, bahsettiğimiz şekilde çalışan iki program göreceğiz. İlk program, her nesnenin kendisini dosyaya yazıp dosyadan geri okuyabildiği sıradan üye fonksiyonları kullanır. İkinci örnekte ise, statik üye fonksiyonlarının bir sınıftaki tüm nesneleri tek adımda nasıl okuyup yazabildikleri anlatılmaktadır.

Kendilerini Okuyup Yazabilen Nesneler

Bazen bir sınıfın her bir üyesinin kendisini dosyaya yazıp dosyadan okuyabilmesine izin vermek anlamlı olur. Bu basit bir yaklaşımdır ve aynı anda yazılıp okunması gereken çok sayıda nesne olmadıkça iyi sonuç verir. Örneğimizde **person** sınıfına iki üye nesne ekleyeceğiz: **diskOut()** ve **diskIn()**. Bu fonksiyonlar bir **person** nesnesinin kendisini diske yazmasını ve diskten geri okumasını sağlarlar.

Olayı basitleştirmek için bazı varsayımlarda bulunduk. İlk olarak, sınıftaki tüm nesnelerin **PERSFILE.DAT** adlı tek bir dosyada tutulacağını varsaydık. İkinci olarak da, yeni nesnelerin her zaman dosyanın sonuna eklendiği varsayımında bulunduk.

diskIn() fonksiyonuna aktarılan bir argüman, dosyadaki herhangi bir kişiye ait veriyi okumamızı sağlar. Dosya sonunun ötesinde okuma girişimlerini önlemek için, dosyadaki kişi sayısını tutan **diskCount()** adlı bir statik üye fonksiyonu da ekledik. Programa veri girerken sadece soyadlarını kullanın. Boşluk karakteri kullanılmamalıdır. **REWOBJ** programının kaynak kodu şöyledir:

```
// rewobj.cpp
// disk giris/cikisi yapan kisi nesneleri
#include <fstream>                      //dosya akisları icin
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person                            //kisi sinifi
   {
   protected:
      char name[40];                    //ismi
      int age;                          //yasi
   public:
      void getData(void)                //kisinin verilerini al
         {
         cout << "\n   Enter last name: "; cin >> name;
         cout << "   Enter age: "; cin >> age;
         }
      void showData(void)               //kisinin verilerini goster
         {
         cout << "\n   Name: " << name;
         cout << "\n   Age: " << age;
         }
      void diskIn(int);                 //dosyadan oku
      void diskOut();                   //dosyaya yaz
      static int diskCount();           //dosyadaki kisilerin
                                        //sayisini dondur
   };
//--------------------------------------------------------------
void person::diskIn(int pn)             //kisi numarasi pn'yi
   {                                    //dosyadan oku
   ifstream infile;                     //akis olustur
   infile.open("PERSFILE.DAT", ios::binary);      //akisi ac
   infile.seekg( pn*sizeof(person) );             //dosya isaretcisini tasi
   infile.read( (char*)this, sizeof(*this) );     //bir kisinin verisini oku
   }
//--------------------------------------------------------------
void person::diskOut()                  //kisiyi dosya sonuna ekle
   {
   ofstream outfile;                    //akis olustur
                                        //akisi ac
   outfile.open("PERSFILE.DAT", ios::app | ios::binary);
   outfile.write( (char*)this, sizeof(*this) );   //akisa yaz
   }
//--------------------------------------------------------------
int person::diskCount()                 //dosyadaki kisi
   {                                    //sayisini dondur
   ifstream infile;
   infile.open("PERSFILE.DAT", ios::binary);
   infile.seekg(0, ios::end);           //sondan 0 byte geriye git
                                        //kisi sayisini hesapla
   return (int)infile.tellg() / sizeof(person);
   }
////////////////////////////////////////////////////////////////
int main()
   {
   person p;
   char ch;                             //bos bir kisi olustur

   do {
                                        //kisileri diske yaz
```



```
   cout << "Enter data for person:";
   p.getData();                         //veriyi al
   p.diskOut();                         //diske yaz
   cout << "Do another (y/n)? ";
   cin >> ch;
   } while(ch=='y');                    //kullanici 'n' girene dek

int n = person::diskCount();            //dosyada kac kisi var?
cout << "There are " << n << " persons in file\n";
for(int j=0; j<n; j++)                  //her biri icin,
   {
   cout << "\nPerson " << j;
   p.diskIn(j);                         //diskten kisileri oku
   p.showData();                        //kisiyi goster
   }
cout << endl;
return 0;
}
```

Örneğimizde fazla sürpriz bir unsur yoktur herhalde. Bu programdaki unsurların tamamına yakınını daha önceden görmüştünüz. Program aslında **DISKFUN** ile aynı şekilde çalışır. Ancak, bu sefer disk işlemlerinin ayrıntılarını **main()** programın görmediğine dikkat edin. Tüm bu işlemler artık **person** sınıfının içine gizlenmiştir. Her nesne bellekte farklı bir yerde olduğu için okuyup yazacağımız verinin yerini baştan bilme imkanımız yok. Ancak **this** adlı işaretçi, bir üye fonksiyondayken yerimizi her zaman bize bildirir. **read()** ve **write()** akış fonksiyonlarında okunacak ya da yazılacak nesnenin bellek adresi ***this**, boyutu da **sizeof(*this)**'dir.

Program başlamadan önce dosyada iki kişi bulunduğu varsayımıyla örnek bir çıkış aşağıdaki gibidir:

```
Enter data for person:
   Enter name:Acheson
   Enter age:63
Enter another (y/n)?y

Enter data for person:
   Enter name:Dulles
   Enter age:72
Enter another (y/n)?n

Person #1
   Name:Stimson
   Age:45
Person #2
   Name:Hull
   Age:58
Person #3
   Name:Acheson
   Age:63
Person #4
   Name:Dulles
   Age:72
```

Dosya isimlerini kodun içine yazdık. Böyle yapmayıp da dosya isimlerinin kullanıcı tarafından verilmesini isterseniz, statik bir üye değişken (örneğin **char fileName[]** adında) tanımlayabilir ve statik bir fonksiyonun bu değişkene değer atamasını sağlayabilirsiniz. Alterna-

tif olarak, her nesneye kendisi ile ilişkili dosyanın ismini verebilirsiniz ve bunun için statik olmayan bir fonksiyon kullanabilirsiniz.

Kendi Kendilerini Okuyup Yazan Sınıflar

Bellekte çok sayıda nesneniz olduğunu ve bunların tümünü diske yazmak istediğinizi varsayalım. **REWOBJ** örneğinde olduğu gibi her nesneye ait üye fonksiyonların tek tek dosya açıp bir nesneyi yazıp dosyayı kapatmaları hiç de verimli bir yöntem değildir. Dosyayı bir kez açtıktan sonra tüm nesneleri yazıp dosyayı ondan sonra kapatmak daha hızlı olur. Üstelik nesne sayısı arttıkça bu yöntemin kazancı da artar.

Statik Fonksiyonlar

Çok sayıda nesneyi tek defada yazmanın bir yolu, tek bir nesne değil sınıfın tümü üzerinde geçerli olan statik bir üye fonksiyonu kullanmaktır. Bu fonksiyon tüm nesneleri tek defada yazabilir. Peki böyle bir fonksiyon tüm nesnelerin yerini nereden bilecektir? Söz konusu fonksiyon statik veri olarak tutulabilecek bir işaretçi dizisine erişebilir. Her nesne oluşturulurken, o nesneyi gösteren bir işaretçi de bu diziye yerleştirilir. Statik bir veri üyesi de o ana kadar kaç nesne oluşturulduğunu tutar. Statik **write()** fonksiyonu dosyayı açabilir, diziyi takip eden bir döngü içinde tek tek her nesneyi dosyaya yazabilir ve en sonunda da dosyayı kapatabilir.

Türetilmiş Nesnelerin Boyutu

İşleri gerçekten ilginç bir noktaya getirmek için gelin bir varsayımda daha bulunalım ve bellekte tutulan nesnelerin farklı boyutlarda olduğunu farz edelim. Neden böyle bir şeye ihtiyaç duyulsun ki? Böyle bir durum, tipik olarak, bir temel sınıftan çeşitli sınıflar türetildiğinde ortaya çıkar. Örneğin, "Kalıtım" adlı Bölüm 9'da ele aldığımız **EMPLOY** programını inceleyelim. Burada, **manager**, **scientist** ve **laborer** sınıflarının temel sınıfını oluşturan bir **employee** sınıfımız var. Türetilmiş sınıflara ait nesneler farklı boyutlardadır. Çünkü farklı miktarda veri taşırlar. Özeline inersek; tüm çalışanlar için öngörülen isim ve personel koduna ek olarak yönetici için unvan ve golf kulübü aidatı vardır. Bilim adamı için de yayınladığı eserlerin sayısı eklenmiştir.

Bu türetilmiş sınıfların üçünün de (**manager**, **scientist**, **laborer**) yer aldığı bir listeyi basit bir döngü ve **ofstream** üyesi **write()** kullanarak yazmak istiyoruz. Ancak bu fonksiyonu kullanabilmek için fonksiyonun ikinci argümanını yani nesnenin boyutunu bilmemiz gerekir.

employee tipinden nesneleri gösteren bir işaretçi dizimiz olduğunu düşünün (adı **arrap**[] olsun). Bu işaretçiler her üç türetilmiş sınıfın nesnelerine de işaret edebilirler. (Türetilmiş sınıfları gösteren işaretçi dizilerine bir örnek olarak Bölüm 11'deki **VIRTPERS** programına bakın.) Sanal fonksiyonlarla aşağıdaki gibi ifadelere yer verebileceğimizi biliyoruz:

```
arrap[j]->putdata();
```

Burada temel sınıfın fonksiyonu yerine işaretçinin gösterdiği nesneye uygun olan **putdata()** versiyonu kullanılacaktır. Ama bir işaretçi argümanının boyutunu döndürmek için **sizeof()** fonksiyonunu da kullanabilir miyiz? Bir başka deyişle, aşağıdaki ifade uygun mudur?

```
ouf.write( (char*)arrap[j], sizeof(*arrap[j]) );  //yanlis
```

Uygun değildir, çünkü **sizeof()** bir sanal fonksiyon değildir. Bu fonksiyon, işaretçinin tipini değil de işaret edilen nesnenin tipini dikkate alması gerektiğini bilmez. Her zaman temel sınıfı nesnesinin boyutunu döndürür.

typeid() Fonksiyonunun Kullanımı

Elimizde sadece o nesneyi gösteren bir işaretçi varsa nesnenin boyutunu nasıl bulabiliriz? Bir yöntem, Bölüm 11'de gördüğümüz **typeid()** fonksiyonunun kullanılmasıdır. Bu fonksiyon aracılığı ile nesnenin sınıfını bulabiliriz ve **sizeof()**'la bu nesnenin ismini kullanabiliriz. **typeid()** fonksiyonunu kullanabilmek için *Run-Time Type Information (RTTI)* adlı bir derleyici seçeneğini aktif duruma getirmeniz gerekebilir. ("Microsoft Visual C++" adlı Ek C'de bahsedildiği üzere, mevcut Microsoft derleyicisinde buna gerek duyulur.)

Sıradaki örneğimizde **typeid()** fonksiyonunun nasıl kullanıldığını görüyoruz. Nesnenin boyutunu öğrenir öğrenmez, nesneyi diske yazmak için **write()** fonksiyonunu kullanabiliriz.

EMPLOY programına basit bir basit bir kullanıcı arabirimi ekledik ve nesneleri gösteren bir işaretçi dizisi kullanabilmek için üyelere özgü fonksiyonları sanal hale getirdik. Son bölümde ele aldığımız hata yakalama tekniklerini de programa ekledik.

Bu program biraz iddialı bir kod oldu ama bir yandan da dört başı mamur bir veritabanı uygulamasında kullanılabilecek tekniklerin pek çoğuna yer vermesi bakımından gayet yararlıdır. Örneğimiz nesne yönelimli programlamanın gerçek gücünü göstermektedir. Bir dosyaya farklı boyutlarda nesne başka nasıl yazılabilir ki? **EMPL_IO**'nun kaynak koduna bakalım:

```
// empl_io.cpp
// employee nesneleri uzerinde dosya giris/cikisi yapar
// farkli boyutlardaki nesneler uzerinde calisir
#include <fstream>                      //dosya-akis fonksiyonlari icin
#include <iostream>
#include <typeinfo>                     //typeid() icin
using namespace std;
#include <process.h>                    //exit() icin

const int LEN = 32;                     //soyadinin maksimum uzunlugu
const int MAXEM = 100;                  //maksimum personel sayisi
enum employee_type {tmanager, tscientist, tlaborer};
////////////////////////////////////////////////////////////////
class employee                          //employee sinifi
   {
   private:
      char name[LEN];                   //personelin ismi
      unsigned long number;             //personel kodu
      static int n;                     //mevcut personel sayisi
      static employee* arrap[];         //emps'e isaret eden isaretciler dizisi
   public:
      virtual void getdata()
         {
         cin.ignore(10, '\n');
         cout << "   Enter last name: ";    cin >> name;
         cout << "   Enter number: ";       cin >> number;
         }
      virtual void putdata()
         {
         cout << "\n   Name: " << name;
         cout << "\n   Number: " << number;
         }
      virtual employee_type get_type();     //tipi al
      static void add();                    //personel ekle
      static void display();                //tum personeli goster
      static void read();                   //disk dosyasindan oku
      static void write();                  //disk dosyasina yaz
```

```
    };
//-------------------------------------------------------------
//statik degiskenler
int employee::n;                        //mevcut personel sayisi
employee* employee::arrap[MAXEM];       //emps'e isaret eden isaretciler dizisi
////////////////////////////////////////////////////////////////
//yonetici sinifi
class manager : public employee
   {
   private:
      char title[LEN];                  //unvan: "genel muduru yardimcisi" vs..
      double dues;                      //golf kulubu aidati
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter title: ";              cin >> title;
         cout << "   Enter golf club dues: ";     cin >> dues;
         }
      void putdata()
         {
         employee::putdata();
         cout << "\n   Title: " << title;
         cout << "\n   Golf club dues: " << dues;
         }
   };
////////////////////////////////////////////////////////////////
//bilimadami sinifi
class scientist : public employee
   {
   private:
      int pubs;                         //yayinladigi eserlerin sayisi
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter number of pubs: "; cin >> pubs;
         }
      void putdata()
         {
         employee::putdata();
         cout << "\n   Number of publications: " << pubs;
         }
   };
////////////////////////////////////////////////////////////////
//isci sinifi
class laborer : public employee
   {
   };
////////////////////////////////////////////////////////////////
//personeli bellekteki listeye ekle
void employee::add()
   {
   char ch;
   cout << "'m' to add a manager"
        "\n's' to add a scientist"
        "\n'l' to add a laborer"
        "\nEnter selection: ";
   cin >> ch;
```



```
   switch(ch)
      {
      case 'm': arrap[n] = new manager;    break;   //belirlenen personel tipini olustur
      case 's': arrap[n] = new scientist;  break;
      case 'l': arrap[n] = new laborer;    break;
      default: cout << "\nUnknown employee type\n"; return;
      }
   arrap[n++]->getdata();                          //personel verisini kullanicidan al
   }
//-------------------------------------------------------------
//tum personeli goster
void employee::display()
   {
   for(int j=0; j<n; j++)
      {
      cout << (j+1);                               //sayiyi goster
      switch( arrap[j]->get_type() )               //tipi goster
         {
         case tmanager:   cout << ". Type: Manager";     break;
         case tscientist: cout << ". Type: Scientist";   break;
         case tlaborer:   cout << ". Type: Laborer";     break;
         default: cout << ". Unknown type";
         }
      arrap[j]->putdata();                         //personel verisini goster
      cout << endl;
      }
   }
//-------------------------------------------------------------
//bu nesnenin tipini dondur
employee_type employee::get_type()
   {
   if( typeid(*this) == typeid(manager) )
      return tmanager;
   else if( typeid(*this)==typeid(scientist) )
      return tscientist;
   else if( typeid(*this)==typeid(laborer) )
      return tlaborer;
   else
      { cerr << "\nBad employee type "; exit(1); }
   return tmanager;
   }
//-------------------------------------------------------------
//tum mevcut bellek nesnelerini dosyaya yaz
void employee::write()
   {
   int size;
   cout << "Writing " << n << " employees.\n";
   ofstream ouf;                                   //ofstream'i ikili olarak ac
   employee_type etype;                            //her personel nesnesinin tipi

   ouf.open("EMPLOY.DAT", ios::trunc | ios::binary);
   if(!ouf)
      { cout << "\nCan't open file\n"; return; }
   for(int j=0; j<n; j++)                          //her personel nesnesi icin:
      {                                            //nesnenin tipini al
      etype = arrap[j]->get_type();
                                                   //tipi dosyaya yaz
      ouf.write( (char*)&etype, sizeof(etype) );
      switch(etype)                                //boyutunu bul
```

```
            {
            case tmanager:    size=sizeof(manager);   break;
            case tscientist:  size=sizeof(scientist); break;
            case tlaborer:    size=sizeof(laborer);   break;
            }                              //personel nesnesini dosyaya yaz
         ouf.write( (char*)(arrap[j]), size );
         if(!ouf)
            { cout << "\nCan't write to file\n"; return; }
         }
      }
   //-------------------------------------------------------------
   //tum personelin verisini dosyadan bellege oku
   void employee::read()
      {
      int size;                        //personel nesnesinin boyutu
      employee_type etype;             //personelin turu
      ifstream inf;                    //ifstream'i ikili olarak ac
      inf.open("EMPLOY.DAT", ios::binary);
      if(!inf)
         { cout << "\nCan't open file\n"; return; }
      n = 0;                           //henuz bellekte personel yok
      while(true)
         {                             //siradaki personelin tipini oku
         inf.read( (char*)&etype, sizeof(etype) );
         if( inf.eof() )               //eof gelirse donguden cik
            break;
         if(!inf)                      //tipi okurken hata olursa
            { cout << "\nCan't read type from file\n"; return; }
         switch(etype)
            {                          //yeni personel olustur
            case tmanager:             //personelin turune gore
               arrap[n] = new manager;
               size=sizeof(manager);
               break;
            case tscientist:
               arrap[n] = new scientist;
               size=sizeof(scientist);
               break;
            case tlaborer:
               arrap[n] = new laborer;
               size=sizeof(laborer);
               break;
            default: cout << "\nUnknown type in file\n"; return;
            }                          //yeni personel icin dosyadan veri oku
         inf.read( (char*)arrap[n], size );
         if(!inf)                      //hata var ama eof hatasi degil
            { cout << "\nCan't read data from file\n"; return; }
         n++;                          //personel sayisi
         }     //while'in sonu
      cout << "Reading " << n << " employees\n";
      }
   ////////////////////////////////////////////////////////////////
   int main()
      {
      char ch;
      while(true)
         {
         cout << "'a' -- add data for an employee"
              "\n'd' -- display data for all employees"
```



```
              "\n'w' -- write all employee data to file"
              "\n'r' -- read all employee data from file"
              "\n'x' -- exit"
              "\nEnter selection: ";
      cin >> ch;
      switch(ch)
         {
         case 'a':                     //listeye personel ekle
            employee::add(); break;
         case 'd':                     //tum personeli goster
            employee::display(); break;
         case 'w':                     //tum personeli dosyaya yaz
            employee::write(); break;
         case 'r':                     //tum personeli dosyadan oku
            employee::read(); break;
         case 'x': exit(0);            //programdan cik
         default: cout << "\nUnknown command";
         }     //switch'in sonu
      }     //while'in sonu
   return 0;
   }     //main()'in sonu
```

Nesne Tipi İçin Kod Numarası

Bellekteki bir nesnenin sınıfını bulmayı biliyoruz. Peki, diskten okumaya hazırlandığımız verinin nesnesinin hangi sınıftan olduğunu nereden bulacağız? Bu işi yapmak için sihirli bir fonksiyon yoktur. Bir nesnenin verisini diske yazarken veriden hemen önce nesneyi belirten bir de kod numarası yazmamız gerekir (`enum` değişkeni `employee_type`). Daha sonra nesne verisini diskten okuyacağımız zaman önce bu kod numarasını okuruz ve belirilen tipten bir nesne oluştururuz. Son olarak da veriyi dosyadan alıp bu yeni nesneye aktarırız.

Ev Yapımı Nesnelerden Kaçınalım Lütfen

Bir nesnenin verisini herhangi bir yere okuyup, örneğin `char` tipi bir diziye aktarıp, sonra da o nesneyi gösteren bir işaretçiyi bu bölgeye işaret ettirme hevesine kapılabilirsiniz. Belki bir de `tip ataması` kullanıp yönteminizi doğrulama çabasına girebilirsiniz.

```
char someArray[MAX];
aClass* aPtr_to_Obj;
aPtr_to_Obj = reinterpret_cast<aClass*>(someArray);        //boyle yapmayin
```

Ancak bu şekilde bir nesne oluşturmuş olmazsınız. Yazdığınız ifade bir işaretçiyi sanki bir nesneyi gösteriyormuş gibi kullanmaya çalışır ki, bunun sonu felakettir. Bir nesne oluşturmanın sadece iki meşru yolu vardır. Nesneyi ya derleme sırasında açıkça oluşturursunuz:

```
aClass anObj;
```

Ya da programın çalışması sırasında `new` diyerek oluşturur ve nesnenin bellekteki yerini bir işaretçiye atarsınız:

```
aPtr_to_Obj = new aClass;
```

Bir nesneyi düzgün bir şekilde oluşturduğunuzda bir kurucu fonksiyon çağrılır. Bir kurucu fonksiyon tanımlamamış ve varsayılan kurucuyu kullanıyor olsanız bile bu gereklidir. Bir nesne, içinde veri bulunan bir bellek alanından ibaret değildir; aynı zamanda bazılarını sizin görmediğiniz üye fonksiyonlar da içerir.

EMPL_IO ile Etkileşim

Programın örnek bir kullanımı aşağıda verilmiştir. Örnekte, bir yönetici, bir bilim adamı ve bir işçiyi bellekte oluşturuyoruz. Bunları diske yazıyoruz, geri okuyoruz ve ekrana getiriyoruz. (Basit olsun diye birden fazla kelime içeren isim ve unvan kullanımına izin vermiyoruz; örneğin Vice President değil VicePresident diyoruz.)

```
'a' -- add data for an employee
'd' -- display data for all employees
'w' -- write all employee data to file
'r' -- read all employee data from file
'x' -- exit
Type selection: a
'm' to add a manager
's' to add a scientist
'l' to add a laborer
Type selection: m
   Enter last name: Johnson
   Enter number: 1111
   Enter title: President
   Enter golf club dues: 20000

'a' -- add data for an employee
'd' -- display data for all employees
'w' -- write all employee data to file
'r' -- read all employee data from file
'x' -- exit
Type selection: a
'm' to add a manager
's' to add a scientist
'l' to add a laborer
Type selection: s
   Enter last name: Faraday
   Enter number: 2222
   Enter number of pubs: 99

'a' -- add data for an employee
'd' -- display data for all employees
'w' -- write all employee data to file
'r' -- read all employee data from file
'x' -- exit
Type selection: a
'm' to add a manager
's' to add a scientist
'l' to add a laborer
Type selection: l
   Enter last name: Smith
   Enter number: 3333

'a' -- add data for an employee
'd' -- display data for all employees
'w' -- write all employee data to file
'r' -- read all employee data from file
'x' -- exit
Type selection: w
Writing 3 employees

'a' -- add data for an employee
```



```
'd' -- display data for all employees
'w' -- write all employee data to file
'r' -- read all employee data from file
'x' -- exit
Type selection: r
Reading 3 employees

'a' -- add data for an employee
'd' -- display data for all employees
'w' -- write all employee data to file
'r' -- read all employee data from file
'x' -- exit
Type selection: d
1. Type: Manager
   Name: Johnson
   Title: President
   Golf club dues: 20000
2. Type: Scientist
   Name: Faraday
   Number: 2222
   Number of publications: 99
3. Type: Laborer
   Name: Smith
   Number: 3333
```

Elbette, diske veri yazdıktan sonra programdan çıkabilirsiniz. Programı yeniden çalıştırdığınızda dosyayı okuyabilirsiniz ve bu sayede verilere yeniden kavuşabilirsiniz.

Bu program üzerinde birkaç kolay değişiklik yaparak bir personel kaydını silen, belli bir personelin kaydını diskten okuyan ve belli özelliklere sahip personeli arayan fonksiyonlar ekleyebilirsiniz.

Çıkarma ve Ekleme Operatörlerini Aşırı Yüklemek

Şimdi, akış bağlanatılı bir başka konuya geçelim: Çıkarma ve ekleme operatörlerini aşırı yüklemek. Bu, C++'ın güçlü bir özelliğidir. Bu özellik, kullanıcının tanımladığı tipler üzerinden yapılan I/O işlemlerini **int** ve **double** gibi temel veri tipleri ile aynı şekilde ele almanıza imkan verir. Örneğin, **crawdad** sınıfına ait **cd1** adında bir nesneniz varsa, bu nesneyi tıpkı bir temel veri tipiymiş gibi şu ifade ile ekranda gösterebilirsiniz:

```
cout << "\ncd1=" << cd1;
```

Çıkarma ve ekleme operatörlerini aşırı yükleyebiliriz. Böylece bu operatörler ekran ve klavye ile tek başına çalışabilirler (**cout** ve **cin**). Bir parça daha ihtimamla bu operatörleri disk dosyalarıyla da çalışabilecek şekilde aşırı yükleyebilirsiniz. Şimdi her iki durumla ilgili örneklere bakalım.

cout ve cin için Aşırı Yüklemek

Şimdi, **Distance** sınıfı için ekleme ve çıkarma operatörlerini aşırı yükleyen **ENGLIO** isminde bir örnek verelim. Bu sayede, ekleme ve çıkarma operatörleri **cout** ve **cin** ile çalışabilirler.

```
// englio.cpp
// asiri yuklenmis << ve >> operatorleri
```

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance                              //English Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0)     //kurucu (argumansiz)
         { }
                                            //kurucu (iki argumanli)
      Distance(int ft, float in) : feet(ft), inches(in)
         { }
      friend istream& operator >> (istream& s, Distance& d);
      friend ostream& operator << (ostream& s, Distance& d);
   };
//-----------------------------------------------------------
istream& operator >> (istream& s, Distance& d)     //mesafe olcusunu
   {                                               //kullanicidan
   cout << "\nEnter feet: "; s >> d.feet;          //asiri yuklenmis
   cout << "Enter inches: "; s >> d.inches;        //>> operatorunu kullanarak
   return s;                                       //al
   }
//-----------------------------------------------------------
ostream& operator << (ostream& s, Distance& d)     //Mesafeyi
   {                                               //asiri yuklenmis
   s << d.feet << "\'-" << d.inches << '\"';       //<< operatorunu
   return s;                                       //kullanarak
   }                                               //goster
////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist2;              //Mesafeleri tanimla
   Distance dist3(11, 6.25);           //dist3: tanimla ve ilk kullanima hazirla

   cout << "\nEnter two Distance values:";
   cin >> dist1 >> dist2;              //kullanicidan degerleri al
                                       //mesafeleri goster
   cout << "\ndist1 = " << dist1 << "\ndist2 = " << dist2;
   cout << "\ndist3 = " << dist3 << endl;
   return 0;
   }
```

Bu program, kullanıcının iki **Distance** değeri girmesini ister, sonra bu değerlerin ve program içinde atanmış olan bir başka değerin dökümünü alır. İşte örnek bir etkileşim:

```
Enter feet: 10
Enter inches: 3.5
Enter feet: 12
Enter inches: 6
dist1 = 10'-3.5"
dist2 = 12'-6"
dist3 = 11'-6.25"
```

Aşağıdaki gibi ifadeler kullanarak **Distance** nesnelerini herhangi bir veri tipi gibi ele almanın ne kadar rahat ve doğal olduğuna dikkat edin:


```
cin >> dist1 >> dist2;
```

ve

```
cout << "\ndist1=" << dist1 << "\ndist2=" << dist2;
```

<< ve **>>** operatörleri aynı şekilde aşırı yüklenir. Operatörler referans yoluyla bir **istream** (**>>** için) veya **ostream** (**<<** için) nesnesi döndürürler. Bu dönüş değerleri zincirleme kullanıma imkan verir. Operatörler, her ikisi de referans olarak aktarılan iki argüman alırlar. **>>** için birinci argüman bir **istream** nesnesidir (mesela, **cin**). **<<** için ise bir **ostream** nesnesidir (mesela, **cout**). İkinci argüman, görüntülenecek olan sınıfa ait bir nesnedir; mesela bu örnekte **Distance**. **>>** operatörü, birinci argümanla belirtilen akıştan girdiyi alır ve bunu ikinci argüman ile belirtilen nesnenin üye verisine yerleştirir. **<<** operatörü ikinci argümanla belirtilen nesneden verileri çıkarır ve bunu birinci argümanda belirtilen akışa gönderir.

istream ve **ostream** nesneleri operatörün sol tarafında yer aldığı için **<<** ve **>>** operatörleri **Distance** sınıfının **friend**'leri olmalıdır. (Bölüm 11'deki **friend** fonksiyonlar bahsine bakın.)

Aynı adımları takip ederek, ekleme ve çıkarma operatörlerini diğer sınıflar için de aşırı yükleyebilirsiniz.

Dosyalar İçin Aşırı Yüklemek

Bir sonraki örneğimiz, **<<** ve **>>** operatörlerinin **cout** ve **cin**'in yanı sıra dosya I/O işlemlerinde de çalıştırılabilmeleri için operatörleri **Distance** sınıfı içinde nasıl aşırı yükleyebileceğimizi gösteriyor.

```
// englio2.cpp
// asiri yuklenmis << ve >> operatorleri dosyalarla da calisabilir
#include <fstream>
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance                              //English Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0)     //kurucu (argumansiz)
         { }                                //kurucu (iki argumanli)
      Distance(int ft, float in) : feet(ft), inches(in)
         { }
      friend istream& operator >> (istream& s, Distance& d);
      friend ostream& operator << (ostream& s, Distance& d);
   };
//-----------------------------------------------------------
istream& operator >> (istream& s, Distance& d)     //Mesafeyi
   {                                               //dosyadan ya da
   char dummy;    //('), (-), ve (") icin          //klavyeden
                                                   //asiri yuklenmis
   s >> d.feet >> dummy >> dummy >> d.inches >> dummy;
                                                   //>> operatoru ile
   return s;                                       //al
   }
//-----------------------------------------------------------
ostream& operator << (ostream& s, Distance& d)     //Mesafeyi
                                                   //dosyaya
   {
```

```
    s << d.feet << "\'-" << d.inches << '\"';      //ya da ekrana
    return s;                                      //asiri yuklenmis
    }                                              //<< operatoru ile gonder
//////////////////////////////////////////////////////////////
int main()
   {
   char ch;
   Distance dist1;
   ofstream ofile;                                 //cikis akisini
   ofile.open("DIST.DAT");                         //olustur ve ac

   do {
      cout << "\nEnter Distance: ";
      cin >> dist1;                                //mesafeyi kullanicidan al
      ofile << dist1;                              //cikis akisina yaz
      cout << "Do another (y/n)? ";
      cin >> ch;
      } while(ch != 'n');
   ofile.close();                                  //cikis akisini kapat

   ifstream ifile;                                 //giris akisini
   ifile.open("DIST.DAT");                         //olustur ve ac

   cout << "\nContents of disk file is:\n";
   while(true)
      {
      ifile >> dist1;                              //mesafeyi akistan oku
      if( ifile.eof() )                            //EOF'ta dur
         break;
      cout << "Distance = " << dist1 << endl;      //mesafeyi goster
      }
   return 0;
   }
```

Aşırı yüklenmiş operatörlerin kendilerini asgari düzeyde değiştirdik. >> operatörü artık girdi istemiyle kullanıcıyı yönlendirmez; çünkü dosya için kullanıcıyı yönlendirmenin pek bir esprisi yoktur. Kullanıcının ayak ve inç değerlerini, çeşitli noktalama işaretleriyle birlikte nasıl girece-ğini tam olarak bildiğini varsayıyoruz. << operatörü değiştirilmemiştir. Program, kullanıcıdan girdi ister ve elde edilen her **Distance** değerini dosyaya yazar. Kullanıcı girdi işlemini tamamla-yınca program, dosyadaki değerlerin tümünü okur ve görüntüler. Programla örnek bir etkileşim şöyledir:

```
Enter Distance:3'-4.5"
Do another (y/n)? yes

Enter Distance:7'-11.25"
Do another (y/n)? yes

Enter Distance:11'-6"
Do another (y/n)? no

Contents of disk file is:
Distance=3'-4.5"
Distance=7'-11.25"
Distance=11'-6"
```


Uzaklıklar dosyada karakter karakter saklanır. Bu örnekte dosyanın içeriği şu şekilde olmalı-dır:

```
3'-4.5"7'-11.25"11'-6
```

Eğer kullanıcı, uzaklıkları doğru noktalama ile girmeyi beceremezse, bu değerler dosyaya doğru biçimde yazılmayacaktır ve dosya, << operatörü tarafından okunamayacaktır. Gerçek bir programda girdiler üzerinde hata kontrolü yapmak kaçınılmazdır.

Bir Akış Nesnesi Olarak Bellek

Belleğin bir bölümünü bir akış nesnesi olarak ele alabilir, tıpkı bir dosyaya veri ekler gibi verilerinizi ekleyebilirsiniz. Bu, çıktınızı belirli bir şekilde biçimlendirmeniz gerektiğinde (me-sela, ondalık noktanın sağında tam olarak iki basamak görüntülemek istediğinizde) işinize ya-rar. Fakat, bunun için bir de, girdi olarak bir karakter katarı gerektiren bir metin çıkış fonksiyo-nuna ihtiyacınız olacaktır. Bu, çıktı fonksiyonlarını GUI ortamlarında (mesela, Windows) çağırırken çok kullanılır, çünkü bu fonksiyonlar argüman olarak genellikle bir karakter katarı gerektirirler. (C programcıları, bu amaçla **.sprintf()** fonksiyonunun kullanıldığını hatırlayacaklardır.)

Bir grup akış sınıfı bu tür bellek içi biçimlendirmeyi gerçekleştirir. Çıktıyı belleğe almak için **ostream** sınıfından türetilen (diğer sınıfların yanı sıra) bir **ostrstream** sınıfı mevcuttur. Girdiyi bellekten almak için ise **istream** sınıfından türetilen **istrtream** sınıfı mevcuttur. Ayrıca, hem giriş hem çıkış yapan bellek nesneleri için de **iostream**'den türetilen **strstream** sınıfı vardır.

Daha çok **ostrstream**'i kullanmayı tercih edeceksiniz. Bir sonraki örneğimiz bunun nasıl çalıştığını gösterir. Bellekteki bir veri tamponu ile başlanır. Sonra, bellek tamponunu kullanarak bir **ostrstream** nesnesi tanımlanır. Bellek tamponunun boyutu akışın kurucu fonksiyonuna argüman olarak aktarılır. Artık biçimlendirilmiş metnin çıktısı, sanki bir akış nesnesiymiş gibi bellek tamponuna gönderilebilir. **OSTRSTR** programının listesi şöyledir:

```
// ostrstr.cpp
// bicimlendirilmis veriyi bellege yazar
#include <strstream>
#include <iostream>
#include <iomanip>                          //setiosflags() icin
using namespace std;
const int SIZE=80;                          //bellekte olusturulan tamponun boyutu

int main()
   {
   char ch = 'x';                           //sinama verileri
   int j = 77;
   double d = 67890.12345;
   char str1[] = "Kafka";
   char str2[] = "Freud";
   char membuff[SIZE];                      //bellekteki tampon
   ostrstream omem(membuff, SIZE);          //akis nesnesi olustur

   omem << "ch=" << ch << endl              //bicimlendirilmis veriyi
        << "j=" << j << endl                //nesneye gir
        << setiosflags(ios::fixed)          //ondalik nokta ile bicimlendir
        << setprecision(2)                  //noktanin saginda iki hane olsun
        << "d=" << d << endl
```

```
        << "str1=" << str1 << endl
        << "str2=" << str2 << endl
        << ends;                          //tamponu '\0 ' ile sonlandir
    cout << membuff;                      //bellekteki tamponu goster
    return 0;
    }
```

Programı çalıştırdığınızda, **membuf** biçimlendirilmiş metin ile dolacaktır:

```
ch=x\nj=77\nd=67890.12\nstr1=Kafka\nstr2=Freud\n\0
```

Kayan noktalı sayıları sıradan yöntemlerle biçimlendirebiliriz. Bu programda, **ios::fixed** ile sabit bir ondalık biçim (üstel biçim yerine) belirlenir ve ondalık noktanın sağında iki basamak olacağı belirtilir. **ends** manipülatörü, bir **EOF** sağlamak amacıyla karakter katarının sonuna '**\n**' karakterini ekler. Bu tamponun içeriğini **cout** kullanarak sıradan bir yoldan görüntülemek, programın çıktısını üretir:

```
ch=x
j=77
d=67890.12
str1=Kafka
str2=Freud
```

Bu örnekte, program tamponun içeriğini, yalnızca bu içeriğin neye benzediğini göstermek için görüntüler. Genellikle bu tür biçimlendirilmiş veriler çok daha karmaşık bir kullanım gerektirir.

Komut Satırı Argümanları

Daha önce MS-DOS kullandıysanız, programları çalıştırmak için kullanılan komut satırı argümanlarına muhtemelen aşinasınızdır. Bunlar genellikle bir veri dosyasının ismini bir uygulamaya aktarmak için kullanılır. Örneğin, bir kelime işlemci programını, programın üzerinde çalışacağı belge ile aynı anda çağırabilirsiniz:

```
C>wordproc afile.doc
```

Burada **afile.doc** bir komut satır argümanıdır. Bir C++ programının komut satırı argümanlarını okumasını nasıl sağlayabiliriz? İşte bir örnek, **COMLINE**. **COMLINE**, tuşlayabildiğiniz kadar çok sayıda komut satırı argümanını okur ve görüntüler (argümanlar boşluk ile ayrılmalıdır):

```
// comline.cpp
// komut satiri argumanlarinin kullanimini gosterir
#include <iostream>
using namespace std;

int main(int argc, char* argv[] )
   {
   cout << "\nargc=" << argc << endl;        //arguman sayisi

   for(int j=0; j<argc; j++)                 //argumanlari goster
   cout << "Argument " << j << " = " << argv[j] << endl;
```



```
   return 0;
   }
```

İşte programla gerçekleştirilebilecek örnek bir etkileşim:

```
C:\C++BOOK \Chap12>comline uno dos tres

argc = 4
Argument 0 = C:\CPP \CHAP12 \COMLINE.EXE
Argument 1 = uno
Argument 2 = dos
Argument 3 = tres
```

Komut satırı argümanlarını okumak için **main()** fonksiyonunun kendisine (unutmayın ki, **main()** de bir fonksiyondur!) iki argüman verilmelidir. İlk argüman olan **argc** (*argument count* anlamında), komut satırı argümanlarının toplam sayısını simgeler. İlk komut satırı argümanı daima şu an kullanılan programın dizin yolunu (path) gösterir. Kalan argümanlar ise kullanıcının girdiği argümanlardır; argümanlar boşluk karakteri ile birbirinden ayrılırlar. Önceki örnekte, bu argümanlar **uno**, **dos** ve **tres**'tir.

Sistem, komut satırı argümanlarını bellekte birer karakter katarı olarak saklar ve bu karakter katarlarına işaret eden işaretçilerden bir dizi oluşturur. Örnekte, bu dizi **argv** (*argument values* anlamında) olarak adlandırılır. Dizinin içindeki karakter katarlarına tek tek ilgili işaretçiler yardımıyla erişilir. Yani, birinci karakter katarı **argv[0]**, ikinci **argv[1]**'dir vs . **COMLINE** sırayla argümanlara erişir ve **argc**'yi (komut satırı argümanlarının sayısını) üst limit olarak kullanarak bir **for** döngüsü içinde bu argümanların dökümünü alır.

main()'in argümanları olarak belirli bir isim olan **argc** ve **argv**'yi kullanmak zorunda değilsiniz; fakat, bunlar öylesine yaygındır ki başka bir isim kullanmak, derleyici haricinde herkesin şaşırmasına neden olacaktır.

Şimdi, yararlı bir iş için komut satırı argümanı kullanan bir programa göz atalım. Bu program, ismi, kullanıcı tarafından komut satırından girilen bir metin dosyasının içeriğini görüntüler. Yani, program DOS komutlarından **TYPE**'ı taklit eder. **OTYPE** programının listesi şöyledir:

```
// otype.cpp
// TYPE komutunu taklit eder
#include <fstream>                        //dosya fonksiyonlari icin
#include <iostream>
using namespace std;
#include <process.h>                      //exit() icin

int main(int argc, char* argv[] )
   {
   if(argc != 2)
      {
      cerr << "\nFormat: otype filename";
      exit(-1);
      }
   char ch;                               //okunacak karakter
   ifstream infile;                       //giris icin dosya olustur
   infile.open( argv [1] );               //dosyayi ac
   if( !infile )                          //hata kontrolu yap
      {
      cerr << "\nCan't open " << argv[1];
      exit(-1);
```

```
    }
  while( infile.get(ch) != 0 )          //bir karakter oku
    cout << ch;                         //karakteri goster
  return 0;
  }
```

Program önce kullanıcının doğru sayıda komut satırı argümanı girip girmediğini anlamak için argüman sayısını kontrol eder. `OTYPE.EXE`'nin kendisinin bulunduğu dizin yolunun isminin (pathname), her zaman ilk argüman olduğunu hatırınızdan çıkarmayın. İkinci argüman, görüntülenecek dosyanın ismidir. Kullanıcı, programı çağırdığı sırada bu dosya ismini girmiş olması gerekir:

```
C>otype ichar.cpp
```

Yani, komut satırı argümanlarının toplam sayısı 2'ye eşit olmalıdır. Eğer değilse, kullanıcı muhtemelen programı nasıl kullanacağını anlamamıştır; program sorunu açıklamak için `cerr` üzerinden bir hata mesajı gönderir.

Argümanların sayısı doğruysa program, ikinci komut satırı argümanında (`argv[1]`) ismi bulunan dosyayı açmaya çalışır. Eğer dosya yine açılmazsa program hata sinyali verir. Son olarak, program bir `while` döngüsü içinde dosyadan tek tek karakterleri okur ve ekrana yazar.

0 karakter değeri dosyanın sonuna (`EOF`) işaret eder. Bu `EOF`'ı kontrol etmenin bir başka yoludur. Ayrıca, daha önceden yaptığımız gibi, dosya nesnesinin kendisini de kullanabilirsiniz:

```
while( infile )
   {
   infile.get(ch);
   cout << ch;
   }
```

Ayrıca, bu `while` döngüsünün tamamını aşağıdaki ifade ile değiştirebilirsiniz:

```
cout << infile.rdbuf();
```

Bu kullanımı daha önce `ICHAR2` programında görmüştük.

Yazıcı Çıktısı

Verileri yazıcıya göndermek için konsol modundaki programları kullanmak oldukça kolaydır. İşletim sistemi tarafından donanım aygıtları için birkaç özel dosya ismi tanımlanır. Bu, aygıtları birer dosya gibi ele almayı mümkün kılar. Tablo 12.11 C++ içinde tanımlanmış isimleri gösterir.

TABLO 12.11: Donanım Aygıtlarının İsimleri

İsim	Aygıt
`con`	Konsol (klavye ve ekran)
`aux` veya `com1`	Birinci seri port
`com2`	İkinci seri port

TABLO 12.11: Donanım Aygıtlarının İsimleri

İsim	Aygıt
`prn` veya `lpt1`	Birinci paralel port
`lpt2`	İkinci paralel port
`lpt3`	Üçüncü paralel port
`nul`	Dublör (mevcut olmayan) aygıt

Bir çok sistemde yazıcı, ilk paralel porta bağlanır. Bu nedenle, yazıcı için dosya ismi `prn` veya `lpt1` olmalıdır. (Eğer sizin sisteminiz farklı kurulmuş ise bunu uygun bir isimle değiştirebilirsiniz.)

Aşağıdaki `EZPRINT` adlı program, yazıcıya bir karakter katarı ve bir de sayı gönderir. Programda, ekleme operatörü yardımıyla biçimlendirilmiş bir çıktı kullanılır.

```
// ezprint.cpp
// yaziyiya basit cikis yapilmasi
#include <fstream>                      //dosya akislari icin
using namespace std;

int main()
   {
   char* s1 = "\nToday's winning number is ";
   int n1 = 17982;

   ofstream outfile;                    //dosya olustur
   outfile.open("PRN");                 //dosyayi yazici icin ac
   outfile << s1 << n1 << endl;         //yaziciya veri gonder
   outfile << '\x0C';                   //sayfayi cikarmak icin formfeed
   return 0;
   }
```

Yazıcıya bu şekilde istediğiniz miktarda biçimlendirilmiş çıktı gönderebilirsiniz. `'\x0C'` karakteri, sayfanın yazıcıdan çıkarılmasını sağlar.

Bir sonraki örnek olan `OPRINT`, komut satırında belirtilen disk dosyasının içeriğini yazıcıya gönderir. Program, bu veri transferi için karakter karakter kopyalama yöntemini kullanır.

```
// oprint.cpp
// PRINT komutunu taklit eder
#include <fstream>                      //dosya fonksiyonlari icin
#include <iostream>
using namespace std;
#include <process.h>                    //exit() icin

int main(int argc, char* argv[] )
   {
   if(argc != 2)
      {
      cerr << "\nFormat: oprint filename";
      exit(-1);
      }
   char ch;                             //okunacak karakterler
   ifstream infile;                     //giris icin dosya olustur
   infile.open( argv[1] );              //dosyayi ac
```

```
if( !infile )                          //hata kontrolu yap
   {
   cerr << "\nCan't open " << argv[1];
   exit(-1);
   }
ofstream outfile;                      //dosya olustur
outfile.open("PRN");                   //dosyayi yazici icin ac
while(infile.get(ch) != 0)             //bir karakter oku
   outfile.put(ch);                    //yaziciya karakter yaz
outfile.put('\x0C');                   //sayfayi cikarmak icin formfeed
return 0;
}
```

Herhangi bir metin dosyasını, mesela `.CPP` uzantılı kaynak dosyalarınızın herhangi birini, yazıcıda yazdırmak için bu programı kullanabilirsiniz. Program, aynı DOS `PRINT` komutu gibi çalışıyor. `OTYPE` örneği gibi, bu program da komut satırı argümanlarının doğru sayıda olup olmadığını ve belirtilen dosyanın başarılı bir şekilde açılıp açılmadığını kontrol eder.

Özet

Bu bölümde kısaca, akış sınıflarının hiyerarşisini inceledik ve çeşitli I/O hatalarını nasıl ele alabileceğimizi gösterdik. Sonra, dosya giriş/çıkışlarını çeşitli yollardan nasıl gerçekleştirebileceğimizi gördük. C++'taki dosyalar, çeşitli sınıfların nesneleri ile ilişkilendirilir. Çoğunlukla çıktı için `ofstream`, girdi için `ifstream` ve hem girdi hem çıktı için ise `fstream` kullanılır. I/O işlemlerini gerçekleştirmek için bu akışların üye fonksiyonları veya temel sınıflarından yararlanılır. `<<`, `put()` ve `write()` gibi operatör ve fonksiyonlar çıktı için; `>>`, `get()` ve `read()` ise girdi için kullanılır.

`read()` ve `write()` fonksiyonları ikili modda çalışır; bu sayede, nesnelerin bütünü, ne tür bir veri içerdiklerinden bağımsız olarak, diske kaydedilebilir. Tek bir nesnenin yanı sıra diziler veya birçok nesne içeren diğer yeri yapıları da diskte saklanabilir. Dosya giriş/çıkışları üye fonksiyonlar yardımıyla ele alınabilir. Bu işlem, nesnelerin tek tek kendi sorumluluklarında olabilir ya da sınıf, statik üye fonksiyonlarını kullanarak I/O işlemlerini kendisi yürütebilir.

Her dosya işleminden sonra hata durumlarının kontrolü yapılmalıdır. Bir hata durumunda dosya nesnesinin kendisi 0 değerini alır. Ayrıca, belirli tipteki hataları saptamak için birkaç üye fonksiyon da kullanılabilir. Çıkarma operatörünün (`>>`) ve ekleme operatörünün (`<<`) program tarafından tanımlanan veri tipleriyle çalışabilmeleri için işlevleri artırılabilir. Bellek, bir akış gibi ele alınabilir; sanki bir dosyaymış gibi belleğe veri gönderilebilir.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Bir C++ akışı
   a. bir fonksiyon yardımıyla gerçekleştirilen akış kontrolüdür.
   b. bir yerden başka bir yere veri akışıdır.
   c. belirli bir sınıf ile ilişkilidir.
   d. bir dosyadır.
2. Bir çok akış sınıfının temel sınıfı _____ sınıfıdır.
3. Dosya giriş/çıkışlarında en yaygın olarak kullanılan üç akış sınıfın isimlerini belirtin.
4. `ofstream` sınıfının `salefile` adında bir nesnesini tanımlayacak bir ifade yazın. Bunu `SALES.JUN` isimli bir dosya ile ilişkilendirin.

5. Doğru/Yanlış: Bazı akışlar girdi ile, bazıları çıktı ile çalışır.
6. `foobar` adındaki bir `ifstream` nesnesinin, dosyanın sonuna gelip gelmediğini ya da bir hatayla karşılaşıp karşılaşmadığını kontrol eden bir `if` ifadesi yazın.
7. Ekleme operatörünü (`<<`) kullanarak `ofstream` sınıfının bir nesnesine metin çıktısı gönderebiliriz çünkü
   a. `oftsream` sınıfı bir akıştır.
   b. ekleme operatörü her türlü sınıfla çalışır.
   c. aslında `cout`'a çıkış yapılır.
   d. ekleme operatörünün `ofstream` içinde aşırı yüklenmiştir.
8. `ofstream` sınıfına ait `fileOut` adında bir nesneye tek bir karakter yazan bir ifade yazın.
9. `float` tipinde değişkenler içeren verileri, `ofstream` tipinde bir nesneye yazmak için
   a. ekleme operatörünü kullanmalısınız.
   b. `seekg()` fonksiyonunu kullanmalısınız.
   c. `write()` fonksiyonunu kullanmalısınız.
   d. `put()` fonksiyonunu kullanmalısınız.
10. `ifile` adında bir `ifstream` nesnesinin içeriğini `buff` adında bir diziye okuyan bir ifade yazın.
11. `app` ve `ate` gibi mod bitleri
    a. `ios` sınıfı içinde tanımlanır.
    b. bir dosyanın okuma veya yazma için açıldığını belirtebilir.
    c. `put()` ve `get()` fonksiyonları ile çalışır.
    d. bir dosyayı açmak için çeşitli yollar sunarlar.
12. Dosyalar söz konusu olduğunda *mevcut konumun* (*current position*) ne anlama geldiğini tanımlayın.
13. Doğru/Yanlış: Bir dosya işaretçisi her zaman dosyanın adresini içerir.
14. `f1` adındaki bir akış nesnesi içinde mevcut konumu 13 byte geriye kaydıran bir ifade yazın.
15. Şu ifade

```
f1.write( (char*)&obj1, sizeof(obj1) );
```

    a. `obj1`'in üye fonksiyonlarını `f1`'e yazar.
    b. `obj1`'in verilerini `f1`'e yazar.
    c. `obj1`'in üye fonksiyonlarını ve verilerini `f1`'e yazar.
    d. `obj1`'in adresini `f1`'e yazar.
16. Komut satırı argümanları
    a. askeri anlaşmazlıklardır.
    b. komut satırında program isminden sonra yazılırlar.
    c. argümanlar sayesinde `main()`'e ulaştırılırlar.
    d. sadece disk dosyaları tarafından erişilebilirler.
17. `cin` ile birlikte kullanıldığında `skipws` işareti neyi gerçekleştirir?
18. Komut satırı argümanlarına imkan tanıyan bir `main()` için deklaratör yazın.
19. Konsol modundaki programlarda yazıcı _____ öntanımlı dosya ismi kullanılarak erişilebilir.
20. `istream` sınıfının bir nesnesinden çıktı alan ve bunu `Sample` sınıfının bir nesnesinin içeriği olarak gösteren, aşırı yüklenmiş bir `>>` operatörü için deklaratör yazın.

Alıştırmalar

Yıldızla işaretli soruların cevaplarını Ek G'de bulabilirsiniz.

1. Bölüm 6'da "Nesneler ve Sınıflar" bahsinde **ENGLCON** örneğindeki **Distance** sınıfı ile başlayın. Bu bölümde **DISKFUN** örneğindeki döngünün bir benzerini kullanarak kullanıcıdan **Distance** değerinde bir sayı alın ve bunu bir disk dosyasına yazın. Bu sayıları dosyanın içindeki mevcut değerlere (eğer varsa) ekleyin. Kullanıcı daha fazla değer girilmeyeceğini bildirince dosyayı okuyun ve değerlerin tümünü ekranda gösterin.*
2. DOS **COPY** komutunun işlevini gerçekleştiren bir program yazın. Yani, program bir metin dosyasının (mesela, **.CPP** uzantılı herhangi bir dosya) içeriğini bir başka dosyaya kopyalamalı. Programı iki komut satırı argümanı – kaynak dosya ve hedef dosya – ile şu şekilde çağırın:*

```
C>ocopy srcfile.cpp destfile.cpp
```

   Programın içinde, kullanıcının doğru sayıda komut satırı argümanı girip girmediğini ve belirtilen dosyaların açılıp açılamayacağını kontrol edin.
3. İsmi komut satırından girilen bir programın byte cinsinden uzunluğunu döndüren bir program yazın:*

```
C>filesize program.ext
```

4. Bir döngü içinde kullanıcının bir *isim verisi* girmesini sağlayın. İsim verisi şu öğeleri içermeli: Ad, göbek adı, soyadı ve çalışma numarası (**unsigned long** tipinde). Sonra, ekleme operatörü (**<<**) ile birlikte biçimlendirilmiş I/O kullanarak bu dört veri öğesini bir **ofstream** nesnesine yazın. Karakter katarlarının boşluk karakter veya başka bir boşlukla sona ermesi gerektiğini unutmayın. Kullanıcı daha fazla isim verisi girilmeyeceğini bildirince **ofstream** nesnesini kapatın, bir **ifstream** nesnesi açın, dosyadaki tüm verileri okuyup ekrana yazdırın ve programı sona erdirin.
5. Saat, dakika ve saniye için tamsayı üye değerleri içeren (sırasıyla, **hours**, **minutes** ve **seconds**) bir **time** sınıfı tanımlayın. Kullanıcıdan bir saat değeri alan **get_time()** adında bir üye fonksiyon oluşturun. Bir de, saati 12:59:59 biçiminde gösterecek bir **put_time()** fonksiyonu tanımlayın. Kullanıcı hatalarını en aza indirmek için **get_time()** fonksiyonuna hata kontrolünü de ekleyin. Bu fonksiyon, saat, dakika ve saniyeyi ayrı ayrı istemeli ve her birini **ios** hata durum işaretleri açısından kontrol etmeli. Ayrıca, doğru aralıkta olup olmadıklarını sınamalı. Saat 0 ve 23 arasında, dakika ve saniye 0 ile 59 arasında olmalı. Bu değerleri karakter katarı olarak alıp sonra çevirmeyin; doğrudan tamsayı olarak okuyun. **ENGL_IO** örneği birden fazla ondalık nokta ile karşılaşırsa bunları ayıklayabilir. Bu alıştırmadaki örnekte ise birden fazla ondalık noktayı ayıklama imkanınız olmayacak. Ama bunun önemli olmadığını varsayıyoruz.
   main() içinde, **get_time()**'ı kullanarak kullanıcıdan tekrar tekrar **time** değerleri almak için bir döngü kullanın; bu değerleri **put_time()** ile ekranda gösterin. İşte şöyle:

```
Enter hours: 11
Enter minutes: 59
Enter seconds: 59
time = 11:59:59
```



```
Do another (y/n)? y
Enter hours: 25
Hours must be between 0 and 23
Enter hours: 1
Enter minutes: 10
Enter seconds: five
Incorrect seconds input
Enter seconds: 5
time = 1:10:05
```

6. Dördüncü alıştırmadaki verileri (ad, göbek adı, soyadı ve çalışma numarası) kullanarak **name** adında bir sınıf oluşturun. Bu sınıf için, **ofstream** kullanarak nesne verilerini disk dosyasına yazan ve dosyadan okuyan üye fonksiyonları tanımlayın. **<<** ve **>>** operatörlerinden yararlanarak biçimlendirilmiş veriler kullanın. Okuma ve yazma ile ilgili üye fonksiyonları kendi kendilerini içermelidir: İlgili akışı açmak ve okumak veya yazmak için gerekli olan ifadeleri içermeleri gerekir.
   Yazma fonksiyonu basitçe, verileri dosyanın sonuna ekleyebilir. Okuma fonksiyonu, okuyacağı kaydı seçmek için bir yönteme gerek duyacaktır. Bunu gerçekleştirmenin bir yolu, bu fonksiyonu, kayıt numarasını simgeleyen bir parametre ile çağırmak olabilir. Okuma fonksiyonu, hangi kaydı okuması gerektiğini öğrenince bu kaydı nasıl bulabilir? **seekg()** fonksiyonunu kullanabileceğinizi düşünüyor olabilirsiniz, fakat bu pek fazla işe yaramaz çünkü biçimlendirilmiş I/O kayıtlarının her biri (karakter katarlarındaki karakter sayısına ve tamsayılardaki basamak sayısına bağlı olarak) farklı uzunluklardadır. Bu nedenle, istediğiniz kayda ulaşana kadar aslında kayıtları tek tek okumanız gerekecektir.
   main() içinde, kullanıcının birkaç tane nesne için veri girmesini sağlamak için bu üye fonksiyonları çağırın. Nesneler kullanıcıdan alınırken dosyaya yazılmalı. Program sonra bu verilerin tümünü dosyadan okuyarak ekranda göstersin.
7. Bir nesneye dosya akış giriş/çıkışları eklemenin bir başka yolu, dosya akışının kendisini nesnenin bir statik üyesi yapmaktır. Buna ne gerek var? Akışı, bir sınıfı oluşturan nesneler ile tek tek ilişkilendirmek yerine sınıfın bütünüyle ilişkilendirerek düşünmek kavramsal açıdan genellikle çok daha kolaydır. Ayrıca, akışı sadece bir kez açmak ve gerektikçe nesneleri okuyup yazmak çok daha verimlidir. Örneğin, dosya bir kez açılırsa, okuma fonksiyonunun her çağrılışında dosya içindeki bir sonraki veri döndürülebilir. Dosya işaretçisi dosya içinde otomatik olarak ilerleyecektir, çünkü dosya okumalar arasında kapatılmamıştır.
   Bir **fstream** nesnesini isim sınıfının bir statik veri öğesi olarak kullanmak için 4. ve 6. alıştırmalardaki programı yeniden yazın. Bu alıştırmalardaki işlevselliği aynen koruyun. Bu akışı açmak için bir statik fonksiyon; dosya işaretçisini dosyanın başına işaret edecek şekilde ayarlamak için bir başka statik fonksiyon yazın. Yazma işlemini tamamladığınızda kayıtların tümünü dosyadan geri okumak istediğinizde, dosya işaretçisini dosya başına taşıyan bu fonksiyonu kullanabilirsiniz.
8. Bölüm 10'da "İşaretçiler" bahsindeki **LINKLIST** programı ile başlayarak, kullanıcıya dört seçenek sunan bir program yazın. Seçenekler bir tuşa basarak seçilebilir.
   - Bellekteki listeye bir bağlantı eklemek (bir tane tamsayıdan ibaret olan veriyi kullanıcı temin eder)
   - Bellekteki bağlantıların tümünün verilerini görüntülemek
   - Bağlantıların tümünün verilerini bir disk dosyasına yazmak (gerekirse dosyayı keserek veya ilk kez oluşturarak)

- Verilerin tümünü dosyadan geri okumak ve bu verileri saklamak için yeni bir bağlı liste oluşturmak

İlk iki seçenek, önceden **LINKLIST**'te uygulanmış olan üye fonksiyonları kullanabilir. Disk dosyasından okuma ve dosyaya yazmak için ilgili fonksiyonları yazmanız gerekecek. Tüm okuma ve yazma işlemleri için aynı dosyayı kullanabilirsiniz. Dosya sadece verileri saklamalıdır. İşaretçi içeriğini saklamanın pek bir esprisi yoktur, çünkü liste geri okunduğunda bu işaretçilerin içeriği muhtemelen alakasız olacaktır.

9. "Operatörlerin Aşırı Yüklenmesi" adlı Bölüm 8'deki 7. alıştırma ile başlayın. Dört fonksiyonlu hesap makinesindeki **frac** sınıfı için ekleme (<<) ve çıkarma (>>) operatörlerinin işlevlerini artırın. Operatörleri zincirleme kullanabileceğinizi dikkate alın; yani, kullanıcının bir kesir, bir operatör ve bir kesir daha girmesini istemek tek bir ifade gerektirmelidir:

```
cin >> frac1 >> op >> frac2;
```

10. Dokuzuncu alıştırmadaki **frac** sınıfının çıkarma (>>) operatörüne hata kontrolü ekleyin. Hata kontrolü yapılacaksa verileri 9. alıştırmada gösterildiği gibi tek bir ifade ile almak yerine kullanıcıyı önce ilk kesir, sonra operatör ve tekrar ikinci kesir için yönlendirmek muhtemelen daha uygun olur. Programla etkileşimin içine hata mesajları katıldığında bu yaklaşım biçimin daha anlaşılır olmasını sağlar.

```
Enter first fraction: 5/0
Denominator cannot be 0
  Enter fraction again: 5/1
Enter operator (+,-,*,/): +
Enter second fraction: one third
Input error
  Enter fraction again: 1/3
Answer is ---------- 16/3
Do another (y/n)?
```

Bu örnek etkileşimden de anlaşılacağı üzere, **ios** hata işaretlerini ve paydanın 0 olup olmamasını kontrol etmelisiniz. Eğer hata varsa kullanıcıyı kesri yeniden girmesi için yönlendirin.

11. Son olarak Bölüm 11'de 5. alıştırmada karşılaştığımız **bMoney** sınıfı ile başlayın. **bMoney** miktarları üzerinde I/O gerçekleştirmek için ekleme (<<) ve çıkarma (>>) operatörlerinin işlevlerini artırın. **main()** içinde birkaç örnek I/O gerçekleştirin.
12. Bu bölümdeki **EMPL_IO** programına bir beceri ekleyin: Program, belirli bir personel numarasına sahip bir nesneyi bulmak için disk dosyası içindeki personel nesnelerinin tümü üzerinde arama yapsın. Program eğer böyle bir nesne bulursa bu personele ait bilgileri ekranda göstermeli. Kullanıcı bu arama [**find()**] fonksiyonunu 'f' karakterini tuşlayarak çağırabilir. Fonksiyon çağrılınca kullanıcıdan personel numarasını sormalıdır. Bu fonksiyonun statik mi, sanal mı yoksa başka bir şey mi olması gerektiğine karar verin. Bu arama ve ekranda gösterme işlemleri bellekteki veri ile karışmamalıdır.

NOT **EMPLD_IO programı ile üretilen bir dosyayı okumaya çalışmayın. Yeni programdaki find() üye fonksiyonundan dolayı sınıflar aynı değildir; bu nedenle, bölüm içinde anlatıldığı gibi, eğer veriler karışırsa sonuç bir felaket olabilir. Derleyicinizin RTTI seçeneğini aktif hale getirmeniz gerekebilir. İlgili durumlarda "Microsoft Visual C++" adlı Ek C veya "Borland C++ Builder" adlı Ek D bölümlerine bakın.**

ONÜÇÜNCÜ BÖLÜM

13

BİRDEN FAZLA DOSYA KULLANAN PROGRAMLAR

Birden Fazla Dosya İçeren Programları Niçin Kullanırız?

Birden Fazla Dosya Kullanan Bir Program Oluşturmak

Dosyalar Arası Haberleşme

Çok Uzun Bir Sayı Sınıfı

Bir Yükselen Asansör Simülasyonu

Önceki bölümlerde, bir C++ programının çeşitli parçalarının – mesela, sınıf deklarasyonları, üye fonksiyonlar ve `main()` fonksiyonu – nasıl birleştirildiğini gördük. Bununla birlikte, o bölümlerdeki programların tümü tek bir dosyadan ibaretti. Şimdi, programlarımıza birden fazla dosyayı dahil ederek program organizasyonuna daha geniş açıdan bakalım. Dosyalar arası haberleşmenin nasıl yürütüldüğünü ve başlık dosyalarının bu duruma nasıl uyduğunu göreceğiz.

Bu bölümde birden fazla dosya kullanan programları genel olarak ele almanın yanı sıra daha uzun ve daha iddialı birkaç uygulamayı da tanıtacağız. Bu programlardaki amacımız, programların işleyişini kesinlikle tüm ayrıntılarıyla anlamanızı sağlamak değil, fakat büyük programların elemanlarının nasıl birbiriyle bağlantılı olduğunu genel olarak anlamanızı sağlamaktır. Bu programlar ayrıca sınıfların, şimdiye kadar gördüğümüz kısa örnekler yerine daha gerçekçi uygulamalarda nasıl kullanılabileceğini de gösterirler. Öte yandan, bu programlar, içlerinde ilerlerken mevsimlerin değişeceği kadar da uzun değildir.

Birden Fazla Dosya İçeren Programları Niçin Kullanırız?

Birden fazla dosya içeren programları kullanmak için birkaç tane sebep mevcuttur. Sınıf kütüphanelerinin kullanımı, bir proje üzerinde çalışan programcıların organizasyonu ve bir programın kavramsal tasarımı bu sebepler arasında yer alır. Şimdi, bu konular üzerinde kısaca duralım.

Sınıf Kütüphaneleri

Geleneksel prosedürel dillerde, yazılım satıcılarının fonksiyon kütüphanelerini tedarik etmesi uzun zamandır alışkanlık haline gelmiştir. Diğer programcılar, son kullanıcı için uygulama geliştirmek için bu kütüphaneleri sonra kendilerine ait ısmarlama yazılmış rutinlerle birleştirirler.

Kütüphaneler, çok çeşitli alanlarda kullanıma hazır fonksiyonlar sunarlar. Örneğin, bir satıcı istatistik hesaplarını ele alan bir fonksiyon kütüphanesi tedarik edebilirken bir başkası ileri bellek yönetimi için bir kütüphane sağlayabilir.

C++, fonksiyonlardan ziyade sınıflar etrafında organize olduğu için, C++ program kütüphanelerinin de sınıflardan oluşması şaşırtıcı değildir. Şaşırtıcı olan, bir sınıf kütüphanesinin eski moda bir fonksiyon kütüphanesinden ne kadar daha iyi olduğudur. Sınıflar verileri ve fonksiyonları bir arada paketledikleri için ve sınıflar gerçek dünyadaki nesneleri çok daha yakından modelledikleri için bir sınıf kütüphanesi ile bu kütüphaneden yararlanacak olan uygulama arasındaki arayüz, bir fonksiyon kütüphanesinin sağlayacağından çok daha net olabilir.

Bu nedenlerden ötürü, sınıf kütüphanelerinin C++ programlama içinde üstlendikleri görev, fonksiyonların geleneksel programlamada üsteledikleri görevlerden çok daha önemlidir. Bir sınıf kütüphanesi programlama yükünün büyük bir bölümünü üstlenebilir. Bir uygulama programcısı, eğer doğru kütüphane sınıfı hazır olarak bulunuyorsa, son ürünü geliştirmek için sadece asgari düzeyde bir programlama gerektiğini fark edebilir. Ayrıca, giderek daha fazla sınıf kütüphanesi geliştirildikçe, belirli bir programlama probleminizi çözecek birini bulma şansınız da artmaya devam edecektir.

Bir sınıf kütüphanesinin önemli bir örneğini "Standart Şablon Kütüphanesi" adlı Bölüm 15'te göreceğiz.

Bir sınıf kütüphanesi genellikle iki bileşenden oluşur: *arayüz* ve *uygulama*. Şimdi her ikisi arasındaki farkı görelim.

Arayüz

Bir sınıf kütüphanesini yazan kişiye *sınıf geliştiricisi*; kütüphaneyi kullanan kişiye de *programcı* adını verelim.

Bir sınıf kütüphanesini kullanmak için, programcının sınıf deklarasyonları da dahil olmak üzere çeşitli deklarasyonlara erişmesi gerekir. Bu deklarasyonlar, kütüphanenin public parçası olarak düşünülebilir ve genellikle `.H` uzantılı bir başlık dosyası olarak kaynak kod şeklinde sağlanırlar. Bu dosya çoğunlukla müşterinin kaynak koduna `#include` ifadesi ile birleştirilir.

Bu tür bir başlık dosyasındaki deklarasyonlar birkaç sebepten ötürü public olmalıdır. Birincisi, müşteri açısından sınıfın esas tanımlarını görmek, tasvirlerini okumak zorunda olmaktan çok daha rahattır. Daha da önemlisi, programcının bu sınıflara dayanan nesneler tanımlaması ve bu nesnelerin üye fonksiyonlarını çağırması gerekecektir. Bu sadece, bu sınıfların deklarasyonlarını kaynak dosya içine yerleştirmekle mümkün olur.

Bu deklarasyonlara *arayüz* denir, çünkü sınıfı kullanan kişinin (programcı) gördüğü ve etkileşimde bulunduğu budur. Programcı kütüphanenin diğer parçasıyla *-uygulamayla* – ilgilenmek zorunda değildir.

Uygulama

Öte yandan, çeşitli sınıfların üye fonksiyonlarının nasıl çalıştıkları programcı tarafından bilinmek zorunda değildir. Sınıf geliştiricileri, tıpkı diğer yazılım geliştiriciler gibi, eğer becerebilirlerse kaynak kodlarını piyasaya sunmayı istemezler, çünkü kodlar gayri meşru olarak değiştirilebilir veya kodların korsan kullanımı/satışı yapılabilir. Üye fonksiyonlar – kısa yerel fonksiyonlar haricindekiler – bu nedenle, genellikle nesne formunda `.OBJ` dosyası olarak veya kütüphane dosyası (`.LIB`) olarak dağıtılırlar.

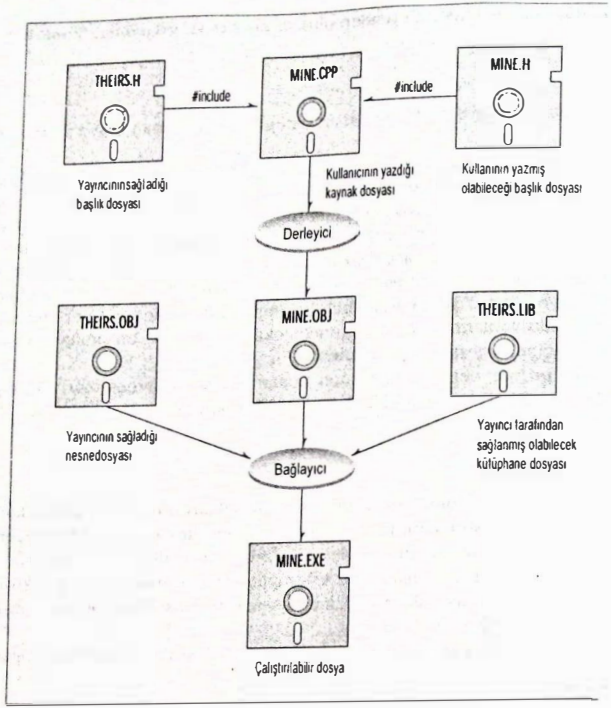
Şekil 13.1 birden fazla dosya içeren bir sistemde çeşitli dosyalar arasındaki bağlantıları gösteriyor.

Organizasyon ve Kavramsal Modelleme

Programlar, sınıf kütüphanelerini barındırmaktan başka nedenlerden ötürü de çeşitli dosyalara bölünebilir. Diğer programlama dillerinde olduğu gibi, birkaç programcı (veya programcılardan oluşan bir takım) içeren bir proje sıkça karşılaşılan bir durumdur. Her programcının sorumluluklarını ayrı bir dosya içinde sınırlandırmak projenin düzenlenmesine ve programın farklı parçaları arasında daha temiz bir arayüz tanımlanmasına yardımcı olur.

Bir programın ayrıca işlevselliğe göre ayrı dosyalara bölünmesi de sık sık söz konusu olur: Bir dosya, söz gelişi, grafik ekranda gösterilecek bir kodu idare ederken bir başka dosya matematiksel analizi ele alabilir ve üçüncü ise disk giriş/çıkışlarını düzenleyebilir. Büyük programlarda tek bir dosya, rahatlıkla idare etmek için fazla büyük olmuş olabilir.

Programı bölmek için sebep ne olursa olsun, birden fazla dosya içeren programlarla çalışırken kullanılan teknikler aynıdır.



ŞEKİL 13.1: Birden fazla dosya içeren bir uygulamada dosyalar.

Birden Fazla Dosya Kullanan Bir Program Oluşturmak

Diyelim ki, `THEIRS.OBJ` isimli önceden yazılmış bir sınıf satın aldınız. (`.LIB` uzantılı bir kütüphane dosyası da hemen hemen aynı şekilde ele alınır.) Bu sınıf muhtemelen bir başlık dosyası ile birlikte gelecektir; buna `THEIRS.H` diyelim. Kütüphanedeki sınıfları kullanmak için siz de kendi programınızı yazmıştınız; sizin kaynak dosyanız `MINE.CPP` olsun. Şimdi, bu bileşen dosyaları – `THEIRS.OBJ`, `THEIRS.H` ve `MINE.CPP` – tek bir çalıştırılabilir program içinde birleştirmek istiyorsunuz.

Başlık Dosyaları

`THEIRS.H` başlık dosyası bir `#include` ifadesi ile kolaylıkla kendi kaynak kodunuza (`MINE.CPP`) dahil edilebilir:

```
#include "theirs.h "
```


Dosya ismi etrafında açılı parantez yerine çift tırnak işareti kullanmak derleyiciye, dosyayı varsayılan içerik (include) dizininde değil de şu an içinde bulunulan dizinde aramasını bildirir.

Dizin

Tüm bileşen dosyalarının, `THEIRS.OBJ`, `THEIRS.H` ve `MINE.CPP`, aynı dizinde olduğundan emin olun. Aslında, karışıklığı önlemek için muhtemelen her proje için ayrı bir dizin oluşturmayı tercih edeceksiniz. (Bu tam manasıyla şart değildir, fakat en basit yönterndir.)

Her derleyici kendi kütüphane dosyalarını (mesela, `IOSTREAM` ve `CONIO.H`) genellikle `INCLUDE` adı verilen ve çoğunlukla derleyici dizin yapısı içinde birkaç kaç düzey aşağıya gömülen belirli bir dizin içinde saklar. Derleyici bu dizinin nerede olduğunu önceden bilir.

Derleyiciye ayrıca kendi tanımladığınız diğer içerik dizinlerini de bildirebilirsiniz. Başlık dosyalarınızın bazılarını, birkaç projede kullanılmak üzere hazır bulunmaları amacıyla bu tür bir dizin içinde saklamak isteyebilirsiniz. "Microsoft Visual C++" adlı Ek C'de ve "Borland C++ Builder" adlı Ek D'de bu tür bir dizinin nerede bulunduğunu derleyiciye nasıl bildireceğimizi açıklıyoruz.

Projeler

Bir çok derleyici, birden fazla dosyayı bir proje metaforu kullanarak idare eder. Bir proje bir uygulama için gerekli olan dosyaların tümünü içerir. Proje ayrıca bu dosyaları birleştirmek için gerekli komutları da, genellikle *proje dosyası* denilen özel bir dosya içinde içerir. Bu dosyanın uzantısı derleyici satıcısına bağlı olarak değişir. Borland için bu uzantı `.BPR` ve Microsoft için `.DSP`'dir. Modern derleyiciler bu dosyayı otomatik olarak kurup idare ederler. Bu nedenle, sizin dert etmenize gerek yoktur. Genel olarak, derleyiciye kullanmayı planladığınız kaynak dosyaların tümünü bildirmelisiniz ki, bu dosyalar projenize eklenebilsin. `.OBJ` ve `.LIB` dosyalarını aynı şekilde ekleyebilirsiniz. Ek C ve Ek D'de, belirli derleyiciler için birden fazla dosya içere programların oluşturulmasıyla ilgili ayrıntılar sunulmaktadır.

Kaynak (`.CPP` ve `.H`) dosyalarının tümünü derlemek ve elde edilen `.OBJ` dosyalarını (ve diğer `.OBJ` veya `.LIB` dosyalarını) en son bir `.EXE` dosyası olarak bağlamak için tek bir komut verilmesi yeterlidir. Buna *kurma* (*build*) işlemi denir. Çoğunlukla `.EXE` dosyası da ayrıca çalıştırılır. (Windows'ta ve diğer ileri programlama dillerinde başka tür dosyalar da mevcuttur.)

Bir projenin hoş tarafı, kaynak dosyalarını derlediğiniz sırada projenin tarihi takip etmesidir. Son kurma işleminden beri sadece üzerinde değişiklik yapılmış kaynak dosyaları yeniden derlenir. Bu özellikle büyük projelerde zamandan büyük ölçüde tasarruf sağlar. Bazı derleyiciler Make komutu ile Build komutunu birbirinden ayırt ederler. Make, en son kur işleminden beri sadece değiştirilen dosyaları derler; Build ise tarihe bakmaksızın tüm dosyaları derler.

Dosyalar Arası Haberleşme

Birden fazla dosya içeren programlarda, farklı dosyalardaki program öğeleri birbirleriyle haberleşmek zorundadır. Bu bölümde bunun nasıl mümkün olabileceğini göreceğiz. Öncelikle, ayrı ayrı derlenmiş ve bir araya getirilip bağlanmış kaynak (`.CPP`) dosyaları arasında haberleşmenin nasıl yürütüldüğünü ele alacağız. Sonra, kaynak dosyaları içine dahil edilen başlık dosyalarının duruma ne kadar uygun düştüğünü göreceğiz.

Kaynak Dosyaları Arasında Haberleşme

Bu bölümde, ayrı kaynak dosyalarının öğelerinin nasıl haberleştiğini araştıracağız. Üç tür programlama öğesini inceleyeceğiz Değişkenler, fonksiyonlar ve sınıflar. Dosyalar arası kullanımla ilgili her birinin kendisine ait kuralları var.

Kapsam fikri burada önemli olacak; bu nedenle, Bölüm 5'teki kapsam ve depolama sınıfı ile ilgili anlatımımıza geri dönüp göz atmak isteyebilirsiniz. Kapsam, bir program içinde bir değişkenin veya bir başka program öğesinin erişilebildiği alandır. Bir fonksiyon içinde bildirilen öğeler *yerel* kapsama sahiptir; yani, sadece fonksiyonun gövdesi içinde erişilebilirler. Benzer şekilde, sınıf üyeleri de yalnızca sınıf içinde erişilebilirler (kapsam çözünürlük operatörü kullanılmadığı takdirde).

Herhangi bir fonksiyonun veya sınıfın dışında bildirilmiş program öğeleri *global* kapsama sahiptir: Tanımlandıkları noktadan başlayarak tüm program boyunca kullanılabilirler. Az sonra göreceğimiz gibi bunlara, diğer dosyalardan da erişilebilir.

Dosyalar Arası Değişkenler

Basit değişkenlerle başlayacağız. Deklarasyon ve tanım arasındaki farkı hatırlayın. Basit bir değişkeni, ismini ve tipini vererek *deklare ederiz*. Bunun, değişken için bellekte fiziksel bir yer sağlaması şart değildir. Bu sadece derleyiciye, bu isimde ve tipte bir değişkeninin bir yerlerde mevcut olabileceğini bildirir. Bir değişkene bellekte değişkenin değerini tutabilecek bir yer verilince değişken *tanımlanmış* olur. Tanım, "gerçek" değişkeni oluşturur.

Deklarasyonların birçoğu ayrıca birer tanımdır. Aslında, basit bir değişkenin tanım olmayan tek deklarasyonu, **extern** anahtar kelimesi kullanıldığında ortaya çıkar (ilk kullanıma hazırlanmadan).

```
int someVar;            //deklarasyon ve ayni zamanda tanim
extern int someVar;     //sadece deklarasyon
```

Tahmin edeceğiniz gibi, bir global değişken bir program içinde sadece tek bir yerde tanımlanabilir.

```
//A dosyasi
int globalVar;       //A dosyasi icindeki tanim

//B dosyasi
int globalVar;       //kural disi: B dosyasi icindeki de ayni tanim
```

Elbette bu görüş sadece global değişkenler için geçerlidir. Değişkenlerin farklı fonksiyonlara veya sınıflara yerel olmaları şartıyla, aynı isimde ve tipte istediğiniz kadar çok sayıda değişken tanımlayabilirsiniz.

Bir dosya içindeki bir global değişkene farklı bir dosyadan nasıl erişirsiniz? Bağlayıcının (linker) aynı global değişkeni birden fazla dosya içinde tanımlanmasına itiraz etmesi, bir dosya içindeki bir değişkenin otomatik olarak diğer dosyalardaki kodların tümü tarafından erişilebilir olması anlamına gelmez. Değişkeni kullanan dosyaların her birinin içinde değişkeni tanımlamalısınız. Aşağıdaki gibi yazarsanız derleyici, **globalVar**'ın tanımlanmamış bir tanımlayıcı olduğunu bildirecektir.

```
//A dosyasi
int globalVar;    //tanimlandi
```



```
//B dosyasi
globalVar=3;      //kural disi, globalVar burada bilinmiyor
```

Bir değişkenin tanımlandığı dosyanın haricindeki dosyalardan da erişilebilmesi için bu değişkeni diğer dosyalarda **extern** anahtar kelimesini kullanarak bildirmeniz gerekir.

```
//A dosyasi
int globalVar;          //tanim

//B dosyasi
extern int globalVar;   //deklarasyon
globalVar = 3;          //bu simdi tamam
```

Deklarasyon, A dosyasındaki **globalVar** değişkeninin B dosyasından da erişilebilmesini sağlar. **extern** anahtar kelimesi, deklarasyonun *yalnızca* deklarasyon olduğuna, tanım olmadığına işaret eder. **extern** anahtar kelimesi derleyiciye (derleyici bir kerede sadece bir dosya görebilir) B dosyasındaki **globalVar** değişkeninin orada tanımsız olmasından dolayı kaygılanmamasını bildirir. Bağlayıcı (tüm dosyaları görür), bir dosya içindeki bir değişkene yapılan bir referansı, değişkenin bir başka dosya içinde bulunan tanımı ile birleştirme görevini yerine getirecektir.

Muhtemelen şaşırtıcı gelebilecek bir kısıtlamaya dikkat etmelisiniz: Bir değişkene **extern** deklarasyon içinde ilk değer ataması yapamazsınız. Aşağıdaki ifade, derleyicinin, **globalVar** değişkenini sadece deklare etmek değil aynı zamanda tanımlamak niyetinde olduğunuzu zannetmesine neden olacaktır:

```
extern int globalVar = 27;    //dusundugunuz gibi degil
```

Derleyici **extern** anahtar kelimesini dikkate almayacak ve değişkenin tanımını oluşturacaktır. Eğer değişken bir başka dosya içinde tanımlanmışsa, bağlayıcıdan "önceden tanımlı" hata mesajını alacaksınız demektir.

Peki, global değişkenleri aynı isimle farklı dosyalar içinde kullanmayı gerçekten istiyorsanız ne olacak? Bu durumda, bu değişkenleri **static** anahtar kelimesini kullanarak tanımlayabilirsiniz. Bu anahtar kelime, bir değişkenin erişilebilirliğini tanımlandığı dosya ile sınırlandırır. Diğer değişkenler aynı isim ile diğer dosyalarda kullanılabilir.

```
//A dosyasi
static int globalVar;   //tanim; sadece A dosyasi icinden erisilebilir

//B dosyasi
static int globalVar;   //tanim; sadece B dosyasi icinden erisilebilir
```

Burada aynı isimli iki değişken tanımlanmış olmasına rağmen, bir karışıklık doğmaz. **globalVar** değişkenini kullanmak isteyen A dosyasındaki kod, kendi dosyası içindeki değişkene erişir, B dosyasındaki kod da aynı şekilde davranıyor.

Statik değişkenlerin *dahili bağlandığı* (*internal linkage*), statik olmayan global değişkenlerin *harici bağlandığı* (*external linkage*) söylenir. (Bu bölümde az sonra göreceğimiz gibi, bir değişkenin kapsamını tek bir dosya ile sınırlandırmak için isim uzaylarını da kullanabilirsiniz.)

Birden fazla dosya içeren programlarda, global değişkenlerin diğer dosyalar tarafından erişilmediği her duruma global değişkenleri statik yapmak iyi bir fikirdir. Bu sayede, aynı isim

kazara bir başka dosya içinde kullanıldığında ortaya çıkacak problemler önlenmiş olur. Ayrıca listeyi inceleyen birine de, herhangi bir yerden erişilmekte olan bir değişken için kaygılanmamak gerektiğini açıkça bildirir.

Dikkat ederseniz, **static** anahtar kelimesinin yerel ya da global değişken ile kullanılıp kullanılmadığına bağlı olarak farklı anlamları vardır. "Fonksiyonlar" adlı Bölüm 5'te, **static** anahtar kelimesinin bir yerel değişkeni (bir fonksiyon içinde tanımlanmış olan) nitelediğinde, değişkenin ömrünü fonksiyonunkinden çıkarıp programının ömrüne eşit olacak şekilde değiştirdiğini, fakat erişilebilirliğinin fonksiyon ile sınırlı kaldığını görmüştük. "Nenseler ve Sınıflar" adlı Bölüm 6'da ele aldığımız gibi, **static** bir sınıfa ait bir veri üyesi her nesne için ayrı bir değer içermektense tüm nesneler için aynı değere sahiptir. Bununla birlikte, bir global değişken için **static** anahtar kelimesini kullanmak sadece, değişkenin erişilebilirliğini kendi dosyasına sınırlar.

Bir dosya içinde tanımlı bir **const** değişken normal olarak bir başka dosya içinden erişilemez. Bu bağlamda, sanki bir **static** değişken gibidir. Yine de, hem tanımda hem deklarasyonda **extern** anahtar kelimesini kullanarak bir **const** değişkeninin diğer dosyalardan da erişilmesini sağlayabilirsiniz.

```
//A dosyasi
extern const int conVar2 = 99;      //tanim

//B dosyasi
extern const int conVar2;           //deklarasyon
```

Burada, **B** dosyası, **A** dosyası içindeki **const** değişkene erişim hakkına sahiptir. Derleyici bir **const** tanım ile deklarasyon arasındaki farkı, değişkene değer atanan yere bakarak anlayabilir.

Dosyalar Arası Fonksiyonlar

Hatırlarsanız, bir fonksiyon deklarasyonu fonksiyonun ismini, döndürdüğü tipi ve argümanlarının tiplerini belirtir. Bir fonksiyon tanımı ise fonksiyonun gövdesini içeren bir deklarasyondur. (Gövde, küme parantezleri içine yazılan koddur.)

Derleyici bir fonksiyona çağrıda bulunduğu zaman, fonksiyonun nasıl çalıştığını bilmesi gerekmez. Bilmesi gerekenler sadece fonksiyonun ismi, fonksiyonun döndürdüğü tip ve argümanlarının tipleridir. Deklarasyonun açıkça belirttiği bunlardan ibarettir. Ek olarak başka hiçbir anahtar kelimeye (**extern** gibi) gerek yoktur. Gerekli olan tek şey, fonksiyonu çağırmadan önce ikinci dosya içinde fonksiyonun deklarasyonunun yapılmış olmasıdır.

```
//A dosyasi
int add(int a, int b)       //fonksiyon tanimi
   { return a+b; }          //(fonksiyon govdesini icerir)

//B dosyasi
int add(int, int);          //fonksiyon deklarasyonu (govdesiz)
. . .
int answer = add(3, 2);     //fonksiyon cagrisi
```

Fonksiyonlarda **extern** anahtar kelimesini kullanmanıza gerek yoktur, çünkü derleyici fonksiyonun deklarasyonu ile tanımı arasındaki farkı anlayabilir: Deklarasyonun gövdesi yoktur.

Bu arada, bir fonksiyonun veya başka bir program öğesinin istediğiniz kadar çok sayıda deklarasyonunu (tanımını değil) yapabilirsiniz. Deklarasyonların uyuşması şartıyla derleyici buna itiraz etmeyecektir.

```
//A dosyasi
int add(int, int);      //deklarasyon
int add(int, int);      //bir deklarasyon daha yapilabilir
```

Değişkenler gibi fonksiyonlar da **static** olarak bildirilerek diğer dosyalar tarafından erişilebilir hale getirilebilir.

```
//A dosyasi
static int add(int a, int b)     //fonksiyon tanimi
   { return a+b; }

//B dosyasi
static int add(int a, int b)  //farkli bir fonksiyon
   { return a+b; }
```

Bu kod iki farklı fonksiyon tanımlar. Bu fonksiyonların hiçbiri diğer dosya içinden erişilemez.

Dosyalar Arası Sınıflar

Sınıflar, basit değişkenlere benzemez. Şöyle ki, bir sınıf tanımı bellekten hiç yer ayırmaz. Bir sınıf tanımı sadece derleyiciye bu sınıfı oluşturan üyelerin neler olduğunu bildirir. Bu, **int** tipi için kaç byte kullanılacağını belirtmek gibi bir şeydir. Ancak, derleyici **int** tipinin ayrıntıları hakkında zaten bilgi sahibidir, fakat **someClass** tipi hakkında siz tanımlayana kadar bir şey bilmez.

Bir sınıf tanımı, üyelerinin tümünün deklarasyonlarını veya tanımlarını içerir:

```
class someClass                 //sinif tanimi
   {
   private:
      int memVar;               //uye veri tanimi
   public:
      int memFunc(int, int);  //uye fonksiyon tanimi
   };
```

Üyeler, sınıf tanımı içinde bildirilmelidir, fakat tanımlanmak zorunda değildir. Önceden gördüğümüz gibi, üye fonksiyon tanımları rutin olarak sınıfın dışına yerleştirilir ve kapsam çözünürlük operatörü kullanılarak teşhis edilir.

Bir sınıf deklarasyonu, sınıf için kullanılacak belirli bir isim içeren bir ifadeden ibarettir. Sınıf deklarasyonu, sınıfın üyeleri hakkında derleyiciye hiçbir bilgi taşımaz.

```
class someClass;      //sinif deklarasyonu
```

Bir sınıf tanımını, bu sınıfa ait bir nesnenin tanımı (oluşturulması) ile karıştırmayın:

```
someClass anObj;
```

Sınıf tanımından farklı olarak bir nesnenin tanımı, nesne için bellekte yer ayırır. Dosyalar arası haberleşmede sınıflar değişkenlerden ve fonksiyonlardan farklı davranırlar. Birden fazla kaynak dosyasının içinden bir sınıfa erişmek için sınıfı, sınıfın nesnesinin kullanılacağı dosyaların tümünde tanımlamak (sadece bildirmek yetmez) gereklidir. Bir sınıfın `A` dosyası içinde tanımlanmış olması ve `B` dosyası içinde bildirilmesi, derleyicinin `B` dosyası içinde bu sınıfın nesnelerini oluşturabileceği anlamına gelmez.

Bir sınıf neden kullanıldığı her dosyada tanımlanmak zorundadır? Derleyicinin, derlediği her şeyin veri tipini bilmesi gerekir. Basit değişkenler için bir deklarasyon, derleyici için yeterli olur; çünkü deklarasyon zaten derleyicinin önceden bildiği bir tipi belirtir.

```
//deklarasyon
extern int someVar;      //derleyici bunu gorurse,
someVar = 3;             //bunu uretebilir
```

Aynı şekilde, bir fonksiyon deklarasyonu da bir fonksiyon çağrısı için gerekli olabilecek her türlü veri tipini açığa çıkarır.

```
//deklarasyon
int someFunc(int, int);       //derleyici bunu gorurse,
var1 = someFunc(var2, var3); //bunu uretebilir
```

Bununla birlikte, bir sınıf için üye verilerini ve fonksiyonlarını açıkça belirtmek amacıyla bütün bir sınıf tanımı gereklidir.

```
//tanim
class someClass                  //derleyici bunu gorurse,
   {
   private:
      int memVar;
   public:
      int memFunc(int,int);
   };
someClass someObj;               //bunu ve
v1 = someObj.memFunc(v2, v3);    //bunu uretebilir
```

Sınıf nesneleri ile ilgilenmek (nesnelere işaret eden işaretçiler ve referanslar hariç) amacıyla kod üretmek için sadece bir deklarasyon, derleyici açısından yeterli değildir.

Bir sınıfı bir kaynak dosya (`.CPP`) içinde birden fazla kez tanımlayamazsınız, ancak programdaki kaynak dosyaların her biri, aynı sınıfın kendilerine ait tanımlarına sahip olabilirler. Gerçekten de, eğer dosya böyle bir sınıfın nesneleri ile çalışmak durumda ise bu tür bir tanıma sahip olmalıdır. Aşağıda, bir sınıf tanımını birden fazla dosyaya sunmak için bir başlık dosyasının nasıl kullanıldığını göstereceğiz.

Başlık Dosyaları

Bölüm 2'de belirttiğimiz gibi, `#include` önişlemci direktifi bir dosyadaki metnin bir başka dosyaya eklenmesini sağlayarak tıpkı bir kelime işlemcideki "yapıştır" fonksiyonu gibi davranır. Kaynak dosyalarımıza dahil edilmiş bir çok kütüphane dosyası örneği gördük, mesela `IOSTREAM` bunlardan biridir.

Ayrıca, kendi başlık dosyalarımızı (genellikle `.H`) yazabilir ve kaynak dosyalarımıza bu başlık dosyalarını dahil edebiliriz.

Yaygın Bir Bilgi

Bir başlık dosyasını kullanmanın bir sebebi, iki veya daha fazla kaynak dosyasına aynı bilgiyi sağlamaktır. Başlık dosyası, değişken ve fonksiyon deklarasyonlarını tutar ve kaynak dosyalarına dahil edilir. Değişkenlere veya fonksiyonlara bu şekilde birçok dosyadan erişilebilir.

Elbette, programın öğelerinin her biri ayrıca bir yerlerde tanımlanmalıdır. Bu örnekte, bir değişken ile bir fonksiyon `FILE.H` içinde bildirilir, `FILEA.CPP` içinde de tanımlanırlar. `FILEB.CPP` içindeki kod daha sonra, kendisinin ilave deklarasyonlar yapmasına gerek kalmadan bu öğeleri kullanabilir.

```
// fileH.h
extern int gloVar;                  //degisken deklarasyonu
int gloFunc(int);                   //fonksiyon deklarasyonu

//fileA.cpp
int gloVar;                         //degisken tanimi
int GloFunc(int n)                  //fonksiyon tanimi
   { return n; }

//fileB.cpp
#include "fileH.h"
. . .
gloVar = 5;                         //degisken ile calisir
int gloVarB = gloFunc(gloVar);      //fonksiyon ile calisir
```

Dikkat: Bir başlık dosyası içine deklarasyonları yerleştirebilirsiniz, fakat değişken ve fonksiyon tanımlarını (`static` veya `const` değillerse), birkaç kaynak dosyası tarafından paylaşılacak olan bir başlık dosyası içine yerleştiremezsiniz. Eğer yerleştirirseniz, aynı tanımlar iki farklı kaynak dosya içinde mevcut olmuş olur ve bağlayıcı "birden fazla tanımlama" hatası verir.

Çok yaygın ve neredeyse bir o kadar da temel bir teknik, bir sınıf tanımını, ihtiyaç duyulan tüm kaynak dosyalarına dahil edilmiş bir başlık dosyasının içine yerleştirmektir. Bu, birden fazla tanımlama problemine yol açmaz, çünkü bir sınıf tanımı bellekten yer ayırmaz, bu sadece bir belirtimden ibarettir.

```
// fileH.h
class someClass                     //sinif tanimi
   {
   private:
      int memVar;
   public:
      int memFunc(int, int);
   };

// fileA.cpp
#include "fileH.h"
int main()
   {
   someClass obj1;                  //bir nesne olustur
   int var1 = obj1.memFunc(2, 3);   //nesne ile calis
   }

// fileB.cpp
#include "fileH.h"
int func()
   {
```

```
    someClass obj2;                    //bir nesne olustur
    int var2 = obj2.memFunc(4, 5);     //nesne ile calis
    }
```

Bir başlık dosyası kullanmak yerine aslında sınıf tanımının metnini kopyalayıp kaynak dosyalarının her birine elle yapıştırırsanız ne olur? Bu durumda, sınıf üzerinde yapılan her değişiklik dosyaların her birinin ayrı ayrı düzeltilmesini gerektirecektir. Bu zaman alıcı ve hatalara açık bir yöntem olacaktır.

Şimdiye kadar, içinde harici (**extern**) üye fonksiyon tanımı içermeyen sınıf tanımlarını gösterdik. Harici üye fonksiyonlar nerede tanımlanabilir? Sıradan fonksiyonlar gibi harici üye fonksiyonlar da herhangi bir kaynak dosya içine yerleşebilir. Bağlayıcı, ihtiyaç oldukça bunları bağlayacaktır. Sınıf tanımı, dosyaların her biri içindeki üye fonksiyonları bildirme işlevini görür. Tek bir dosyada geçerli olduğu gibi, üye fonksiyon tanımı sınıf ismini ve kapsam çözünürlük operatörünü içermek zorundadır.

```
// fileH.h
class someClass                        //sinif tanimi
   {
   private:
      int memVar;
   public:
      int memFunc(int, int);           //uye fonksiyon deklarasyonu
   };
;

// fileA.cpp
#include "fileH.h"
int someClass::memFunc(int n1,int n2)  //uye fonksiyon tanimi
   { return n1 + n2; }

// fileB.cpp
#include "fileH.h"
someClass anObj;                       //bir nesne olustur
int answer = anObj.memFunc(6, 7);      //uye fonksiyonu kullan
```

Birden Fazla "Include" Tehlikesi

Birden fazla kaynak dosyası tarafından paylaşılacak olan bir başlık dosyasının içinde bir fonksiyon veya değişken tanımlayamayacağınızı belirtmiştik. Böyle yapmanız "birden fazla tanım" hatalarına neden olabilir. Aynı başlık dosyasını bir kaynak dosyası içine iki kez dahil ettiğinizde de aynı problem ortaya çıkar. Böyle bir şey nasıl meydana gelir? Muhtemelen bu kadar bariz bir hata yapmazsınız:

```
//app.cpp dosyasi
#include "headone.h"
#include "headone.h"
```

Fakat, diyelim ki, **APP.CPP** adında bir kaynak dosyanız ve **HEADONE.H** ve **HEADTWO.H** adında iki başlık dosyanız var. **HEADONE.H**'nin de **HEADTWO.H**'yi içerdiğini farz edelim. Ne yazık ki, bunu unutuyorsunuz ve her ikisini de **APP.CPP** içine dahil ediyorsunuz:

```
// headtwo.h dosyasi
int globalVar;
```



```
//headone.h dosyasi
#include "headtwo.h"

//app.cpp dosyasi
#include "headone.h"
#include "headtwo.h"
```

Şimdi **APP.CPP**'yi derlerseniz ne olur? **#include** direktifleri başlık dosyalarının metinlerini **APP.CPP** içine yapıştırdıkları andan itibaren şöyle bir durumla karşılaşırsınız:

```
//app.cpp dosyasi
. . .
int globalVar;          //headone.h uzerinden head2.h dosyasindan
. . .
int globalVar;          //dogrudan head2.h dosyasindan
```

Bu durum, **globalVar** iki kez tanımlandığından dolayı derleyicinin itiraz etmesine yol açacaktır.

Birden Fazla "Include"u Önlemek

Şimdi, "birden fazla tanım" hatalarını, bir başlık dosyası kaynak dosyanız içinde birden fazla kere dahil edilse bile nasıl önleyeceğimizi görelim. Başlık dosyası içindeki tanımların önüne aşağıdaki önişlemci direktifi yerleştirilir:

```
#if !defined( HEADCOM )
```

(Sadece **HEADCOM**'u kullanmanız şart değil, herhangi bir tanımlayıcı kullanabilirsiniz.) Bu ifade, eğer **HEADCOM** *tanımlanmamışsa* (ünlem işareti mantıksal **NOT** (DEĞİL) operatörüdür), bu direktifi takip eden metnin tümünün ifadeyi kapatan **#endif** direktifine kadar normal biçimde kaynak dosyaya yapıştırılacağını bildirir. Fakat eğer **HEADCOM** *tanımlanmışsa*, tanımlama şu direktifle gerçekleştirilebilir:

```
#define HEADCOM
```

Bu durumda, takip eden metin kaynak dosya içine dahil edilmez. Bu metinle ilk kez karşılaşıldığında **HEADCOM** tanımlanmamış olduğu için ve **#if !defined()** direktifinden hemen sonra tanımlandığı için bu direktifle ifadeyi kapatan **#endif** direktifi arasındaki metin, ilk kez karşılaşıldığında dosyaya dahil edilecek ama bir daha asla dahil edilmeyecektir. Düzenleme şu şekilde yapılır:

```
#if !defined( HEADCOM )    //eger HEADCOM tanimlanmamissa,
#define HEADCOM            //tanimla

int globalVar;             //bu degiskeni tanimla
int func(int a, int b)     //bu fonksiyonu tanimla
   { return a+b; }
#endif                     //kosulun sonu
```

Bu yaklaşımı, bir başlık dosyasının bir kaynak dosyası içine birden fazla kez dahil edilmesi ihtimali olduğunda kullanmalısınız.

Eski bir direktif olan `#ifndef` de `#if !defined()` ile aynı şekilde kullanılırdı; derleyicinizle birlikte gelen başlık dosyalarının birçoğunda bu direktifi göreceksiniz. Buna rağmen, bu direktifin kullanımı artık desteklenmez.

Dikkat ederseniz, `#if !defined()` yaklaşımı, `globalVar`'ın (veya bir başka değişkenin ya da fonksiyonun) tanımının, aynı kaynak dosya içinden birden fazla kez dahil edilmesiyle sonuçlanan durumlarda işe yarar. `globalVar`, `H` dosyasının içinde tanımlıysa ve `H` dosyası `A` ve `B` gibi farklı kaynak dosyalarının içine dahil edilmişse bu yaklaşım işe yaramaz. Ön işlemci, ayrı dosyalar içindeki birden fazla ifadeyi tespit etmekte yetersizdir; bu nedenle bağlayıcı, `globalVar`'ın birden fazla tanımlandığını belirtip, itiraz edecektir.

İsim Uzayları

Program öğelerinin erişilebilirliklerini, bu öğeleri bir dosya veya sınıf içinde tanımlayarak veya global öğeleri `static` veya `const` yaparak sınırlandırabileceğimizi gördük. Yine de, bazen daha çok yönlü bir yaklaşım gerekecektir.

Söz gelişi, bir sınıf kütüphanesi yazarken programcılar, üye olmayan fonksiyonlar ve sınıflar için kısa ve sık kullanılan isimleri, mesela `add()` ve `book`, tercih edeceklerdir. Bununla birlikte, kısa ve sık kullanılan isimler, başka bir kütüphanenin yazarları tarafından veya bu kütüphaneyi kullanan bir uygulama tarafında tercih edilen isimlerle aynı olabilir. Bu durum "isimlerin çarpışmasına" yol açabilir ve derleyicinizin "birden fazla tanım" hatası vermesine neden olabilir. İsim uzayları ortaya çıkana kadar programcılar bu problemi çözmek için uzun isimler kullanmaya mecbur kalırlardı:

```
Henry's_Simplified_Statistics_Library_add();
```

Ancak, uzun isimlerin okunması ve yazması zordur; ayrıca, program listesinde haddinden fazla yer işgal ederler. İsim uzayları bu problemi çözebilir. (Dikkat ederseniz, üye fonksiyonlar isimlerin çarpışmasına neden olmaz, çünkü üye fonksiyonların geçerli oldukları alan, zaten sınıf ile sınırlıdır.)

Bir İsim Uzayını Tanımlamak

İsim uzayı, bir dosyanın isim verilen bir bölümüdür. Aşağıdaki kod, bazı deklarasyonlar içeren `geo` adında bir isim uzayı tanımlar:

```
namespace geo
   {
   const double PI = 3.14159;
   double circumf(double radius)
      { return 2 * PI * radius; }
   }     //namespace geo'nun sonu
```

Küme parantezleri isim uzayını sınırlandırır. Küme parantezleri içinde bildirilen değişkenler ve diğer program öğeleri isim uzayı *üyeleri* olarak adlandırılıyor. Dikkat ederseniz, kapanış küme parantezinin ardından sınıflarda olduğu gibi noktalı virgül kullanılmaz.

İsim Uzayı Üyelerine Erişmek

İsim uzayının dışında kalan kod, isim uzayının içindeki öğelere erişemez; en azından normal bir yoldan erişemez. İsim uzayı bu öğeleri erişilmez kılar:


```
namespace geo
   {
   const double PI = 3.14159;
   double circumf(double radius)
      { return 2 * PI * radius; }
   }//namespace geo'nun sonu

double c = circumf(10);    //burada calismayacaktir
```

Öğelerin isim uzayının dışından da erişilebilir olmasını sağlamak için bu öğeleri kullanırken isim uzayının adını da kullanmanız gerekir. Bunu iki şekilde yapabilirsiniz. Birincisi, her öğenin isminin önünde isim uzayının adını ve kapsam çözünürlük operatörünü kullanabilirsiniz:

```
double c = geo::circumf(10);  //tamam
```

Ya da `using` direktifini kullanabilirsiniz:

```
using namespace geo;
double c = circumf(10);       //tamam
```

`using` direktifi her zaman olduğu gibi, o noktadan itibaren isim uzayının erişilebilir olmasını sağlar. Bununla birlikte, `using` direktifinin etkili olduğu alanı belirli bir blokla (mesela bir fonksiyonla) sınırlandırabilirsiniz:

```
void seriousCalcs()
   {
   using namespace geo;
   //burada baska kodlar var
   double c = circumf(r);     //tamam
   }

double c = circumf(r);        //tamam degil
```

Burada, isim uzayının üyeleri sadece fonksiyonun gövdesi içinden erişilebilirler.

Başlık Dosyaları İçindeki İsim Uzayları

İsim uzayları en yaygın olarak kütüphane sınıfları veya fonksiyonları içeren başlık dosyalarında kullanılır. Bu tür kütüphanelerin her biri kendi isim uzayına sahip olabilir. Şimdilik `std` isim uzayı ile tanışıyorsunuz. `std` isim uzayının üyeleri Standart C++ Kütüphanesini oluşturur.

Birden Fazla İsim Uzayı Tanımı

Aynı isim uzayı tanımının birden fazla örneği olabilir:

```
namespace geo
   {
   const double PI = 3.14159;
   }     //namespace geo'nun sonu

//(burada baska bir kod olabilir)

namespace geo
   {
```

```
    double circumf(double radius)
       { return 2 * PI * radius;}
    }     //namespace geo'nun sonu
```

Bu yeni bir tanım gibi görünebilir, ama aslında sadece aynı tanımın devamıdır. Bu, bir isim uzayının farklı başlık dosyalarında kullanılmasına imkan verir. Daha sonra bu başlık dosyalarını tümü bir kaynak dosya içine dahil edilebilir.

Standart C++ Kütüphanesinde, düzinelerce başlık dosyası **std** isim uzayını kullanır.

```
//fileA.h
namespace alpha
   {
   void funcA();
   }
//fileB.h
namespace alpha
   {
   void funcB();
   }

fileMain.cpp
#include "fileA.h "
#include "fileB.h "
using namespace alpha;
funcA();
funcB();
```

Sanki bir isim uzayının içindeymiş gibi davranan deklarasyonları, bir isim uzayının dışına yerleştirebilirsiniz. Bunun için tüm ihtiyacınız olan kapsam çözünürlük operatörü ile isimuzayının adıdır:

```
namespace beta
   {
   int uno;
   }

int beta::dos;
```

Burada, hem **uno** hem de **dos**, **beta** isim uzayının içinde bildirilirler.

İsimsiz İsim Uzayları

İsimsiz bir isim uzayı tanımlayabilirsiniz. Böyle yaptığınız taktirde isim uzayına, tanımlı olduğu dosya boyunca erişilebilir, fakat diğer dosyalardan erişilemez. Derleyici, isimsiz isim uzayına dosyaya özgü dahili bir isim verir. İsimsiz isim uzayında bildirilen öğelere, dosyanın herhangi bir yerinden erişilebilir. Aşağıdaki program listesinde **funcA()** ve **funcB()** fonksiyonları **gloVar** değişkenine kendi dosyalarında erişebilirler.

```
//fileA.cpp
namespace
   {                          //fileA.cpp dosyasina ait olan isimsiz isim uzayi
   int gloVar = 111;
   }
funcA()
   { cout << gloVar; }     //ekranda 111 gorunur
//fileB.cpp
namespace                  //fileB.cpp dosyasina ait olan isimsiz isim uzayi
   {
   int gloVar = 222;
   }
funcB()
   { cout << gloVar; }     //ekranda 222 gorunur
```


Bu örnekte, her iki dosya da **gloVar** adında bir değişken içerir, fakat bir karışıklık doğmaz, çünkü değişkenlerin her biri kendi dosyalarına ait isimsiz isim uzaylarında bildirilir ve geri kalan yerlerde erişilemezler.

Bu yöntem, global değişkenlerin geçerlilik alanlarını kendi dosyalarıyla sınırlandırmak için **static** anahtar kelimesinin kullanımına bir alternatif teşkil eder. Aslında, isim uzayı yöntemi artık, öğeleri **static** yapma yönteminden daha çok tercih edilir.

typedef ile Tiplere Yeni İsim Vermek

Belirli durumlarda **typedef** anahtar kelimesini kullanışlı bulabilirsiniz. Üstelik, başkalarının program listelerinde bu kelime ile kesinlikle karşılaşacaksınız. **typedef**, bir veri tipi için yeni bir isim tanımlamanıza imkan verir. Örneğin, aşağıdaki ifade **unlong** kelimesini **unsigned long** ile eşanlamlı kılar.

```
typedef unsigned long unlong;
```

Artık, yeni kelimeyi kullanarak değişkenlerinizi tanımlayabilirsiniz:

```
unlong var1,var2;
```

Bu yöntem belki yerden çok az kazandırabilir ama listenizin daha okunaklı olmasını sağlayabilir. Daha da iyisi, yeni tip isimleri uydurabilirsiniz. Öyle ki, bu tip isimleri bu tipte tanımlanmış herhangi bir değişkenin amacını açıkça ortaya koyar.

```
typedef int FLAG;          //isaret degerlerini tutmak icin kullanilan
                           //tamsayi degiskeni
typedef int KILOGRAMS;     //kilogram degerlerini tutmak icin kullanilan
                           //tamsayi degiskeni
```

C++'ta işaretçilerin belirtiliş biçimini beğenmiyorsanız, değiştirebilirsiniz:

```
int *p1,*p2,*p3;        //normal deklarasyon
typedef int* ptrInt;    //int'e isaret eden isaretci icin yeni bir isim
ptrInt p1,p2,p3;        //sadelestirilmis deklarasyon
```

Bu sayede bütün o sinir bozucu asterikslerden kurtulmuş olursunuz.

C++'ta sınıflar da bir tip olduğu için **typedef** kullanarak sınıflar için de alternatif isimler tanımlayabilirsiniz. Daha önce, geliştiricilerin kimi zaman haddinden fazla uzun isimler tanımladıklarından bahsetmiştik. Bu tür isimleri kullanmanız gerekiyorsa, bu isimleri yazmak uygun olmayabilir, üstelik program listenizin okunmasını zorlaştırabilir. Bu problemi, en azından sınıf isimleri açısından, **typedef** ile çözebilirsiniz:

```
class GeorgeSmith_Display_Utility          //Sinif deklarasyonu
   {
   //uyeler
   };

typedef GeorgeSmith_Display_Utility GSdu;  //sinifa yeni isim ver

GSdu anObj;       //yeni ismi kullanarak yeni nesne tanimla
```

typedef ile tipleri yeniden isimlendirmek özellikle başlık dosyalarında ele alınır; böylece, birden fazla kaynak dosyası aynı isimleri kullanabilir. Birçok yazılım geliştirme kuruluşu **typedef**'i haddinden fazla kullanır. Öyle ki, sonuçta ortaya çıkan program neredeyse farklı bir dilde yazılmış gibi görünür.

Birden fazla dosya içeren programlarla ilgili genel kavramların bazılarını incelediğimize göre artık birkaç örneğe göz atabiliriz. Bu programlar önceki bölümlerde ele aldığımız konuların tümümü kapsamaz, ama bu programlar, bir kütüphane yazarı tarafından sunulan kodun, bir uygulama programcısı tarafından kullanıldığı bazı tipik durumları sizlere gösterecektir.

Çok Uzun Bir Sayı Sınıfı

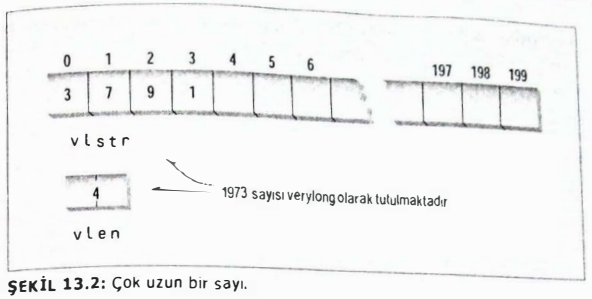
Kimi zaman **unsigned long** veri tipi bile belirli aritmetik işlemleri için yeterli basamak uzunluğunu sağlamaz. **unsigned long**, 4,294,967,295 sayısına kadar tüm tamsayıları tutan ya da yaklaşık 10 basamak uzunluğundaki tamsayıları tutan Standart C++'ın en büyük tamsayı tipidir. Bu, bir cep hesap makinesinin ele alabileceği basamak sayısı ile aşağı yukarı aynıdır. Fakat, bundan daha fazla basamak içeren tamsayılarla çalışmanız gerekiyorsa bir problemle karşı karşıyasınız demektir.

Sıradaki örneğimiz bir çözüm önerir. Bu örnek, uzunluğu 1,000 basamağa kadar varan tamsayıları tutan bir sınıf içerir. Eğer daha da uzun (ya da daha kısa) sayılar elde etmek istiyorsanız, programdaki tek bir sabiti değiştirebilirsiniz.

Karakter Katarı Olarak Sayılar

verylong sınıfı, sayıları rakam karakter katarı olarak saklar. Bunlar eski moda **char*** C karakter katarlarıdır. Bu bağlamda, **string** sınıfı ile çalışmaktansa **char*** C karakter katarlarıyla çalışmak daha kolaydır. C karakter katarlarının kullanımı geniş rakam kapasitesini açıklar: C karakter katarları birer diziden ibaret olduğu için C++ uzun C- karakter katarlarını idare edebilir. Sayıları C karakter katarlarıyla simgeleyerek istediğimiz uzunlukta sayılar yapabiliriz. **verylong**'un içinde iki veri üyesi mevcuttur: Rakam karakter katarını tutan bir **char** dizisi ve karakter katarının ne kadar uzun olduğunu bildirmek için bir **int**. (Bu veri uzunluğu kesinkes gerekli değildir, fakat bu bizi karakter katarı uzunluğunu bulmak için tekrar tekrar **strlen()** fonksiyonunu kullanmak zorunda olmaktan kurtarır.) Karakter katarı içinde rakamlar ters sırada saklanır; en düşük öneme sahip basamak (en sağdaki) ilk önce saklanır ve **vlstr[0]** konumuna yerleştirilir. Bu, karakter katarı üzerinde gerçekleştirilecek çeşitli işlemleri kolaylaştırır. Şekil 13.2'de karakter katarı olarak saklanan bir sayı gösterilmiştir.

verylong tipinde çok uzun sayıların toplanması ve çarpılması ile ilgili kullanıcının erişebileceği rutinleri sunduk. (Çıkarma ve bölme rutinlerini yazma işini okuyucuya alıştırma olarak bırakıyoruz.)



ŞEKİL 13.2: Çok uzun bir sayı.

Sınıf Belirteci

İşte, **VERYLONG** için başlık dosyası. Bu dosya, **verylong** sınıfını açıkça belirtir.

```
// verylong.h
// cok uzun tamsayi tipi icin sinif belirteci
#include <iostream>
#include <string.h>                    //strlen() vs icin
#include <stdlib.h>                    //ltoa() icin
using namespace std;

const int SZ = 1000;
                              //cok uzun sayilarin maksimum basamak sayisi
class verylong
   {
   private:
      char vlstr[SZ];            //cok uzun sayi, karakter katarı olarak
      int vlen;                  //cok uzun sayi karakter katarının uzunlugu
      verylong multdigit(const int) const;        //ozel fonksiyonlar icin
      verylong mult10(const verylong) const;      //prototipler
   public:
      verylong() : vlen(0)                        //argumansiz kurucu fonk.
         { vlstr[0]='\0'; }
      verylong(const char s[SZ])                  // karakter katari icin
         { strcpy(vlstr, s); vlen=strlen(s); }    //tek argumanli kurucu fonk.
      verylong(const unsigned long n)             //long int icin
         {                                        //tek argumanli kurucu fonk.
         ltoa(n,vlstr,10);                        //karakter katarina donustur
         strrev(vlstr);                           //ters cevir
         vlen=strlen(vlstr);                      //uzunlugu hesapla
         }
      void putvl() const;                 //cok uzun sayiyi goster
      void getvl();                       //cok uzun sayiyi kullanicidan al
      verylong operator + (const verylong);       //cok uzun sayilari topla
      verylong operator * (const verylong);       //cok uzun sayilari carp
};
```

verylong sınıfında veri üyelerine ek olarak iki adet özel üye fonksiyonu mevcuttur. Bu fonksiyonlardan biri **verylong** tipindeki çok uzun bir sayıyı bir rakamla çarpar, diğeri ise **verylong** sayıyı 10 ile çarpar. Bu rutinler, çarpma rutini tarafından dahili olarak kullanılır.

Üç adet kurucu fonksiyon vardır. Bunlardan biri, dizinin başına, diziyi sonlandıran bir sıfır karakteri ekleyerek ve **verylong**'un uzunluğunu 0'a ayarlayarak **verylong**'a 0 değerini verir. İkinci kurucu fonksiyon **verylong**'a (ters sıralı) bir karakter katarı değeri atar ve üçüncüsü de bir **long int** değeri atar.

putvl() üye fonksiyonu bir **verylong** sayıyı görüntüler ve **getvl()** fonksiyonu da kullanıcıdan bir **verylong** sayı alır. 1,000'e kadar istediğiniz kadar çok sayıda basamak yazabilirsiniz. Bu rutinde hata kontrolü yapılmadığına dikkat edin; rakam olmayan bir şey tuşladığınızda sonuç hatalı olacaktır.

Aşırı yüklenmiş + ve * operatörleri, toplama ve çarpma işlemlerini gerçekleştirirler. **verylong** aritmetik yapmak için şu şekilde ifadeler kullanabilirsiniz:

```
alpha = beta * gamma + delta;
```

Üye Fonksiyonlar

Üye fonksiyon tanımlarını tutan **VERYLONG.CPP** dosyası şöyledir:

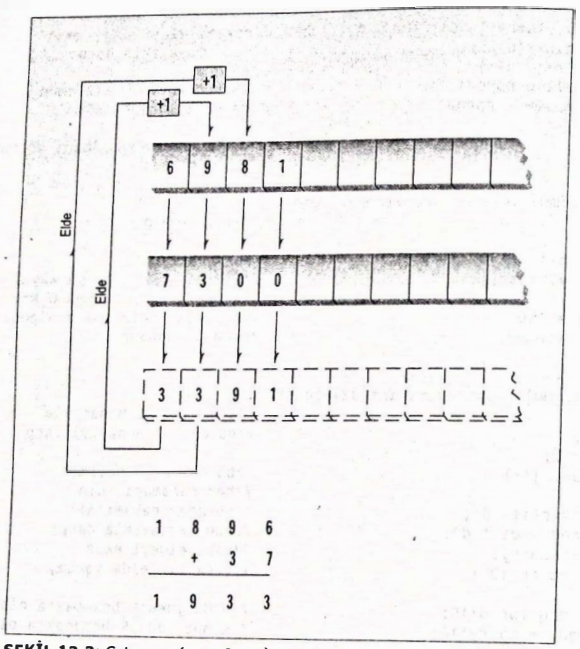
```
// verylong.cpp
// cok uzun tamsayi tipini uygular
#include "verylong.h "                  //verylong icin baslik dosyasi
//-----------------------------------------------------------------
void verylong::putvl() const            //verylong'u goster
   {
   char temp[SZ];
   strcpy(temp,vlstr);                  //kopya cikar
   cout << strrev(temp);                //kopyayi ters cevir
   }                                    //ve ekranda goster
//-----------------------------------------------------------------
void verylong::getvl()                  //kullanicidan cok uzun sayiyi al
   {
   cin >> vlstr;                        //kullanicidan karakter katarini al
   vlen = strlen(vlstr);                //uzunlugunu bul
   strrev(vlstr);                       //ters cevir
   }
//-----------------------------------------------------------------
verylong verylong::operator + (const verylong v)  //verylong'lari topla
   {
   char temp[SZ];
   int j;
                                        //en uzun sayiyi bul
   int maxlen = (vlen > v.vlen) ? vlen : v.vlen;
   int carry = 0;                       //eger toplam >= 10 ise, 1'e ayarlanir
   for(j=0; j<maxlen; j++)              //her basamak icin
      {
      int d1 = (j > vlen-1)   ? 0 : vlstr[j]-'0';          //rakami al
      int d2 = (j > v.vlen-1) ? 0 : v.vlstr[j]-'0';        //rakami al
      int digitsum = d1 + d2 + carry;                      //rakamlari topla
      if( digitsum >= 10 )                                 //elde varsa,
         { digitsum -= 10; carry=1; }                      //toplami 10 azalt,
      else                                                 //eldeye 1 degerini ver
         carry = 0;                                        //aksi halde elde 0'dir
      temp[j] = digitsum+'0';                              //karakteri katara ekle
      }
   if(carry==1)
      temp[j++] = '1';                                     //en sonda elde varsa,
                                                           //son rakam 1'dir
```



```
   temp[j] = '\0';
   return verylong(temp);               //karakter katarini sonlandir
   }                                    //gecici sayiyi dondur
//-----------------------------------------------------------------
verylong verylong::operator * (const verylong v)  //cok uzun sayilari
   {                                               //carp
   verylong pprod;
   verylong tempsum;                               //tek basamagin carpimi
   for(int j=0; j<v.vlen; j++)                     //elde tasinan toplam
      {                                            //argumandaki her basamak icin
      int digit = v.vlstr[j]-'0';
      pprod = multdigit(digit);                    //rakami al
      for(int k=0; k<j; k++)                       //bunu basamakla carp
         pprod = mult10(pprod);                    //sonucu
      tempsum = tempsum + pprod;                   //10'nun kuvveti ile carp
      }                                            //carpimi toplama ekle
   return tempsum;
   }                                               //carpimlarin toplamini dondur
//-----------------------------------------------------------------
verylong verylong::mult10(const verylong v) const //argumani
   {                                               //10 ile carp
   char temp[SZ];
   for(int j=v.vlen-1; j>=0; j--)                  //basamaklari
      temp[j+1] = v.vlstr[j];                      //bir basamak ileriye kaydir
   temp[0] = '0';                                  //sagdaki en uc kisma 0 koy
   temp[v.vlen+1] = '\0';                          //karakter katarini sonlandir
   return verylong(temp);                          //sonucu dondur
   }
//-----------------------------------------------------------------
verylong verylong::multdigit(const int d2) const
   {                                               //argumandaki basamakla
   char temp[SZ];                                  //bu cok uzun sayiyi carp
   int j, carry = 0;
   for(j=0; j<vlen; j++)                           //bu cok uzun sayinin
      {                                            //her basamagi icin
      int d1 = vlstr[j]-'0';                       //buradan rakami al
      int digitprod = d1 * d2;                     //bunu basamakla carp
      digitprod += carry;                          //eski eldeyi ekle
      if( digitprod >= 10 )                        //yeni bir elde varsa,
         {
         carry = digitprod/10;                     //elde yuksek basamakta olsun
         digitprod -= carry*10;                    //sonuc, dusuk basamakta olsun
         }
      else
         carry = 0;                                //aksi halde, elde 0'dir
      temp[j] = digitprod+'0';                     //karakteri katara ekle
      }
   if(carry != 0)                                  //elde, sondaysa,
      temp[j++] = carry+'0';                       //son basamakta demektir
   temp[j] = '\0';                                 //karakter katarini sonlandir
   return verylong(temp);                          //cok uzun sayiyi dondur
   }
```

putvl() ve **getvl()** fonksiyonları oldukça nettir. Bu fonksiyonlar C karakter katarını ters çevirmek için **strrev()** C kütüphane fonksiyonunu kullanırlar. Yani, C karakter katarı, tersten sıralı olarak saklanır ama ekranda normal haliyle gösterilir.

`operator+()` fonksiyonu iki adet `verylong` sayıyı toplar ve sonucu üçüncü bir `verylong`'a bırakır. Bu işlemi, basamakları her seferinde tek tek ele alarak gerçekleştirir. Her ikı sayının 0. basamağını toplar, elde varsa eldeyi saklar. Sonra, 1. basamaktaki rakamları toplar, gerekiyorsa eldeyi ekler. İki sayıdan büyük olanın tüm basamakları toplanana dek bu işlemi sürdürür. Eğer sayılar farklı uzunluktaysa, toplamadan önce kısa sayıda mevcut olmayan basamaklara 0 değeri verilir. Bu işlem Şekil 13.3'te gösterilmiştir.



ŞEKİL 13.3: Çok uzun (`verylong`) sayıları toplamak.

Çarpma için `operator*()` fonksiyonu kullanılır. Bu fonksiyon, çarpılan sayıyı (elle yazdığınızda üstteki sayı), çarpanın (alttaki sayı) her basamağı ile ayrı ayrı çarpar. Bunu gerçekleştirmesi için `multdigit()` rutinini çağırır. Daha sonra, basamakları doğru yerleriyle eşlemek amacıyla, sonuçlar `mult10()` fonksiyonu kullanılarak birkaç kez (uygun sayıda) 10 ile çarpılır. Bu ayrı hesaplamaların sonuçları sonradan `operator+()` fonksiyonu kullanılarak beraber toplanır.

Uygulama Programı

`verylong` sınıfını sınamak için, kullanıcı tarafından girilen bir sayının faktöriyelini hesaplamak amacıyla "Döngüler ve Kararlar" adlı Bölüm 3'teki `FACTOR` programının bir varyasyonunu kullanıyoruz. `VLAPP.CPP` programının listesi şöyledir:


```
// vl_app.cpp
// verylong sinifini kullanarak cok uzun sayilarin faktoriyelini hesaplar
#include "verylong.h"                          //verylong baslik dosyasi

int main()
   {
   unsigned long numb, j;
   verylong fact=1;
                                               //verylong'u ilk kullanima hazirla
   cout << "\n\nEnter number: ";
   cin >> numb;
                                               //bir long int gir
   for(j=numb; j>0; j--)
      fact = fact * j;                         //faktoriyel = numb *
   cout << "Factorial is ";                    //numb-1 * numb-2 *
   fact.putvl();                               //numb-3 vs.
   cout << endl;                               //faktoriyeli goster
   return 0;
   }
```

Bu programda `fact` bir `verylong` değişkenidir. Diğer değişkenler, `numb` ve `j`, `verylong` olmak zorunda değildir çünkü bu değişkenler bu kadar büyümezler. Örneğin 100'ün faktöriyelini hesaplamak amacıyla, `numb` ve `j` için sadece üç basamak gereklidir. Oysa, `fact` 158 basamak gerektirir.

Şu ifadeye dikkat edin:

```
fact = fact * j;
```

Burada, çarpma işlemi gerçekleştirilmeden önce, bir `long` değişkeni olan `j`, tek argümanlı kurucu fonksiyon vasıtasıyla otomatik olarak `verylong`'a çevrilir.

Programdan 100'ün faktöriyelini hesaplamasını istediğimizde program çıktısı şöyle olur:

```
Enter number: 100
Factorial is 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000
```

`long` tipinde değişkenler kullanarak *bunu* deneyin! Şaşırtıcı ama, rutinler oldukça hızlıdır. Bu program bir saniyeden daha az bir sürede çalışmasını tamamlar. Programın 1000 basamak kapasitesini aşmadan önce, 400'e kadar olan sayıların faktöriyellerini hesaplayabilirsiniz.

Bir Yükselen Asansör Simülasyonu

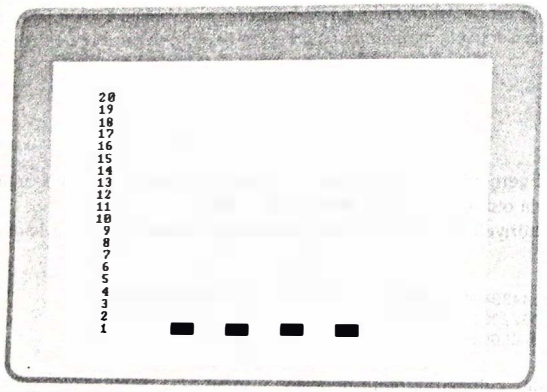
Yüksek bir işyerinde bir daha asansör beklediğinizde, asansörün nereye gideceğine nasıl karar verdiğini kendinize sorun. Eski günlerde, elbette asansör kabinlerinin her birinde bir asansör görevlisi vardı ("Günaydın, Mr. Burberry," "Günaydın, Carl.") Asansöre binenlerin, daha binerken bu görevliye hangi katta ineceklerini söylemeleri gerekiyordu ("Onyedi, lütfen."). Asansör kabini içindeki ışıklı bir göstergede aşağı ya da yukarı yönde hangi katların hizmet istediğini gösteren ışıklar yanıyordu. Asansör görevlileri, bu sözlü talepleri baz alarak ve göstergedeki ışıkları takip ederek hangi yönde gidileceğine ve nerede durulacağına karar veriyorlardı.

Günümüzde asansör sistemleri için asansör kabinlerinin kendi kendilerine işleyebilmelerine imkan veren yeterli zeka geliştirilmiştir. Bir sonraki örneğimizde bir asansör sistemini modellemek için C++ sınıflarını kullanıyoruz.

Böyle bir sistemin bileşenleri nelerdir? Tipik bir binada birkaç tane benzer asansör vardır. Her katta yukarı ve aşağı düğmeleri bulunur. Dikkat ederseniz, her kat için bu düğmelerden genellikle bir çift vardır; düğmeye bastığınızda hangi asansörün sizin için duracağını bilemezsiniz. Asansörün içinde daha fazla sayıda düğme vardır: Her kat için bir tane. Asansöre girdikten sonra asansöre binenler inecekleri katı belirten düğmeye basarlar. Simülasyon programımız bütün bu bileşenleri modelleyecek.

ELEV Programını Çalıştırmak

ELEV programını çalıştırdığınızda, ekranın en altında dört tane asansörün oturduğunu ve sol tarafta, ekranın altından yukarısına doğru ilerleyen 1'den 20'ye kadar sayıların bir listesini göreceksiniz. Asansörler başlangıçta zemin (birinci) kattalar. Bu durum Şekil 13.4'te gösterilmiştir.



ŞEKİL 13.4: **ELEV** programının başlangıç ekranı.

Kat Talebinde Bulunmak

Enter'a basarsanız, ekranın altındaki metin sizi yönlendirir:

```
Enter the floor you 're on:
```

1'den 20'ye kadar herhangi bir kat numarasını girebilirsiniz. Eğer işe yeni gelmişseniz, zemin katta 1'e basarsınız. Eğer yüksek bir katı terk edip yemeğe gidiyorsanız, kendi katınızın numarasını girersiniz. Bir sonraki yönlendirme şöyle olacaktır:

```
Enter direction you want to go (u or d):
```

Eğer birinci kattaysanız yukarı çıkmanız, 20. kattaysanız aşağı inmeniz gerekir. Ara katlarda iki yönde de gidebilirsiniz. Kat talebinde bulunurken, solda ilgili kat numarasının yanında bir

üçgen belirir. Talepte bulunduğunuz yöne bağlı olarak bu üçgen yukarıyı ya da aşağıyı gösterir. Talepler arttıkça ilaveten diğer kat numaralarını yanında da üçgenler belirecektir.

Eğer talepte bulunulan bir katta zaten bir asansör kabini bulunuyorsa kapı hemen açılır. Kabinin dışında ortaya çıkan, sonra açık kapıdan içeri giren bir gülen yüz karakteri görürsünüz. Talepte bulunulan katta eğer asansör kabini yoksa kabinlerden biri yukarı veya aşağı yönde hareket eder ve kata ulaşır ulaşmaz kapısını açar.

Gidilecek Katları Girmek

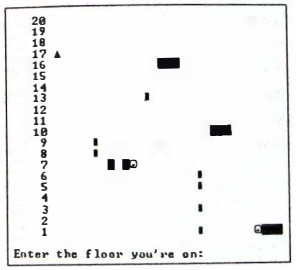
Bir kabin kata ulaştığında ve gülen yüzlü yolcumuz içeri girdiğinde, ekranın alt tarafında bir yönlendirme mesajı belirir:

```
Car 1 has stopped at floor 1
Enter destination floors (0 when finished)
Destination 1: 13
```

Burada yolcu 13 girmiş. Bu arada, gülen yüz bir kerede asansöre binen birden fazla yolcuyu da simgeleyebilir. Her yolcu gitmek için farklı bir katı talep edebilir. Yani program, birden fazla gidilecek katın girilmesine imkan verir. İstediğiniz kadar çok sayı (en az 1 tane ama 20'den de fazla olmasın) girin; bitirdiğinizde ise 0 girin.

Belirli bir kabin içindeki yolcuların talep ettikleri katlar kabinin dışında, tam sol tarafında, talep edilen kat numarasının yanında küçük dikdörtgenlerle gösterilir. Her kabinin kendisine ait gidilecek katlar listesi vardır (Bu, tüm kabinler tarafından paylaşılan kat taleplerinden farklıdır).

İstediğiniz sayıda kat talebinde bulunabilirsiniz. Sistem kat taleplerini, her kabinde seçilen gidiş katlarıyla birlikte hatırlar ve tümüne hizmet vermeye çalışır. Kabinlerin dördü de aynı anda hareket halinde olabilir. Şekil 13.5'te birden fazla kat talebi ve gidilecek kat olduğunda ortaya çıkan durum gösterilmiştir.



ŞEKİL 13.5: Hareket halindeki asansörler.

Sistemi Tasarlamak

Asansör kabinlerinin tümü aşağı yukarı aynıdır. Bu nedenle, bunları **elevator** adında tek bir sınıfın nesneleri yapmak mantıklı görünür. Bu sınıf, her kabine özgü verileri içerecektir: Mevcut konumu, gideceği yön, binenlerin talep ettiği gidilecek katların numaraları vs.

Bununla birlikte, binanın bütünü ilgilendiren başka veriler de vardır. Bu veriler `building` sınıfının bir parçası olacaklardır. Öncelikle *kat taleplerini* tutan bir dizi vardır. Bu, asansörü bekleyen kişilerin, asansörün kendi katlarında durmasını talep etmek için yukarı veya aşağı düğmesine bastıkları katların listesidir. Asansörlerden herhangi biri böyle bir talebe cevap verebilir. Bu nedenle, asansörlerin de birbirlerinden haberleri olması lazım. `Nx2`'lik `bool` tipinde bir dizi kullanılır. `N`, kat sayısını gösterir; `2` ise her kat için aşağı ve yukarı yönleri tutmak için ayrı dizi elemanları kullanmaya imkan verir. Asansörlerin tümü bir sonraki adımda nereye gideceklerini kestirebilmek için bu diziye bakabilirler.

Asansörlerin her birinin, kat taleplerini bilmenin yanı sıra ayrıca, diğer asansörlerin de nerede olduğundan haberdar olmaları lazımdır. Eğer birinci kattaysak ve 10. katta müsait bir asansör zaten varsa 15. kattan gelen bir talebe cevap vermek için telaş etmenin bir manası yoktur. En yakın asansör kabini talebe cevap vermek için yönlenmelidir. Her kabinin diğerlerinden haberdar olmasını en kolay şekilde sağlamak için `building` içinde ikinci veri öğesi olarak asansörlere işaret eden bir işaretçi dizisi mevcuttur. Her asansör kabini ilk kez oluşturulurken kendi bellek adresini bu listeye yerleştirir. Böylece diğer kabinler kendisini bulabilir.

`building` sınıfındaki üçüncü veri öğesi ise şimdiye kadar üretilen kabinlerin sayısıdır. Bu, kabinler üretilirken her kabinin kendisini sırayla numaralandırmasına imkan verir.

Zamanı Kontrol Etmek

`main()` programı, kabinleri çalıştırmak için `building`'in bir üye fonksiyonunu sabit aralıklarla çağırır. Bu fonksiyon `master_tick()` olarak adlandırılır. Bu fonksiyon, sırası geldiğinde, her asansör kabini için `car_tick1()` adında bir fonksiyon çağırır. `car_tick1()`, diğer işlerin yanı sıra kabinlerin her birini ekranda gösterir ve kabinin bir sonraki adımda ne yapması gerektiğine karar vermesi için bir başka fonksiyonu çağırır. Seçenekler yukarıya çıkmak, aşağıya inmek, durmak, yolcu almak veya yolcu indirmek olabilir.

Her kabin daha sonra kendi konumuna döndürülmelidir. Yine de, işler burada biraz daha karmaşık bir hal alır. Her kabin, ne yapacağına karar vermeden önce diğerlerinin nerede olduğunu saptamak zorunda olduğu için kabinlerden herhangi biri harekete geçmeden önce, kabinlerin tümü karar verme sürecinden geçmelidir. Bunun gerçeklendiğinden emin olmak için her kabin için iki adet saat sinyali kullanılır. Böylece, kabinlerin her birinin nereye gideceğine karar vermek için `car_tick1()` fonksiyonunun çağrılmasından sonra kabinleri gerçekten hareket ettirmek için bir başka fonksiyon olan `car_tick2()`, çağrılır. Bu fonksiyon, `current_floor` değişkenini değiştirerek kabinlerin hareket etmesini sağlar.

Yolcu alma işlemi, kabinin istenilen katta durması sırasında, sabit bir dizi adım takip edilerek gerçeklenir. Program sırayla şu şekilde ilerler:

1. Kabinin kapısı kapalı, gülen yüz yok.
2. Kabinin kapısı açık, gülen yüz sol tarafta.
3. Gülen yüz kabinin kapısında, kullanıcıdan gidilecek katı öğren.
4. Kabinin kapısı kapalı, gülen yüz yok.

Yolcuyu indirmek için bu adımlar ters sırayla takip edilir. Bu adımlar şu şekilde sürdürülür: Kronometre (bir tamsayı değişkeni) başlatılır; kronometrenin 3'ten 0'a geri sayması sağlanır ve her saat sinyali ile kronometrenin gösterdiği değer bir azaltılır. `car_display()` fonksiyonu içindeki bir `case` ifadesi daha sonra bu sürecin her aşaması için ilgili kabinin resmini ekrana çizer.

`ELEV` programı, çeşitli konsol grafik fonksiyonları kullandığı için bu kitabın yayıncısından elde edilebilecek bir başlık dosyasını gerektirir. Bu başlık dosyası ya `MSOFTCON.H` (Microsoft

derleyiciler için) ya da `BORLACON.H` (Borland derleyiciler için) olabilir. ("Console Graphics Lite" adlı Ek E'ye bakın.)

ELEV Programının Listeleri

Programı dört dosyaya böldük. Bu dosyalardan, `ELEV.H` ve `ELEV.CPP` adlarına sahip olan ikisi, satıcı tarafından tedarik edilen asansör kontrol yazılımı tarafından üretilebilir. Bu yazılım sonradan, belirli bir bina için asansör sistemi tasarlamakla ilgilenen bir mühendislik şirketi tarafından satın alınacaktır. (Bu Program, Ulusal Asansör Kurulu tarafından onaylanmamıştır; bu nedenle, bunu gerçek asansörlerle denemeyin.) Mühendislik şirketi daha sonra başka bir çift dosya daha yazacaktır: `ELEV_APP.H` ve `ELEV_APP.CPP`. `ELEV_APP.H` dosyası, yüksek binanın özelliklerini belirtir. Bunun ayrı bir dosya olması gerekir, çünkü bu özelliklerin, asansör sınıfının üye fonksiyonları tarafından bilinmesi gerekir ve bunu gerçekleştirmenin en kolay yolu `ELEV_APP.H` dosyasını `ELEV_APP.CPP` dosyasına dahil etmektir. `ELEV_APP.CPP` dosyası, asansörleri ilk kullanıma hazırlar ve sonra, geçen zamanı canlandırmak için sabit aralıklarla asansör fonksiyonlarını çağırır.

Sınıf Belirteci

`ELEV.H` dosyası, asansör sınıfı için spesifikasyonları içerir. Asansörlere işaret eden işaretçi dizisi olan `car_list[]`, her kabinin diğerlerini, bulundukları konum ve gidecekleri yön hakkında sorgulamasına imkan verir. Programın listesi aşağıda görülüyor:

```
// elev.h
// asansorler icin baslik dosyasi -- sinif deklarasyonlarini icerir
#include "elev_app.h"                  //musteri tarafindan saglanir
#include "msoftcon.h"                  //konsol grafikleri icin
#include <iostream>
#include <iomanip>                     //setw() icin
#include <conio.h>                     //ekran ciktisi icin
#include <stdlib.h>                    //itoa() icin
#include <process.h>                   //exit() icin
using namespace std;

enum direction { UP, DN, STOP };
const int LOAD_TIME =  3;              //indirme/bindirme suresi(saat sinyali)
const int SPACING =    7;              //kabinler arasi sanal bosluk
const int BUF_LENGTH = 80;             //yardimci karakter katari tampon uzunlugu
class  building;                       //ileriye yonelik deklarasyon
////////////////////////////////////////////////////////////////
class elevator
   {
   private:
   building* ptrBuilding;              //temel building sinifina isaretci
   const int car_number;               //bizim sayi (0'dan nc-1'e kadar)
   int current_floor;                  //neredeyiz? (0'dan nf-1'e kadar)
   int old_floor;                      //neredeydik? (0'dan nf-1'e kadar)
   direction current_dir;              //hangi yone gidecegiz?
   bool destination[NUM_FLOORS];       //asansore binenler tarafindan secilir
   int loading_timer;                  //eger binen varsa, sifirdan farkli
   int unloading_timer;                //eger inen varsa, sifirdan farkli

   public:
   elevator(building*, int);           //kurucu fonksiyon
```

```
    void car_tick1();                   //her kabin icin 1. saat sinyali
    void car_tick2();                   //her kabin icin 2. saat sinyali
    void car_display();                 //asansoru ekrana ciz
    void dests_display() const;         //asansor taleplerini ekrana ciz
    void decide();                      //ne yapilacagina karar ver
    void move();                        //kabini harekete gecir
    void get_destinations();            //gidilecek katlari ogren
    int get_floor()const;               //su anki kati ogren
    direction get_direction() const;    //su anki yonu ogren
    };
////////////////////////////////////////////////////////////////
class building
    {
    private:
    elevator* car_list[NUM_CARS];       //kabinlere isaret eden isaretciler
    int num_cars;                       //simdiye dek uretilen kabinler
                                        //yukari/asagi dugmelerinin dizisi
    bool floor_request[2][NUM_FLOORS];  //false=UP, true=DN

    public:
    building();                         //kurucu fonk.
    ~building();                        //yok edici fonk.
    void master_tick();                 //tum kabinlere saat sinyalini gonder
    int get_cars_floor(const int) const;    //kabinin nerede oldugunu bul
                                        //kabinin gidecegi yonu bul
    direction get_cars_dir(const int) const;
                                        //spesifik kat talebini kontrol et
    bool get_floor_req(const int, const int) const;
                                        //spesifik kat talebini ayarla
    void set_floor_req(const int, const int, const bool);
    void record_floor_reqs();           //kat taleplerini ogren
    void show_floor_reqs()const;        //kat taleplerini goster
    };
```

Üye Fonksiyonlar

ELEV.CPP dosyası, **elevator** sınıfı ve **building** sınıfı üye fonksiyonlarını ve verilerini içerir. **building** içindeki fonksiyonlar sistemi ilk kullanıma hazırlar, bir ana saat sinyali sağlar, kat taleplerini ekranda görüntüler ve kullanıcıdan kat taleplerini alır. **elevator** içindeki fonksiyonlar kabinleri tek tek ilk kullanıma hazırlar (kurucu fonksiyon ile), her kabin için iki adet saat sinyali sağlar, kabinleri ekranda gösterir, kabinlerin gidecekleri katları gösterir, ne yapılacağına karar verir, kabini yeni bir kata doğru hareket ettirir ve kullanıcıdan gidilecek katları öğrenir. Programın listesi aşağıda görülmektedir:

```
// elev.cpp
// sinif verilerini ve uye fonksiyonlarini icerir

#include "elev.h"               //sinif deklarasyonlarini dahil et
////////////////////////////////////////////////////////////////
//       building sinifi icin fonksiyon tanimlari
////////////////////////////////////////////////////////////////
building::building()                    //kurucu fonk.
   {
   char ustring[BUF_LENGTH];            //kat numaralari icin karakter katari

   init_graphics();
   clear_screen();                      //grafikleri ilk kullanima hazirla
                                        //ekrani temizle
```



```
   num_cars = 0;
   for(int k=0; k<NUM_CARS; k++)        //asansorleri olustur
      {
      car_list[k] = new elevator(this, num_cars);
      num_cars++;
      }
   for(int j=0; j<NUM_FLOORS; j++)      //her kat icin
      {
      set_cursor_pos(3, NUM_FLOORS-j); //ekrana kat
      itoa(j+1, ustring, 10);          //numarasini yerlestir
      cout << setw(3) << ustring;
      floor_request[UP][j] = false;    //kat talebi henuz gelmedi
      floor_request[DN][j] = false;
      }
   } //kurucu fonksiyonun sonu
//--------------------------------------------------------------
building::~building()                   //yok edici fonk.
   {
   for(int k=0; k<NUM_CARS; k++)
      delete car_list[k];
   }
//--------------------------------------------------------------
void building::master_tick()            //ana saat sinyali
   {
   int j;
   show_floor_reqs();                   //kat taleplerini ekrana ciz
   for(j=0; j<NUM_CARS; j++)            //her asansor icin
      car_list[j]->car_tick1();         //1. saat sinyalini gonder
   for(j=0; j<NUM_CARS; j++)            //her asansor icin
      car_list[j]->car_tick2();         //2. saat sinyalini gonder
   } //master_tick()'in sonu
//--------------------------------------------------------------
void building::show_floor_reqs()const //kat taleplerini ekranda ciz
   {
   for(int j=0; j<NUM_FLOORS; j++)
      {
      set_cursor_pos(SPACING, NUM_FLOORS-j);
      if(floor_request[UP][j]==true)
         cout << '\x1E';                //yukari yonlu ok
      else
         cout << ' ';
      set_cursor_pos(SPACING+3, NUM_FLOORS-j);
      if(floor_request[DN][j]==true)
         cout << '\x1F';                //asagi yonlu ok
      else
         cout << ' ';
      }
   } //show_floor_reqs()'in sonu
//--------------------------------------------------------------
//record_floor_reqs() -- kabinin disindaki yolculardan talepleri al
void building::record_floor_reqs()
   {
   char ch = 'x';                       //girdi icin yardimci karakter
   char ustring[BUF_LENGTH];            //girdi icin yardimci karakter katari
   int iFloor;                          //talebin yapildigi kat
   char chDirection;                    //'u' veya 'd' yukari ve asagi icin

   set_cursor_pos(1,22);                //ekranin en alti
   cout << "Press [Enter] to call an elevator: ";
```

```
      if( !kbhit() )                    //tus basilana kadar bekle(CR olmali)
         return;
      cin.ignore(10, '\n');
      if(ch=='\x1B')                    //escape tusu ise, programi sonlandir
         exit(0);
      set_cursor_pos(1,22); clear_line(); //eski metni temizle
      set_cursor_pos(1,22);             //ekranin en alti
      cout << "Enter the floor you 're on: ";
      cin.get(ustring, BUF_LENGTH);     //kati ogren
      cin.ignore(10, '\n');           //newline karakteri de dahil karakterleri al
      iFloor = atoi(ustring);         //tamsayiya cevir

      cout << "Enter direction you want to go (u or d): ";
      cin.get(chDirection);             //(birden fazla linefeed'e engel ol)
      cin.ignore(10, '\n');           //newline karakteri de dahil karakterleri al

      if(chDirection=='u' || chDirection=='U')
         floor_request[UP][iFloor-1]==true;   //yukariya kat talebi
      if(chDirection=='d' || chDirection=='D')
         floor_request[DN][iFloor-1]==true;   //asagiya kat talebi
      set_cursor_pos(1,22); clear_line();     //eski metni temizle
      set_cursor_pos(1,23); clear_line();
      set_cursor_pos(1,24); clear_line();
      } //record_floor_reqs()'in sonu
//-------------------------------------------------------------
//get_floor_req() -- spesifik bir talep olup olmadigini gor
bool building::get_floor_req(const int dir,
                             const int floor) const
   {
   return floor_request[dir][floor];
   }
//-------------------------------------------------------------
//set_floor_req() -- spesifik kat talebini ayarla
void building::set_floor_req(const int dir, const int floor,
                             const bool updown)
   {
   floor_request[dir][floor] = updown;
   }
//-------------------------------------------------------------
//get_cars_floor()-- kabininin nerede oldugunu bul
int building::get_cars_floor(const int carNo) const
   {
   return car_list[carNo]->get_floor();
   }
//-------------------------------------------------------------
//get_cars_dir()-- kabinin hangi yone gidecegini bul
direction building::get_cars_dir(const int carNo) const
   {
   return car_list[carNo]->get_direction();
   }
//-------------------------------------------------------------

//////////////////////////////////////////////////////////////
//         elevator sinifi için sinif tanimlari
//////////////////////////////////////////////////////////////
                                    //kurucu fonksiyon
elevator::elevator(building* ptrB, int nc) :
  {                                 ptrBuilding(ptrB), car_number(nc)
```



```
   current_floor = 0;
   old_floor = 0;                   //0'dan basla (kullanici 1)
   current_dir = STOP;              //onceki kati hatirla
   for(int j=0; j<NUM_FLOORS; j++)  //baslangicta hareketsiz
      destination[j] = false;       //binenler henuz hicbir
   loading_timer = 0;               //dugmeye basmadi
   unloading_timer = 0;             //henuz binen yok
   } //kurucu fonksiyonun sonu      //henuz inen yok
//-------------------------------------------------------------
int elevator::get_floor() const     //su anki kati ogren
   {
   return current_floor;
   }
//-------------------------------------------------------------
direction elevator::get_direction() const  //su anki yonu
   {                                       //ogren
   return current_dir;
   }
//-------------------------------------------------------------
void elevator::car_tick1()          //her kabin icin 1. sinyal
   {
   car_display();                   //asansor kutusunu ciz
   dests_display();                 //gidilecek katlari ekranda goster
   if(loading_timer)                //yukleme zamanini say
      --loading_timer;
   if(unloading_timer)              //yolculari indirme zamanini say
      --unloading_timer;
   decide();                        //ne yapilacagina karar ver
   } //car_tick()'in sonu
//-------------------------------------------------------------
//kabinlerin her biri, diğerleri hareket etmeden once karar vermelidir
void elevator::car_tick2()          //her kabin icin 2. sinyal
   {
   move();                          //musaitse asansoru harekete gecir
   }
//-------------------------------------------------------------
void elevator::car_display()        //asansor seklini ciz
   {
   set_cursor_pos(SPACING+(car_number+1)*SPACING, NUM_FLOORS-old_floor);
   cout << " ";                     //eski konumu sil
   set_cursor_pos(SPACING-1+(car_number+1)*SPACING,
                                    NUM_FLOORS-current_floor);

   switch(loading_timer)
      {
      case 3:
         cout << "\x01\xDB \xDB ";  //gulen yuz solda olacak sekilde
         break;                     //kapisi acik bir kabin ciz
      case 2:
         cout << " \xDB\x01\xDB ";  //gulen yuz kabinin kapisinda
         get_destinations();        //gidilecek katlari ogren
         break;
      case 1:
         cout << " \xDB\xDB\xDB ";  //gulen yuz olmadan
         break;                     //kapisi kapali bir kabin ciz
      case 0:
         cout << " \xDB\xDB\xDB ";  //kapi kapali,
         break;                     //gulen yuz yok (varsayilan)
      }
   set_cursor_pos(SPACING+(car_number+1)*SPACING,
```

```
                                    NUM_FLOORS-current_floor);
      switch(unloading_timer)
         {
         case 3:
            cout << "\xDB\x01\xDB ";      //kapisi acik bir kabin ciz
            break;                        //gulen yuz kabininin icinde
         case 2:
            cout << "\xDB \xDB\x01";      //kapisi acik bir kabin ciz
            break;                        //gulen yuz sagda
         case 1:
            cout << "\xDB\xDB\xDB ";      //kapisi kapali bir kabin ciz
            break;                        //gulen yuz yok
         case 0:
            cout << "\xDB\xDB\xDB ";      //kapi kapali, gulen
            break;                        //yuz yok (varsayilan)
         }
      old_floor = current_floor;          //eski kati hatirla
   }  //car_display()'in sonu
//---------------------------------------------------------------
void elevator::dests_display() const   //kabinin icindeki
   {                                   //dugmelerle secilmis olan
   for(int j=0; j<NUM_FLOORS; j++)     //gidilecek katlari goster
      {
      set_cursor_pos(SPACING-2+(car_number+1)*SPACING, NUM_FLOORS-j);
      if( destination[j]===true )
         cout << '\xFE';               //kucuk kutu
      else
         cout << ' ';                  //bos
      }
   }  //dests_display()'in sonu
//---------------------------------------------------------------
void elevator::decide()                //ne yapilacagina karar ver
   {
   int j;
   //isaretler, gidilecek katlarin veya taleplerin asagimizda mi,
   //yoksa yukarimizda mi kaldigini belirtir
   bool destins_above, destins_below;      //gidilecek katlar
   bool requests_above, requests_below;    //talepler
   //yukarimizda ve asagimizdaki en yakin talebin kat numarasi
   int nearest_higher_req = 0;
   int nearest_lower_req = 0;
   //isaretler, bizle talep edilen en yakin kat (FR) arasinda ayni
   //yonde giden baska bir kabin olup olmadigini belirtir
   bool car_between_up, car_between_dn;
   //en yakin FR'in karsi tarafinda, karsi yonde hareket eden baska bir kabin
   //olup olmadigini belirtir
   bool car_opposite_up, car_opposite_dn;
   //baska bir kabinin kat numarasi ve yonu (bizimki degil)
   int ofloor;                         //kat
   direction odir;                     //yon

   //cok yukariya veya cok asagiya gitmedigimizden emin olalim
   if( (current_floor==NUM_FLOORS-1 && current_dir==UP)
      || (current_floor==0 && current_dir==DN) )
      current_dir = STOP;

   //bu katta inecek varsa, yolculari indir
   if( destination[current_floor]==true )
      {
```



```
      destination[current_floor] = false;  //Gidilen kati temizle
      if( !unloading_timer )               //yolculari indir
         unloading_timer = LOAD_TIME;
      return;
      }
   //bu katta yukari kat talebi varsa ve biz yukari cikiyorsak veya
   //duruyorsak, yolculari indir
   if( (ptrBuilding->get_floor_req(UP, current_floor) &&
         current_dir != DN) )
      {
      current_dir = UP;                    //(eger durdurulduysa)
      //bizim gittigimiz yondeki kat talebini sil
      ptrBuilding->set_floor_req(current_dir,
                                 current_floor, false);
      if( !loading_timer )                 //yolculari bindir
         loading_timer = LOAD_TIME;
      return;
      }
   //eger bu katta asagi kat talebi varsa ve biz asagiya iniyorsak,
   //veya durmussak, yolculari bindir
   if( (ptrBuilding->get_floor_req(DN, current_floor) &&
         current_dir != UP) )
      {
      current_dir = DN;                    //(eger durdurulduysa)
      //bizim gittigimiz yondeki kat talebini sil
      ptrBuilding->set_floor_req(current_dir,
                                 current_floor, false);
      if( !loading_timer )                 //yolculari yukle
         loading_timer = LOAD_TIME;
      return;
      }
   //gidilecek baska kat olup olmadigini veya talep olup olmadigini kontrol et
   //en yakin talebe olan uzakligi kaydet
   destins_above = destins_below = false;
   requests_above = requests_below = false;
   for(j=current_floor+1; j<NUM_FLOORS; j++)
      {                                    //ustteki katlari kontrol et
      if( destination[j] )                 //eger gidilecek katsa
         destins_above = true;             //degiskeni ayarla
      if(ptrBuilding->get_floor_req(UP, j) ||
         ptrBuilding->get_floor_req(DN, j) )
         {                                 //talep varsa
         requests_above = true;            //degiskeni ayarla
         if( !nearest_higher_req )         //onceden ayarlanmamissa,
            nearest_higher_req = j;        //en yakin talebi ayarla
         }
      }
   for(j=current_floor-1; j>=0; j--)       //asagidaki katlari kontrol et
      {
      if( destination[j] )                 //eger gidilecek katsa
         destins_below = true;             //degiskeni ayarla
      if(ptrBuilding->get_floor_req(UP, j) ||
         ptrBuilding->get_floor_req(DN, j) )
         {                                 //talep varsa
         requests_below = true;            //degiskeni ayarla
         if( !nearest_lower_req )          //onceden ayarlanmamissa,
            nearest_lower_req = j;         //en yakin talebi ayarla
         }
      }
```

```
   //yukari veya asagi yonde kat talebi veya gidilecek kat istegi varsa, dur
   if( !destins_above && !requests_above &&
      !destins_below && !requests_below )
      {
      current_dir = STOP;
      return;
      }
   //gidilecek katsa ve biz durmussak veya zaten dogru yonde gidiyorsak,
   //gidilecek kata dogru ilerle
   if( destins_above && (current_dir==STOP || current_dir==UP) )
      {
      current_dir = UP;
      return;
      }
   if( destins_below && (current_dir==STOP || current_dir==DN) )
      {
      current_dir = DN;
      return;
      }
   //diger kabinler olup olmadigini arastir: (a) bizle ve en yakin talebi
   //arasinda ayni yonde giden,(b)kat talebin diger tarafinda ters yonde giden
   car_between_up = car_between_dn = false;
   car_opposite_up = car_opposite_dn = false;

   for(j=0; j<NUM_CARS; j++)              //her kabini kontrol et
      {
      if(j != car_number)                 //biz degilsek
         {                                //katini
         ofloor = ptrBuilding->get_cars_floor(j);        //ve
         odir = ptrBuilding->get_cars_dir(j);            //yonunu ogren

         //eger yukari gidiyorsa ve bizim ustumuzdeki katlardan talep varsa
         if( (odir==UP || odir==STOP) && requests_above )
            //eger bizim yukarimizda ve en yakin katin asagisindaysa
            if( (ofloor > current_floor
               && ofloor <= nearest_higher_req)
            //veya bizimle ayni kattaysa fakat kabin numarasi kucukse
               || (ofloor==current_floor && j < car_number) )
               car_between_up = true;
         //eger asagiya gidiyorsa ve bizim altimizdaki katlardan talep varsa
         if( (odir==DN || odir==STOP) && requests_below )
            //eger bizim asagimizda ve en yakin katin yukarisindaysa
            if( (ofloor < current_floor
               && ofloor >= nearest_lower_req)
               //veya bizimle ayni kattaysa fakat kabin numarasi kucukse
               || (ofloor==current_floor && j < car_number) )
               car_between_dn = true;
         //eger yukari gidiyorsa ve bizim altimizdaki katlardan talep varsa
         if( (odir==UP || odir==STOP) && requests_below )
            //talebin altindaysa ve bizden daha yakinsa
            if(nearest_lower_req >= ofloor
               && nearest_lower_req - ofloor
                  < current_floor - nearest_lower_req)
               car_opposite_up = true;
         //eger asagi gidiyorsa ve bizim ustumuzdeki katlardan talep varsa
         if( (odir==DN || odir==STOP) && requests_above )
            //talebin yukarisindaysa ve bizden daha yakinsa
            if(ofloor >= nearest_higher_req
               && ofloor - nearest_higher_req
```



```
                  < nearest_higher_req - current_floor)
               car_opposite_dn = true;
            } //if'in sonu (biz degiliz)
         } //for'un sonu (her kabin icin)

      //eger yukari gidiyorsak veya durmussak ve bizim ustumuzde bir FR varsa,
      //ve bizle FR arasında yukari yonde giden baska bir kabin yoksa,
      //veya FR'in yukarisindaysa ve asagiya gidiyorsa ve bizden daha yakinsa,
      //o zaman yukari git
      if( (current_dir==UP || current_dir==STOP)
            && requests_above && !car_between_up && !car_opposite_dn )
         {
         current_dir = UP;
         return;
         }

      //eger asagi gidiyorsak veya durmussak ve bizim altimizda bir FR varsa,
      //ve bizle FR arasinda asagi yonde giden baska bir kabin yoksa,
      //veya FR'in asagisindaysa ve yukari gidiyorsa ve bizden daha yakinsa,
      //o zaman asagi git
      if( (current_dir==DN || current_dir==STOP)
            && requests_below && !car_between_dn && !car_opposite_up )
         {
         current_dir = DN;
         return;
         }
      //bunun haricinde bir sey olmuyorsa, dur
      current_dir = STOP;
      } //decide()'in sonu, nihayet!
   //-----------------------------------------------------------------
   void elevator::move()
      {                                   //inen veya binen varsa,
      if(loading_timer || unloading_timer)    //hareket etme
         return;
      if(current_dir==UP)                 //yukari gidiyorsa, yukari git
         current_floor++;
      else if(current_dir==DN)            //asagi gidiyorsa, asagi git
         current_floor--;
      }//move()'un sonu
   //-----------------------------------------------------------------
   void elevator::get_destinations()      //dur, gidilecek katlari ogren
      {
      char ustring[BUF_LENGTH];           //girdi icin yardimci tampon
      int dest_floor;                     //gidilecek kat

      set_cursor_pos(1,22); clear_line();    //en ust satiri temizle
      set_cursor_pos(1,22);
      cout << "Car " << (car_number+1)
           << " has stopped at floor " << (current_floor+1)
           << "\nEnter destination floors (0 when finished)";
      for(int j=1; j<NUM_FLOORS; j++)        //kat taleplerini al
         {                                   //maksimum; genellikle daha az
         set_cursor_pos(1,24);
         cout << "Destination " << j << ": ";

         cin.get(ustring, BUF_LENGTH);       //(birden fazla linefeed'e engel ol)
         cin.ignore(10, '\n');               //newline karakteri dahil, karakterleri al
         dest_floor = atoi(ustring);
         set_cursor_pos(1,24); clear_line();  //eski girdi satirini temizle
```

```
      if(dest_floor==0)                    //daha fazla talep yoksa,
         {                                 //en alttaki uc satiri temizle
         set_cursor_pos(1,22); clear_line();
         set_cursor_pos(1,23); clear_line();
         set_cursor_pos(1,24); clear_line();
         return;
         }
      --dest_floor;                        //0'dan basla, 1'den degil
      if(dest_floor==current_floor)        //bu kat secilmis,
         { --j;continue; }                 //oyleyse bunu unut
      //eger durdurulduysak, ilk yapilan tercih, yonu ayarlar
      if(j==1 && current_dir==STOP)
         current_dir = (dest_floor < current_floor) ? DN : UP;
      destination[dest_floor] = true;      //secimi kaydet
      dests_display();                     //gidilecek katlari ekranda ciz
      }
   } //get_destinations()'in sonu
```

Uygulama

ELEV_APP.H ve **ELEV_APP.CPP** dosyaları, aklında belirli bir bina olan biri tarafından oluşturulur. Bu kişi, bu yazılımı kendi binası için uyarlamak ister. **ELEV_APP.H**, iki sabit tanımlayarak bu işi gerçekleştirir. Sabitler kat sayısını ve binada mevcut olacak asansör sayısını belirtirler. Dosya-nın listesi şöyledir:

```
// elev_app.h
// building sinifinin ozelliklerini belirtmek icin sabitleri sunar

const int NUM_FLOORS = 20;    //kat sayisi
const int NUM_CARS = 4;       //asansor kabini sayisi
```

ELEV_APP.CPP dosyası, **building** sınıfı içindeki verileri ilk kullanıma hazırlar; **new** operatö-rünü kullanarak birkaç tane **elevator** nesnesi oluşturur. (Bir dizi de kullanılabilirdi.) Sonra, bir döngü içinde **building** fonksiyonlarından **master_tick() ve get_floor_requests()**'i tekrar tekrar çağırır. **wait()** fonksiyonu (**MSOFTCON.H** ve **BORLACON.H** dosyalarında deklare edilir) işleri yavaşlatarak, insanların çalışma temposuna düşürür. Kullanıcı girdi girerken, zaman durur (kullanıcının zamanına karşı, programın saati durur). **ELEV_APP.CPP**'nin listesi şöyledir:

```
// elev_app.cpp
// musteri tarafindan saglanan dosya

#include "elev.h"       //sinif deklarasyonlari icin

int main()
   {
   building theBuilding;
   while(true)
      {
      theBuilding.master_tick();          //tum kabinlere saat sinyali gonder
      wait(1000);                         //bekle
                                          //kullanicidan kat taleplerini al
      theBuilding.record_floor_reqs();
      }
   return 0;
   }
```


Asansör Stratejisi

Asansör kabinlerine gerekli zekayı kurmak kolay bir değildir. Bu, bir seri kurallar içeren **decide()** fonksiyonunda ele alınır. Bu kurallar öncelik sırasına göre düzenlenir Eğer içlerinden biri uygulanabiliyorsa, ilgili faaliyet geçekleştirilir; diğer kurallar sorgulanmaz. Biraz basitleştirilmiş bir uygulamayı aşağıda listeledik:

1. Eğer asansör şaftın alt tarafına veya tavanına çarpmak üzereyse, dur.
2. Eğer bu, gidilecek katsa, yolcuları indir.
3. Eğer asansör bu kattayken yukarıdaki katlardan talep varsa ve biz yukarı gidiyorsak, yol-cuları bindir.
4. Eğer asansör bu kattayken aşağıdaki katlardan talep varsa ve biz aşağı gidiyorsak, yolcuları bindir.
5. Yukarıdaki veya aşağıdaki katlardan talep yoksa ve gidilecek kat belirtilmemişse, dur.
6. Yukarımızdaki katlarda inecek varsa, yukarı çık.
7. Aşağımızdaki katlarda inecek varsa, aşağı in.
8. Eğer duruyorsak veya yukarı çıkıyorsak ve yukarımızdaki katlardan talep varsa ve bizle talepte bulunan kat arasında yukarı çıkan başka kabin yoksa veya talepte bulunan katın üstünde, aşağı doğru inen ve o kata bizden daha yakın olan başka bir kabin yoksa, yu-karı çık.
9. Eğer duruyorsak veya aşağıya iniyorsak ve aşağımızdaki katlardan talep varsa ve bizle talepte bulunan kat arasında aşağı inen başka kabin yoksa veya talepte bulunan katın altında, yukarı doğru çıkan ve o kata bizden daha yakın olan başka bir kabin yoksa, aşağı in.
10. Diğer kurallar geçerli değilse, dur.

8. ve 9. kurallar oldukça karmaşıktır. Bu kurallar, iki veya daha fazla kabininin aynı kattan gelen talebe cevap vermek için koşturmasını önlemeye çalışırlar. Yine de, sonuçlar kusursuz değildir. Kimi durumlarda kabinler, talepleri karşılamakta yavaş kalırlar, çünkü başka bir kabi-nin bu talebi karşılamak üzere yolda olmasından çekinirler. Oysa, gerçekte diğer kabin farklı bir kattan gelen talebe cevap veriyordur. **decide()** fonksiyonu, mevcut kabinin yukarısından veya aşağısından kat talepleri olup olmadığını kontrol ederken, fonksiyonun yukarıdan ve aşağıdan gelen talepleri ayırt etmesine imkan vererek programın stratejisi geliştirilebilir. Ancak bu, zaten yeterince uzun olan **decide()** fonksiyonunu daha da karmaşıklaştırır. Bu tür incelik-leri size bırakıyoruz.

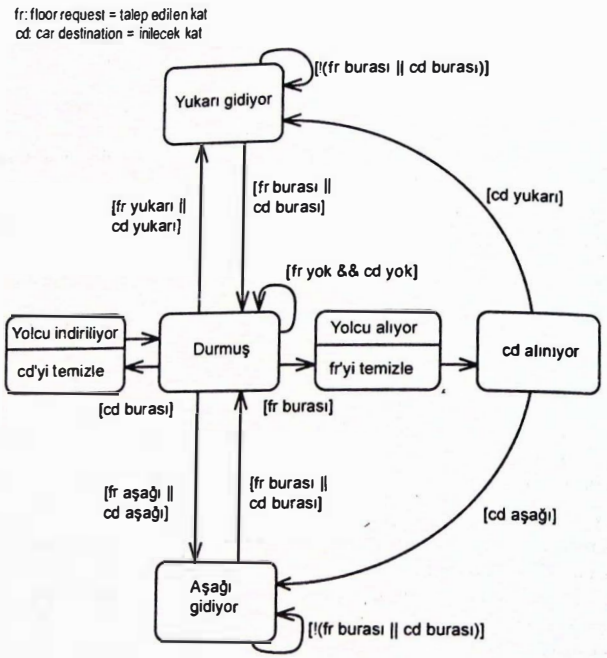
ELEV Programı İçin Durum Şeması

UML durum şemalarını "İşaretçiler" adlı Bölüm 10'da tanıtmıştık. Şimdi bir **elevator** nesne-sine göz atalım. İşleri biraz basitleştirmek için binada sadece bir kişi olduğunu ve bir asansö-rün kullanımda olduğunu varsayacağız. Yani, bir kerede yalnızca tek kattan talep gelebilir ve yolcu, inmek için yalnızca tek kat seçilebilir. Asansör kabini, diğer kabinlerin ne yaptıklarını dert etmek zorunda değildir. Bunun nasıl olduğu Şekil 13.6'da gösterilmiştir.

Şemada "cd", asansörün gideceği katı gösterir. Yani, kabinin içinde basılan düğmedir ve programda kabaca, **destination** dizisindeki bir değere karşılık gelir. Ayrıca "fr", kat talebini gösterir. Yani, kabinin dışından basılan düğmedir ve **floor_req** değişkenine karşılık gelir.

Durumlar, **current_dir** değişkeninin ve kabinin durumunu gösteren **loading_timer** ve **unloading_timer** değişkenlerinin değerlerinden türetilir. Tüm geçişler saat sinyallerinin so-

nucu olduğu için şemada sadece koruyucu koşullar gösterilir. Koruyucular, kat talepleri ve kabinin gideceği katlar hakkında kabinin sahip olduğu bilgiyi simgeler.



ŞEKİL 13.6: Bir `elevator` nesnesinin durum şeması.

Özet

Satıcı tarafından tedarik edilen nesne kütüphaneleri genellikle, bir `.H` başlık dosyasındaki sınıf deklarasyonlarını içeren bir public bileşen (arayüz) ve bir `.OBJ` veya `.LIB` kütüphane dosyasındaki üye fonksiyon tanımlarını içeren bir private bileşen (uygulama) şeklinde dağıtılır.

C++ derleyicileri, birkaç kaynak veya nesne dosyasını tek bir çalıştırabilir dosya içinde birleştirmeye imkan verir. Bu, nihai uygulamayı elde etmek için bir satıcıdan temin edilen dosyaların bir diğerinden temin edilen dosyalarla birleştirmesini mümkün kılar. Proje özelliği, derlenmesi gereken dosyaların takibini kolaylaştırır. Son bağlamadan bu yana değişikliğe uğramış dosyaları derler ve elde edilen nesne dosyalarını bağlar.

Dosyalar arası haberleşme, değişken, fonksiyon ve sınıf nesnelerinin tek bir dosyada tanımlanmasını ve kullanıldıkları diğer dosyalarda bildirilmelerini gerekli kılar. Bir sınıf tanımı, nesnelerin örneklendiği tüm dosyalara yerleştirilmelidir. Birden fazla tanımın ortaya çıkmamasını garanti etmek için hem kaynak dosyaları hem de başlı dosyaları dikkatle ele alınmalıdır.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Bir programı birkaç dosyaya bölmek istenilen bir davranıştır, çünkü
   a. bazı dosyalar her seferinde yeniden derlenmek zorunda kalmaz.
   b. bir program işlevsel olarak bölünebilir.
   c. dosyalar nesne formunda pazarlanabilir.
   d. farklı programcılar farklı dosyalar üzerinde çalışabilir.
2. _____ kullanılarak bir `.H` dosyası, bir `.CPP` dosyası ile ilişkilendirilir.
3. _____ kullanılarak bir `.OBJ` dosyası, bir `.CPP` dosyasına iliştirilebilir.
4. Bir *proje* dosyası aşağıdakilerden hangisini (hangilerini) içerir?
   a. Projedeki dosyaların içeriklerini.
   b. Projedeki dosyaların tarihlerini.
   c. Derleme ve bağlama komutlarını.
   d. C+ değişkenlerinin tanımlarını.
5. Ayrı bir ürün olarak tedarik edilen bir grup birbiriyle bağlantılı sınıf genellikle _____ olarak adlandırılır.
6. Doğru/Yanlış: Bir başlık dosyası, bir projedeki birden fazla kaynak dosyası tarafından erişilmek zorunda olabilir.
7. Bir sınıf kütüphanesinin özel dosyalar denilen dosyaları
   a. bir şifre gerektirir.
   b. `friend` fonksiyonlar tarafından erişilebilir.
   c. kodun korsan kullanımını önlemeye yardımcı olur.
   d. tek bir nesne kodunu içerebilir.
8. Doğru/Yanlış: Sınıf kütüphaneleri, fonksiyon kütüphanelerinden daha güçlü olabilir.
9. Doğru/Yanlış: Arayüz public; uygulama, private'tır.
10. Bir sınıf kütüphanesinin açık bölümü genellikle aşağıdakilerden hangisini (hangilerini) içerir?
    a. Üye fonksiyon deklarasyonlarını.
    b. Üye fonksiyon tanımlarını.
    c. Sınıf bildirimlerini.
    d. Yerel (inline) fonksiyonların tanımlarını.
11. İki veya daha fazla kaynak dosyası _____ birleştirilebilir.
12. Doğru/Yanlış: Bir fonksiyon gövdesi içinde tanımlanan bir değişken, tanımlı olduğu dosya boyunca erişilebilir.
13. `A` dosyası içinde bir global değişken tanımlanıyor. `B` dosyası içindeki bir değişkene erişmek için
    a. bu değişkeni `extern` anahtar kelimesini kullanarak `B` dosyası içinde tanımlamalısınız.
    b. bu değişkeni `static` anahtar kelimesini kullanarak `B` dosyası içinde tanımlamalısınız.
    c. başka hiçbir şey yapmanıza gerek yoktur.
    d. bu değişkeni `extern` anahtar kelimesini kullanarak `B` dosyası içinde deklare etmelisiniz.

14. Bir değişkenin programın diğer bölümlerindeki değişkenler tarafından erişilebildiği program alanına değişkenin _____ denir.
15. Gerçekte bağlayıcı (linker) tarafından birleştirilen dosyalara _____ denir.
16. `A` dosyası içinde bir fonksiyon tanımlanıyor. Bunu `B` dosyası içinden çağırmak için fonksiyon öncelikle _____ içinde _____.
17. Doğru/Yanlış: Bir fonksiyon deklarasyonu `extern` anahtar kelimesini gerekli kılmaz.
18. Farklı dosyalar içinde sınıf nesneleri tanımlamak için her dosya içinde
    a. sınıfı deklare etmelisiniz.
    b. sınıfı tanımlamalısınız.
    c. sınıfı `extern` kullanarak deklare etmelisiniz.
    d. sınıfı `extern` kullanarak tanımlamalısınız.
19. Doğru/Yanlış: Bir başlık dosyası içinde tanımlı bir değişkene, başlık dosyasını dahil etmiş olan iki kaynak dosyası tarafından erişilebilir.
20. Birden fazla tanımlamaya engel olmak amacıyla `#if !defined()...#endif` ifadesi ne zaman kullanılabilir?
    a. Bir kaynak dosyası içine iki tane başlık dosyası dahil edildiğinde.
    b. İki kaynak dosyası içine bir tane başlık dosyası dahil edildiğinde.
    c. İki kaynak dosyası içine iki tane başlık dosyası dahil edildiğinde.
    d. Bir başlık dosyası bir başka başlık dosyası içine dahil edilip, her ikisi de bir kaynak dosyası içine dahil edildiğinde.
21. İsim uzaylarını hangi sebeple kullanırız?
    a. Değişkenlerine isim verme işini otomatik hale getirmek için.
    b. Program öğelerinin erişilebilir olduğu alanı kısıtlamak için.
    c. Bir programı ayrı dosyalara bölmek için.
    d. Uzun değişken isimlerinin kullanımını önlemek için.
22. Bir isim uzayı tanımlamak için sınıf tanımına çok benzeyen bir format kullanırsınız; ancak, `class` anahtar kelimesinin yerine _____ anahtar kelimesini yerleştirirsiniz.
23. `typedef` kullanmak neye imkan verir?
    a. Uzun değişken isimlerini kısaltmaya.
    b. Bir tip isminin yerine başkasını kullanmaya.
    c. Uzun fonksiyon isimlerini kısaltmaya.
    d. Bir sınıf ismi yerine bir başkasını kullanmaya.

Projeler

Ne yazık ki, bu bölümde ele alınan büyüklükteki programları içeren alıştırmalara bu kitapta yer veremiyoruz. Yine de, kendi kendinize takip edebileceğiniz bazı proje önerilerini listeledik:

1. `VERYLONG` örneğindeki `verylong` sınıfı için çıkarma ve bölme işlemlerini gerçekleştiren üye fonksiyonlar tanımlayın. Bu fonksiyonlar – ve / operatörlerinin işlevlerini artırmalı. Uyarı: Burada yapılması gereken biraz iş var. Çıkarmayı dahil ettiğinizde, herhangi bir `verylong` sayının pozitif olabildiği gibi negatif de olabileceğini hesaba katmalısınız. Bu, sayıların işaretine göre farklı işlemler yapmak zorunda olan toplama ve çıkarma rutinlerini karmaşıklaştırır.

Bölme işleminde kullanılabilecek yöntemlerden birini anlamak amacıyla, elle uzun bir bölme alıştırması yapın ve her adımı not edin. Sonra bu adımları bir bölme üye fonksiyonuna dahil edin. Bazı karşılaştırmalara ihtiyaç duyacağınızı fark edeceksiniz. Bu nedenle, diğer rutinlerin yanı sıra bir de karşılaştırma rutini yazmanız gerekecektir.

2. `ELEV` programını tek bir asansörü ele alacak şekilde yeniden yazın. Bu işleri fazlasıyla kolaylaştıracaktır. Programın gereksiz olan bölümlerini çıkarın. Ya da durum şemasındaki gibi, tek bir asansör ile ayrıca tek bir yolcu olduğunu varsayabilirsiniz.
3. `ELEV` programını, talepleri daha etkili biçimde ele alacak şekilde değiştirin. Programın mevcut optimal olmayan bir davranışına örnek olarak, programı başlatın ve 20. katta aşağı yönde bir talepte bulunun. Sonra 10. katta aşağı yönde talepte bulunun. 1. kabin hemen 20. kata yönelecektir; fakat, 10. kata yönelmesi gereken 2. kabin harekete geçmeden önce 1. kabinin 10. kattan geçmesini bekleyecektir. Böyle bir durumun gerçekleşmemesi için `decide()` fonksiyonunu değiştirin.
4. İlginizi çeken bir şeyi modelleyen bir sınıf kütüphanesi oluşturun. Bunu test etmek için bir `main()` veya "istemci" (client) programı yazın. Sınıf kütüphanenizi pazarlayın; zengin ve ünlü olun.

ONDÖRDÜNCÜ BÖLÜM

14

ŞABLONLAR VE KURAL DIŞI DURUMLAR

Fonksiyon Şablonları

Sınıf Şablonları

Kural Dışı Durumlar (Exceptions)

Bu bölümde C++'ın iki ileri düzey özelliği olan şablonlar ve kural dışı durumlar tanıtılmıştır. Şablonlar, birden fazla veri tipini ele almak için bir fonksiyon veya sınıf kullanmayı mümkün kılar. Kural dışı durumlar, sınıflar içinde meydana gelen hataları kontrol altına almak için kullanışlı ve muntazam bir yöntem sağlarlar. Bu özelliklerin tek bir bölüm içinde birleştirilmesi, çoğunlukla tarihsel nedenlere dayanır: Bu özellikler aynı anda C++'ın parçası haline geldiler. C++'ın orijinal spesifikasyonunda yer almıyorlardı, fakat Ellis ve Stroustrup tarafından "Deneysel" konular olarak tanıtılmışlardı (1990). Ardından Standart C++'a dahil edildiler.

Şablon kavramı iki farklı şekilde kullanılabilir: Fonksiyonlarla ve sınıflarla. Öncelikle fonksiyon şablonlarına göz atacağız; sonra, sınıf şablonlarına geçeceğiz ve son olarak da kural dışı durumlara inceleyeceğiz.

Fonksiyon Şablonları

Diyelim ki, iki sayının mutlak değerlerini döndüren bir fonksiyon yazmak istiyorsunuz. Şüphesiz, ortaokul matematiğinden hatırlayacağınız gibi, bir sayının mutlak değeri, sayının işaretsiz değeridir: 3'ün mutlak değeri 3'tür ve –3'ün mutlak değeri de 3'tür.

Her zamanki gibi bu fonksiyon, belirli bir veri tipi için yazılacaktır:

```
int abs(int n)                  //tamsayilarin mutlak degerleri
   {
   return (n<0)? -n : n;        //eger n negatifse, -n dondur
   }
```

Buradaki fonksiyon, `int` tipinde bir değişken alıp, aynı tipte bir değer döndürmek için tanımlanır. Fakat diyelim ki, artık `long` tipinde bir sayının mutlak değerini bulmak istiyorsunuz. Bu durumda tamamen yeni bir fonksiyon yazmanız gerekecektir:

```
long abs(long n)                //long'larin mutlak degeri
   {
   return (n<0)? -n : n;
   }
```

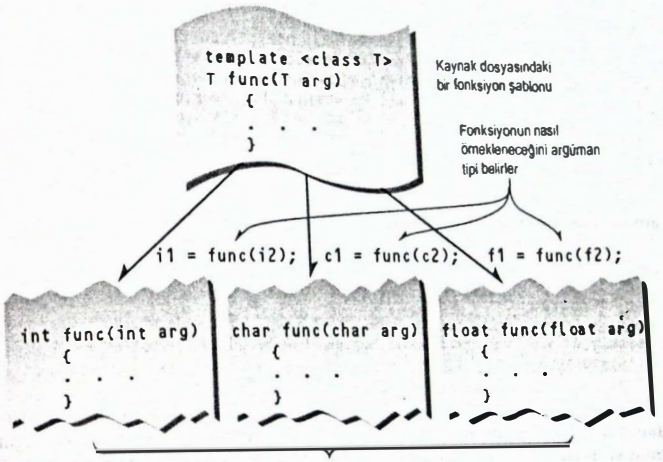
Ayrıca, `float`'lar için yine:

```
float abs(float n)              //float'larin mutlak degeri
   {
   return (n<0)? -n : n;
   }
```

Fonksiyonun gövdesi her durumda aynı şekilde yazılır, fakat bunlar tamamen farklı fonksiyonlardır, çünkü farklı tiplerdeki argümanları ve dönüş değerlerini ele alırlar. C++'ta bu fonksiyonların aynı isimde olacak şekilde aşırı yüklenmeleri mümkündür, ancak yine de, her biri için ayrı bir tanım yazmanız gerekir. (Operatörlerin aşırı yüklenmesini desteklemeyen C dilinde, farklı tipteki fonksiyonlar bile aynı isimde olamaz. Bu durum C fonksiyon kütüphanesinde benzer isimli fonksiyonlara yol açar. Söz gelişi, **abs()**, **fabs()**, **labs()** ve **cabs()**.)

Aynı fonksiyon gövdesini farklı tipler için tekrar tekrar yazmak zaman alıcıdır ve program listesinde yer kaybına neden olur. Ayrıca, eğer böyle bir fonksiyon içinde hata yaptığınızın farkına varırsanız, bunu, fonksiyon gövdelerinin her birinde düzeltmeniz gerektiğini hatırlamalısınız. Bunu tam olarak başaramamak programınıza tutarsızlıklar eklemenin iyi bir yoludur.

Bu tür bir fonksiyonu sadece bir kez yazmak ve farklı birçok veri tipinde fonksiyonun çalışmasını sağlamak bir şekilde mümkün olsa iyi olurdu. İşte, fonksiyon şablonlarının sizin için gerçekleştirdikleri kesinlikle bundan ibarettir. Bu fikir Şekil 14.1'de şematik olarak gösterilmiştir.



ŞEKİL 14.1: Bir fonksiyon şablonu.

Basit Bir Fonksiyon Şablonu

İlk örneğimiz, bir mutlak değer fonksiyonunu şablon olarak nasıl yazacağımızı gösteriyor. Böylece fonksiyon, temel nümerik tiplerinden herhangi biri ile çalışabilecektir. Bu program **abs()** için bir şablon versiyonu tanımlar, sonra fonksiyonun çalıştığını kanıtlamak için **main()**'de bu fonksiyonu değişik veri tipleriyle çağırır. **TEMPABS** programın listesi şöyledir:

```
// tempabs.cpp
// mutlak deger fonksiyonu icin sablon kullanilmasi
#include <iostream>
using namespace std;
//-------------------------------------------------------------
template <class T>                      //fonksiyon sablonu
T abs(T n)
   {
   return (n < 0) ? -n : n;
```

```
    }
//-------------------------------------------------------------
int main()
    {
    int int1 = 5;
    int int2 = -6;
    long lon1 = 70000L;
    long lon2 = -80000L;
    double dub1 = 9.95;
    double dub2 = -10.15;
                              //cagrilar, fonksiyonlarin kopyalarini olusturur
    cout << "\nabs(" << int1 << ")=" << abs(int1); //abs(int)
    cout << "\nabs(" << int2 << ")=" << abs(int2); //abs(int)
    cout << "\nabs(" << lon1 << ")=" << abs(lon1); //abs(long)
    cout << "\nabs(" << lon2 << ")=" << abs(lon2); //abs(long)
    cout << "\nabs(" << dub1 << ")=" << abs(dub1); //abs(double)
    cout << "\nabs(" << dub2 << ")=" << abs(dub2); //abs(double)
    cout << endl;
    return 0;
    }
```

Programın çıktısı şöyledir:

```
abs(5)=5
abs(-6)=6
abs(70000)=70000
abs(-80000)=80000
abs(9.95)=9.95
abs(-10.15)=10.15
```

Görüldüğü gibi, **abs()** fonksiyonu artık argüman olarak kullandığımız üç veri tipinin (**int**, **long** ve **double**) tümüyle çalışır. Diğer temel nümerik tiplerle de çalışır. Hatta küçüktür (<) operatörünün ve tekli eksi operatörünün (-) uygun biçimde aşırı yüklenmesi şartıyla kullanıcı tarafından tanımlanan tiplerle bile çalışır.

Birden fazla veri tipiyle çalışmak üzere **abs()** fonksiyonu şöyle belirtilir:

```
template <class T>      //fonksiyon sablonu
T abs(T n)
   {
   return (n<0) ? -n : n;
   }
```

İlk satırda **template** anahtar kelimesi ile başlayan ve peşinden gelen fonksiyon tanımını içeren söz diziminin tümü *fonksiyon şablonu* (*function template*) olarak adlandırılır. **abs()** fonksiyonunun yeni yazılış yöntemi nasıl bu denli şaşırtıcı bir esneklik verebiliyor?

Fonksiyon Şablonunun Söz Dizimi

Fonksiyon şablonlarındaki en temel yenilik, fonksiyon tarafından kullanılan veri tipini spesifik bir tip (mesela, **int**) olarak değil, fakat *herhangi* bir tipe karşılık gelebilen bir isim olarak simgelemektir. Önceki fonksiyon şablonunda bu isim **T** idi. (Bu ismin hiçbir enteresan yanı yok; istediğiniz herhangi bir şey, isim olabilir, **Type** veya **anyType** veya **FooBar** gibi.) **template** anahtar kelimesi, derleyiciye bir fonksiyon şablonu tanımlamak üzere olduğumuzu bildirir. Açılı parantezler içindeki **class** anahtar kelimesi de **type** olarak adlandırılabilir. Önceden

gördüğümüz gibi, sınıfları kullanarak kendi veri tiplerinizi tanımlayabilirsiniz; bu nedenle, tipler ve sınıflar arasında gerçekte ayırım yapılmaz. Bu örnekte, **class** anahtar kelimesini takip eden değişkene *şablon argümanı* (*template argument*) denir.

Şablonun tanımı boyunca, her zamanki gibi spesifik bir tip (mesela, **int**) yazılacak yerlere **T** argümanı yerleştirilir. **abs()** şablonunda bu isim, her ikisi de ilk satırda (fonksiyon deklarasyonunda), argüman tipi ve dönüş tipi olmak üzere sadece iki kez görünür. Daha karmaşık fonksiyonlarda ayrıca fonksiyon gövdesi içinde de pek çok defa görünebilir.

Derleyici Ne Yapar?

Derleyici, **template** anahtar kelimesini ve peşinden gelen fonksiyon tanımını görünce ne yapar? Aslında o an için hiçbir şey yapmaz. Fonksiyon şablonunun kendisi, derleyicinin herhangi bir kod üretmesine neden olmaz. Derleyici kod üretemez, çünkü henüz fonksiyonun ne tür bir veri tipi ile çalışacağını bilmez. Derleyici sadece ileride kullanmak için şablonu hatırında tutar.

Fonksiyon, program içindeki bir ifade tarafından çağrılana kadar kod üretimi gerçekleştirilmez. **TEMPABS** programında bu, aşağıdaki ifadede yer alan **abs(int1)** gibi deyimlerle gerçekleştirilir:

```
cout << "\nabs(" << int << ")=" << abs(int1);
```

Derleyici bu tür bir fonksiyon çağrısı gördüğünde kullanılacak tipin **int** olduğunu anlar, çünkü bu, **int1** argümanın tipidir. Böylece derleyici, fonksiyon şablonunda **T** ismini gördüğü her yerde T'nin yerine **int**'i yerleştirerek **abs()** fonksiyonunun **int** tipine özel bir versiyonunu üretir. Buna fonksiyon şablonunun *kopyasının oluşturulması* (*örneklenmesi*) (instantiating) denir, fonksiyonun örneklenen versiyonlarının her birine *şablon fonksiyonu* denir. (Yani, *şablon fonksiyonu*, *fonksiyon şablonunun* belirli bir kopyasıdır. Eğlenceli, değil mi?)

Derleyici ayrıca yeni kopyası oluşturulan fonksiyona da bir çağrı üretebilir. Bunu, kodda **abs(int1)**'in olduğu yere ekler. Aynı şekilde, **abs(lon1)** deyimi, **abs()** fonksiyonunun **long** tipi üzerinde işlem gören bir versiyonunun derleyici tarafından üretilmesine neden olur. **abs(dub1)** çağrısı ise **double** tipi üzerinde çalışan bir fonksiyon üretir. Elbette derleyici, her veri tipi için **abs()**'nin sadece bir versiyonunu üretecek kadar akıllıdır. Böylece, fonksiyonun **int** versiyonu için iki çağrı olsa bile, çalıştırılabilir (yürütülebilir) kod içinde bu versiyonun kodu yalnızca bir kez görünecektir.

Program Listesini Sadeleştirmek

Dikkat ederseniz, programın kullandığı RAM miktarı, şablon yöntemini kullansak da, gerçekten üç ayrı fonksiyon yazsak da aynıdır. Şablon yöntemi sadece, bizi kaynak dosyası içinde üç ayrı fonksiyon yazmaktan kurtarır. Bu, program listesini kısaltır ve listenin daha kolay anlaşılmasını sağlar. Ayrıca, eğer fonksiyonun çalışma şeklini değiştirmek istiyorsak, değişikliği liste üzerinde üç ayrı yerde yapmaktansa sadece bir yerde yapmamız gerekir.

Karar Argümanı

Derleyici fonksiyonu nasıl derleyeceğine, bütünüyle, fonksiyon çağrısında kullanılan argümanın (veya argümanların) veri tipine dayanarak karar verir. Fonksiyonun döndürdüğü değer bu kararda yer almaz. Bu, derleyicinin aşırı yüklenmiş birkaç fonksiyondan hangisini çağıracağına karar vermesine benzer.

Bir Başka Tür Taslak

Bir fonksiyon şablonunun gerçek anlamda bir fonksiyon olmadığını gördük, çünkü fonksiyon şablonu aslında program kodunun belleğe yerleştirilmesine neden olmaz. Bunun yerine fonksiyon şablonu, birçok fonksiyon oluşturmak için bir model veya bir taslaktır. Bu kavram, nesne yönelimli programlama felsefesine birebir uyar. Bu tıpkı, bir sınıfın (örneğin bellekteki bir program gibi) somut bir şey olmayıp çok sayıda benzer nesneler için bir taslak olması gibidir.

Birçok Argümanı Olan Fonksiyon Şablonları

Şimdi, bir fonksiyon şablonun bir başka örneğine göz atalım. Bu örnek, iki tanesi şablon argümanı, bir tane de temel veri tipi olmak üzere üç argüman alır. Bu fonksiyonun amacı, bir dizi içinde belirli bir değeri aramaktır. Fonksiyon değeri bulursa, bu değerin dizi indeksini; bulamazsa –1 döndürür. Fonksiyonun argümanları diziye işaret eden bir işaretçi, aranılacak değer ve dizinin büyüklüğüdür. **main()**'de farklı tiplerde dört farklı dizi ve aranılacak dört değer tanımlanır. Aranılan sayı için **char** tipini kullanılır. Sonra, her dizi için şablon şablon fonksiyonu bir kez çağrılır. **TEMPFIND**'ın program listesi şöyledir:

```
// tempfind.cpp
// dizi icindeki sayiyi bulan fonksiyon icin kullanilan sablon
#include <iostream>
using namespace std;
//-------------------------------------------------------------
//fonksiyon, sayinin indeks numarasini dondurur,eger bulamazsa –1 dondurur
template <class atype>
int find(atype* array, atype value, int size)
   {
   for(int j=0; j<size; j++)
      if(array[j]==value)
         return j;
   return -1;
   }
//-------------------------------------------------------------
char chrArr[] =      {1, 3, 5, 9, 11, 13}; //dizi
char ch = 5;                               //aranacak deger
int intArr[] =       {1, 3, 5, 9, 11, 13};
int in = 6;
long lonArr[] =      {1L, 3L, 5L, 9L, 11L, 13L};
long lo = 11L;
double dubArr[] =    {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;

int main()
   {
   cout << "\n 5 in chrArray: index=" << find(chrArr, ch, 6);
   cout << "\n 6 in intArray: index=" << find(intArr, in, 6);
   cout << "\n11 in lonArray: index=" << find(lonArr, lo, 6);
   cout << "\n 4 in dubArray: index=" << find(dubArr, db, 6);
      cout << endl;
      return 0;
   }
```

Bu programda şablon argümanı olan **atype** isimlendirilir. **atype**, fonksiyon argümanlarının ikisinde, diziye işaret eden işaretçi olarak ve aranılacak öğenin tipi olarak görünür. Fonksiyonun üçüncü argümanı olan dizi boyutu ise daima **int** tipindedir, yani şablon argümanı değildir. Programın çıktısı şöyledir:


```
 5 in chrArray: index=2
 6 in intArray: index=-1
11 in lonArray: index=4
 4 in dubArray: index=-1
```

Derleyici, fonksiyonu çağırmak için kullanılan her tip için bir tane olmak üzere fonksiyonun dört farklı versiyonunu üretir. Karakter dizisinde 2 numaralı indekste 5'i buluyor; tamsayı dizisinde 6'yı bulamıyor vs.

Şablon Argümanları Eşlenmelidir

Bir şablon fonksiyonu çağrıldığında, aynı şablon argümanının tüm kopyaları aynı tipte olmak zorundadır. Örneğin, **find()** fonksiyonunda eğer dizi ismi **int** tipinde ise aranılacak değer de **int** tipinde olmalıdır. Şöyle bir ifade kullanamazsınız:

```
int intarray[] = {1, 3, 5, 7};      //int dizisi
float f1 = 5.0;                     //float degeri
int value = find(intarray, f1, 4); //hata
```

Çünkü derleyici, **atype**'ın tüm kopyalarının aynı tipte olmasını bekler. Şöyle bir fonksiyon üretebilir:

```
find(int*, int, int);
```

Fakat şunu üretemez:

```
find(int*, float, int);
```

Çünkü, birinci ve ikinci argümanlar aynı tipte olmalıdır.

Sözdiziminde Değişiklik

Bazı programcılar **template** anahtar kelimesi ile fonksiyon deklaratörünü aynı satıra yerleştirirler:

```
template<class atype> int find(atype* array, atype value, int size)
   {
   //fonksiyon govdesi
   }
```

Elbette, derleyici bu biçimden yeterince mutludur; ancak biz bunu çok satırlı yaklaşımdan daha tehlikeli ve daha az anlaşılır buluyoruz.

Birden Fazla Şablon Argümanı

Bir fonksiyon şablonu içinde birden fazla şablon argümanı kullanabilirsiniz. Örneğin, **find()** fonksiyon şablonu fikrini beğendiniz diyelim; ama bunun hangi büyüklükte bir diziye uygulanabileceğinden emin olamıyorsunuz. Eğer dizi aşırı büyükse, dizi büyüklüğü için **long** tipini kullanmak gerekli olur; yoksa **int** tipi kullanılır. Öte yandan, eğer ihtiyacınız yoksa **long** tipini kullanmayı istemiyorsunuz. Fonksiyonu çağırdığınızda saklanılacak verilerin tipinin yanı sıra dizi boyutunun tipini de seçmek istiyorsunuz. Bunları mümkün kılmak için dizi büyüklüğünü de şablon argümanı haline getirebilirsiniz. Bunu **btype** olarak adlandıracağız:

```
template <class atype, class btype>
btype find(atype* array, atype value, btype size)
   {
   for(btype j=0; j<size; j++)      //btype kullanimina dikkat edin
      if(array[j]==value)
         return j;
   return static_cast<btype>(-1);
   }
```

Artık dizi boyutu için `int` tipi veya `long` tipinden uygun olanını (hatta kullanıcı tarafından tanımlanan tipleri bile) kullanabilirsiniz. Derleyici, sadece dizinin tipini ve aranılacak değeri değil aynı zamanda dizi büyüklüğünün tipini de baz alarak farklı fonksiyonlar üretecektir.

Dikkat ederseniz, çok sayıda şablon argümanı, tek bir şablondan kopyalanan birçok sayıda fonksiyona yol açabilir. Böyle iki tane argüman, her birinde bu argümanların makul olarak kullanılabileceği altı temel veri tipi varsa, 36 adet fonksiyonun oluşturulmasına imkan verecektir. Fonksiyonlar büyük olduğunda, bu çok fazla bellek harcar. Öte yandan, fonksiyonu gerçekten çağırmadığınız taktirde fonksiyonun bir versiyonunu oluşturmazsınız.

Makrolar, Neden Olmasın?

Eski zaman C programcıları, farklı veri tipleri için bir fonksiyonun farklı versiyonlarını oluşturmak amacıyla neden makro kullanmadığımızı merak edebilirler. Örneğin, `abs()` fonksiyonu şöyle tanımlanabilirdi:

```
#define abs(n) ( (n<0) ? (-n): (n) )
```

Bu, `TEMPABS` programındaki sınıf şablonu ile aynı etkiye sahiptir, çünkü bu ifade basit bir metin değişimi gerçekleştirir. Bu nedenle, herhangi bir tiple çalışabilir. Yine de, önceden belirttiğimiz gibi, makrolar C++'ta pek fazla kullanılmazlar. Makrolarla ilgili birçok sorun vardır. Bunlardan biri, makroların hiç tip kontrolü yapmamasıdır. Bir makronun aynı tipte olması gereken birkaç tane argümanı olabilir, fakat derleyici, argümanların aynı olup olmadığını kontrol etmez. Ayrıca, döndürülen değerin tipi de açıkça belirtilmez. Bu nedenle derleyici, bu değeri uyumsuz bir değişkene atayıp atmadığınızı anlayamaz. Her ne olursa olsun, makrolar tek bir ifade ile belirtilen fonksiyonlarla sınırlıdır. Makrolarla ilgili daha ince başka problemler de vardır. Genel olarak, en iyisi bunların kullanımını önlemektir.

Hangisi Çalışır?

Belirli bir veri tipinden bir şablon fonksiyon oluşturabileceğinizi nasıl anlarsınız? Söz gelişi, bir C karakter katarı (`char*`) dizisinde bir C karakter katarı bulmak için `find()` fonksiyonunu kullanır mıydınız? Bunun mümkün olup olmadığını görmek için fonksiyonda kullanılan operatörleri kontrol edin. Eğer bunların tümü aynı veri tipi üzerinde çalışıyorsa, muhtemelen bu fonksiyonu kullanabilirsiniz. Bununla birlikte, `find()` fonksiyonunda eşittir (`==`) operatörü kullanılarak iki değişken karşılaştırılır. Bu operatörü C karakter katarlarıyla kullanamazsınız; `strcmp()` kütüphane fonksiyonunu kullanmalısınız. Yani, `find()` C karakter katarları üzerinde çalışmaz. Buna rağmen `find()`, `string` sınıfı üzerinde kesinlikle çalışır, çünkü bu sınıfta `==` operatörü aşırı yüklenmiştir.

Normal Bir Fonksiyon İle Başlayın

Bir şablon fonksiyonu yazarken, muhtemelen en iyisi, sabit bir tip (mesela, `int`) üzerinde çalışan normal bir fonksiyon ile işe başlamanızdır. Böyle bir fonksiyonu, şablon sözdizimini ve birden fazla tipi dert etmeden tasarlayabilir ve hatalarını ayıklayabilirsiniz. Sonra, her şey düzgün olarak çalışınca, fonksiyon tanımını bir şablona dönüştürebilir ve bunun ilave tiplerle de çalıştığını kontrol edebilirsiniz.

Sınıf Şablonları

Şablon kavramı sınıflara da uyarlanabilir. Sınıf şablonları genellikle veri depolama (konteyner) sınıfları olarak kullanılır. (Bununla ilgili daha önemli bir örneği bir sonraki bölümde "Standart Şablon Kütüphanesi"nde göreceğiz.) Önceki bölümlerde karşılaştığımız yığınlar ve bağlı listeler, veri depolama sınıflarının birer örneğidir. Ancak, şimdiye kadar gösterdiğimiz bu tür sınıf örnekleri sadece bir tek temel tipteki verileri saklayabiliyordu. Örneğin, "Diziler ve Karakter Dizileri" adlı Bölüm 7'deki `STAKARAY` programında yer alan `Stack` sınıfı, yalnızca `int` tipinde verileri saklayabiliyordu. İşte `Stack` sınıfının özet bir versiyonu:

```
class Stack
   {
   private:
      int st[MAX];          //tamsayilar dizisi
      int top;              //yiginin en ustunun indeks numarasi
   public:
      Stack();              //kurucu fonksiyon
      void push(int var);   //arguman olarak int alir
      int pop();            //int degerini dondurur
   };
```

Eğer `long` tipinde verileri yığında saklamak isteseydik, tamamen yeni bir sınıf tanımlamamız gerekecekti:

```
class LongStack
   {
   private:
      long st[MAX];         //long'lar dizisi
      int top;              //yiginin en ustunun indeks numarasi
   public:
      LongStack();          //kurucu fonksiyon
      void push(long var); //arguman olarak long aliyor
      long pop();           //long degeri donduruyor
};
```

Aynı şekilde, saklamak istediğimiz her veri tipi için yeni bir sınıf tanımlamak zorunda kalacaktık. Bir tek temel tip yerine tüm tiplerdeki değişkenler için çalışacak tek bir sınıf spesifikasyonu yazabilmek hoş olurdu. Tahin edeceğiniz gibi, sınıf şablonları bunu gerçekleştirmemize imkan verir. `STAKARAY`'ın sınıf şablonu kullanan bir varyasyonunu tanımlayacağız. `TEMSTAK`'ın program listesi şöyledir:

```
// tempstak.cpp
// yigin sinifini sablon olarak uygular
#include <iostream.h>
using namespace std;
const int MAX = 100;            //dizinin buyuklugu
////////////////////////////////////////////////////////////////
template <class Type>
class Stack
```

```
    {
    private:
       Type st[MAX];          //yigin:herhangi tipte bir dizi
       int top;               //yiginin en ustunun numarasi
    public:
       Stack()                //kurucu fonksiyon
          { top = -1; }
       void push(Type var)    //sayiyi yigina koy
          { st[++top] = var; }
       Type pop()             //sayiyi yigindan al
          { return st[top--]; }
    };
//////////////////////////////////////////////////////////////
int main()
    {
    Stack<float> s1;          //s1, Stack<float> sinifinin bir nesnesidir

    s1.push(1111.1F);         //yigina 3 tane float koy, yigindan 3 tane float al
    s1.push(2222.2F);
    s1.push(3333.3F);
    cout << "1: " << s1.pop() << endl;
    cout << "2: " << s1.pop() << endl;
    cout << "3: " << s1.pop() << endl;

    Stack<long> s2;           //s2, Stack<long> sinifinin bir nesnesidir

    s2.push(123123123L);      //yigina 3 tane long koy, yigindan 3 tane long al
    s2.push(234234234L);
    s2.push(345345345L);
    cout << "1: " << s2.pop() << endl;
    cout << "2: " << s2.pop() << endl;
    cout << "3: " << s2.pop() << endl;
    return 0;
    }
```

Burada **Stack** sınıfı bir şablon sınıfı olarak sunulur. Buradaki yöntem, fonksiyon şablonlarında kullanılana benzer. **template** anahtar kelimesi ve **class Stack**, bütün sınıfın bir şablon olduğuna işaret eder.

```
template <class Type>
class Stack
   {
   //sablon argumani Type'i kullanan veri ve uye fonksiyonlar
   };
```

Sonra sınıf spesifikasyonu içinde **st** dizisinin tipine referans yapılan her yerde (**int** gibi sabit bir veri tipi yerine), bu örnekte **Type** olarak adlandırılan, bir şablon argümanı kullanılır. Bu tür üç yer vardır: **st**'nin tanımı, **push()** fonksiyonunun argüman tipi ve **pop()** fonksiyonunun dönüş tipi.

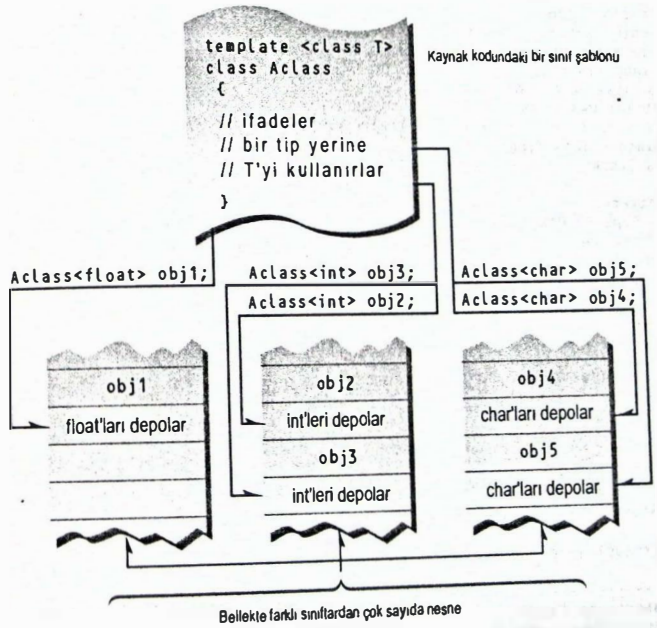
Sınıf şablonları, kopyalanma yöntemine göre fonksiyon şablonlarından ayrılırlar. Bir fonksiyon şablonundan gerçek bir fonksiyon oluşturmak için fonksiyonu belirli tipte argümanlar kullanarak çağırırsınız. Halbuki sınıflar, şablon argümanı kullanılarak bir nesne tanımlamak suretiyle kopyalanırlar.

```
Stack<float> s1;
```


Bu ifade, **float** tipinde sayıları saklayan **s1** adında bir nesne oluşturur. Derleyici, sınıf spesifikasyonu içinde **Type** şablon argümanının göründüğü her yerde **float** tipini kullanarak, bu nesnenin verileri için bellekte yer ayırır. Derleyici ayrıca, üye fonksiyonlar için de yer ayırır (eğer bu fonksiyonlar **Stack<float>** tipinde bir başka nesne tarafından zaten belleğe yerleştirilmemişlerse). Üstelik bu üye fonksiyonlar yalnızca **float** tipi üzerinde işlem görürler. Şekil 14.2'de bir sınıf şablonunun ve spesifik nesnelerin tanımlarının, nesneleri belleğe yerleştirmek için ne denli etkili olduğu gösteriliyor. Aşağıdaki gibi farklı tipte nesneleri saklayan bir **Stack** nesnesi tanımlamak, veriler için farklı bir alan oluşturmakla kalmaz, aynı zamanda **long** tipi üzerinde işlem yapan yeni bir üye fonksiyon kümesi de tanımlar:

```
Stack<long>s2;
```

s1'in tip isminin sınıf ismi, **Stack**, *ve şablon argümanından* oluştuğuna dikkat edin: **Stack<float>**. Bu, aynı şablondan üretilmesi mümkün olan diğer sınıflardan, mesela **Stack<int>** veya **Stack<long>**, **s1**'i ayırt eder.



ŞEKİL 14.2: Bir sınıf şablonu.

TEMPSTAK programında, **s1** ve **s2**'ye üç değer yerleştirerek ve değerleri tekrar geri alıp ekranda göstererek **s1** ve **s2** ile alıştırma yapıyoruz. Programın çıktısı şöyle görünür:

```
1: 3333.3       //float yigini
2: 2222.2
3: 1111.1
1: 345345345    //long yigini
2: 234234234
3: 123123123
```

Bu örnekte şablon yöntemi, bir sınıf karşılığında bize iki sınıf verir. Üstelik, diğer nümerik tiplerdeki sınıf nesnelerini yalnızca tek bir kod satırı ile kopyalayabiliriz.

Sınıf İsmi Bağlamsaldır

TEMPSTAK örneğinde sınıf şablonunun üye fonksiyonlarının tümü sınıf içinde tanımlanmıştı. Eğer üye fonksiyonlar haricen (sınıf spesifikasyonunun dışında) tanımlanırlarsa yeni bir sözdizimine ihtiyacımız olur. Sıradaki örnek bunun nasıl gerçekleştirildiğini gösterir. **TEMPSTAK2** için program listesi şöyledir:

```
// temstak2.cpp
// yigin sinifini sablon olarak uygular
// uye fonksiyonlar sinifin disinda tanimlanir
#include <iostream>
using namespace std;
const int MAX = 100;
////////////////////////////////////////////////////////////////
template <class Type>
class Stack
   {
   private:
      Type st[MAX];              //yigin:herhangi tipte bir dizi
      int top;                   //yiginin en ustunun numarasi
   public:
      Stack();                   //kurucu fonksiyon
      void push(Type var);       //sayiyi yigina koy
      Type pop();                //sayiyi yigindan al
   };
////////////////////////////////////////////////////////////////
template<class Type>
Stack<Type>::Stack()             //kurucu fonksiyon
   {
   top = -1;
   }
//-----------------------------------------------------------
template<class Type>
void Stack<Type>::push(Type var) //sayiyi yigina koy
   {
   st[++top] = var;
   }
//-----------------------------------------------------------
template<class Type>
Type Stack<Type>::pop()          //sayiyi yigindan al
   {
   return st[top--];
   }
```



```
//-----------------------------------------------------------
int main()
   {
   Stack<float> s1;
   s1.push(1111.1F);              //s1, Stack<float> sinifinin bir nesnesidir
   s1.push(2222.2F);              //yigina 3 tane float koy, yigindan 3 tane
   s1.push(3333.3F);              //float al
   cout << "1: " << s1.pop() << endl;
   cout << "2: " << s1.pop() << endl;
   cout << "3: " << s1.pop() << endl;

   Stack<long> s2;                //s2, Stack<long> sinifinin bir nesnesi

   s2.push(123123123L);           //yigina 3 tane long koy, yigindan 3 tane
   s2.push(234234234L);           //long al
   s2.push(345345345L);
   cout << "1: " << s2.pop() << endl;
   cout << "2: " << s2.pop() << endl;
   cout << "3: " << s2.pop() << endl;
   return 0;
   }
```

template<classType> deyimi, sınıf tanımının önünde yer alması gerektiği gibi, ayrıca haricen tanımlanmış üye fonksiyonların her birinin önünde de yer almalıdır. **push()** fonksiyonu işte şöyle görünür:

```
template<class Type>
void Stack<Type>::push(Type var)
   {
   st[++top] = var;
   }
```

Stack<Type> ismi, **push()** fonksiyonunun üyesi olduğu sınıfı belirtmek için kullanılır. Şablonsuz normal bir üye fonksiyonda **Stack** fonksiyon ismi kendi başına yeterli olurdu:

```
void Stack::push(int var)     //sablonsuz bir fonksiyon olarak Stack()
   {
   st[++top] = var;
   }
```

Fakat, fonksiyon şablonu için şablon argümanına da ihtiyacımız olur: **Stack<Type>**.

Böylece, şablon sınıfı isminin farklı bağlamlarda farklı şekilde ifade edildiğini görüyoruz. Şablon sınıfı ismi, sınıf spesifikasyonu içinde yalnızca kendi isminden ibarettir: **Stack**. Haricen tanımlı üye fonksiyonlar için şablon sınıf ismi, sınıf ismi ve şablon argümanından oluşur: **Stack<Type>**. Belirli bir veri tipini saklamak için gerçek nesneler tanımladığınızda ise şablon sınıfı ismi olarak sınıf ismi ve bu belirli tip kullanılır. Örneğin, **Stack<float>**.

```
class Stack                       //Stack sinif belirteci
{};
void Stack<Type>::push(Type var)  //push()tanimi
{}
Stack<float> s1;                  //Stack<float> tipinde nesne
```

Doğru ismi doğru bağlamda kullanmak için çok dikkat etmeniz gerekir. `Stack` kelimesine `<Type>` veya `<float>` deyimlerini eklemek kolaylıkla unutulur. Bunda hata yaparsanız, derleyici bundan hiç hoşlanmaz.

Bu örnekte gösterilmemiş olmasına rağmen, bir üye fonksiyon kendi sınıfına ait bir değeri döndürürken de ayrıca dikkatli olmalısınız. Diyelim ki, tamsayılar için güvenlik özellikleri sağlayan bir `Int` sınıfı tanımlıyoruz ("Operatörlerin Aşırı Yüklenmesi" başlığı altındaki Bölüm 8'de, 4. alıştırmada bu sınıftan bahsedilmişti). Bu sınıfa ait bir üye fonksiyon olan ve `Int` tipini döndüren `xfunc()`'ın harici tanımını kullanmışsanız, dönüş değeri için ve ayrıca kapsam çözünürlük operatörünün önünde `Int<Type>` deyimini kullanmanız gerekecektir:

```
Int<Type> Int<Type>::xfunc(Int arg)
   { }
```

Fonksiyon argümanının tipi olarak kullanılan bir sınıf ismi ise, öte yandan, `<Type>` deyimini içermek zorunda değildir.

Şablon Kullanan Bir Bağlı Liste Sınıfı

Şimdi, şablonların veri depolama sınıfı olarak kullanıldığı bir başka örneğe göz atalım. Bu, "İşaretçiler" adlı Bölüm 10'daki `LINKLIST` programımızın değiştirilmiş bir şeklidir. `LINKLIST` programını yeniden incelemenizi öneriyoruz. Bu programda hem `linklist` sınıfının kendisinin, hem de veri öğelerinin her birini gerçek anlamda saklayan `link` yapısının şablona dönüştürülmesi gerekir. `TEMPLIST`'in program listesi şöyledir:

```
// templist.cpp
// bagli listeyi sablon olarak uygular
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
template<class TYPE>                    //link<TYPE> yapisi
struct link                             //liste elemanlarindan biri
//bu yapi yanimi icinde 'link', link<TYPE> anlamina gelir
   {
   TYPE data;                           //veri ogesi
   link* next;                          //sonraki baglantiya isaret eden
isaretci
   };
////////////////////////////////////////////////////////////
template<class TYPE>                    //linklist<TYPE> sinifi
class linklist                          //baglantilar listesi
//bu sinif yapisi icinde 'linklist ', linklist<TYPE> anlamina gelir
   {
   private:
      link<TYPE>* first;                //ilk baglantiya isaret eden isaretci
   public:
      linklist()                        //argumansiz kurucu fonksiyon
         { first = NULL; }              //ilk baglanti yok
      //dikkat: yok edici fonksiyon iyi olurdu;
      //kodu basitlestirmek icin burada gosterilmemistir
      void additem(TYPE d);             //veri ogesini ekle (bir baglanti)
      void display();                   //baglantilarin tumunu goster
   };
////////////////////////////////////////////////////////////
template<class TYPE>
```



```
void linklist<TYPE>::additem(TYPE d)        //veri ogesini ekle
   {
   link<TYPE>* newlink = new link<TYPE>;    //yeni baglanti olustur
   newlink->data = d;                       //buna veri ata
   newlink->next = first;                   //bu, sonraki baglantiyi gostersin
   first = newlink;                         //simdi ilk baglanti bunu gostersin
   }
//-------------------------------------------------------------
template<class TYPE>
void linklist<TYPE>::display()
   {                                        //baglantilarin tumunu goster
   link<TYPE>* current = first;
   while( current != NULL )                 //isaretci ilk baglantiyi gostersin
      {                                     //son baglantiyla cik
      cout << endl << current->data;        //verileri yazdir
      current = current->next;              //sonraki baglantiya gec
      }
   }
//-------------------------------------------------------------
int main()
   {
   linklist<double> ld;             //ld, linklist<double> sinifinin nesnesidir

   ld.additem(151.5);                       //ld listesine 3 double ekle
   ld.additem(262.6);
   ld.additem(373.7);
   ld.display();                            //ld listesinin tamamini goster

   linklist<char> lch;              //lch,linklist<char> sinifinin nesnesidir

   lch.additem('a');                        //lch listesine 3 karakter ekle
   lch.additem('b');
   lch.additem('c');
   lch.display();                           //lch listesinin tamamini goster
   cout << endl;
   return 0;
   }
```

`main()`'de iki bağlı liste tanımlanır: Biri `double` tipinde sayıları tutmak için, diğeri ise `char` tipinde karakterleri tutmak için. Sonra, `additem()` üye fonksiyonu ile her listeye üç öğe yerleştirilerek ve `display()` üye fonksiyonu ile bütün öğeler görüntülenerek listeler çalıştırılır. `TEMPLIST`'in çıktısı şöyle olur:

```
373.7
262.6
151.5
c
b
a
```

Hem `linklist` sınıfı hem de `link` yapısı, herhangi bir tipe karşılık gelmesi için `TYPE` argümanından yararlanırlar. (Herhangi bir tip de değil aslında; gerçekten hangi tiplerin saklanabileceğini daha sonra ele alacağız.) Yani, sadece `linklist` değil, aynı zamanda `link` de bir şablon olmalı ve aşağıdaki satırı takip etmelidir:

```
template<class TYPE>
```

Dikkat ederseniz burada sadece bir sınıf şablona çevrilmekle kalmaz. Buradaki `link` yapısı gibi, değişken veri tipi kullanan tüm programlama öğelerinin şablona dönüşmesi gerekir. Daha önce olduğu gibi, sınıfın (ve bu programda yapının da) programın farklı bölümlerinde nasıl adlandırıldığına dikkat etmeliyiz. Bir sınıf ya da yapıyı kendi spesifikasyonu içinde yalnız başına kullanabiliriz: **`linklist`** ve **`link`** gibi. Harici üye fonksiyonlarda sınıf ya da yapı ismini ve şablon argümanını kullanmalıyız: **`linklist<TYPE>`**. **`linklist`** tipinden nesneleri tanımladığımızda ise, liste hangi spesifik veri tipini tutmak için kullanacaksa o tipi kullanmalıyız:

```
linklist<double> ld;        //linklist<double> sinifinin ld nesnesini tanimlar
```

Kullanıcı Tarafından Tanımlanan Veri Tiplerini Saklamak

Şu ana kadarki programlarımızda temel veri tiplerini saklamak için şablon sınıflarını kullandık. Örneğin **`TEMPLIST`** programında **`double`** ve `char` tipinden sayıları bir bağlı listede sakladık. Peki, kullanıcı tarafından tanımlanan tipleri (sınıfları) aynı şablon sınıflarında saklamak da mümkün müdür? Bu sorunun cevabı "Evet" ama dikkat etmemiz gereken bir nokta vardır.

Bağlı Liste Haline Personel Verileri

"Kalıtım" adlı Bölüm 9'daki **`EMPLOY`** programında kullandığımız **`employee`** sınıfını inceleyelim (türetilmiş sınıflar konusunda endişe etmeyin). **`employee`** tipinden nesneleri, **`TEMPLIST`** örneğinde gördüğümüz bağlı listede saklayabilir miyiz? Şablon fonksiyonlarda olduğu gibi, bir şablon sınıfın belli bir sınıfın nesneleri üzerinde işlem yapabilip yapamayacağını, şablon sınıfın bu nesneler üzerindeki işlemlerini kontrol ederek anlayabiliriz. **`linklist`** sınıfı, sakladığı nesneleri göstermek için aşırı yüklenmiş ekleme operatörünü (**`<<`**) kullanmaktadır.

```
void linklist<TYPE>::display()
   {
   . . .
   cout << endl << current->data;      //ekleme operatorunu (<<) kullanir
   . . .
   };
```

Bu durum, ekleme operatörünün tanımlı olduğu temel tipler için sorun teşkil etmez. Ancak, **`EMPLOY`** programındaki **`employee`** sınıfı bu operatörü aşırı yükleme yeteneğine sahip değildir. Bunu sağlamak için **`employee`** sınıfı üzerinde değişiklik yapmamız gerekecektir. Kullanıcıdan personel verilerini alma işlemini basitleştirmek için çıkarma operatörünü (**`>>`**) aşırı yüklemeniz gerekir. Bu operatörden gelen veri, bağlı listeye eklenmeden önce **`emptemp`** adlı geçici bir nesnede tutulur. **`TEMLIST2`**'nin kaynak kodu şöyledir:

```
// temlist2.cpp
// bagli listeyi sablon olarak uygular
// employee sinifi ile kullanilan listeyi gosterir

#include <iostream>
using namespace std;
const int LEN = 80;                    //isimlerin maksimum uzunlugu
```



```
////////////////////////////////////////////////////////////
class employee                                  //personel sinifi
   {
   private:
      char name[LEN];                           //personelin adi
      unsigned long number;                     //personel kodu
   public:
      friend istream& operator >> (istream& s, employee& e);
      friend ostream& operator << (ostream& s, employee& e);
   };
//-----------------------------------------------------------
istream& operator >> (istream& s, employee& e)
   {
   cout << "\n   Enter last name: "; cin >> e.name;
   cout << "   Enter number: "; cin >> e.number;
   return s;
   }
//-----------------------------------------------------------
ostream& operator << (ostream& s, employee& e)
   {
   cout << "\n   Name: " << e.name;
   cout <<"\n   Number: " << e.number;
   return s;
   }
////////////////////////////////////////////////////////////
template<class TYPE>                            //"link<TYPE>" yapisi
struct link                                     //listenin bir elemani
   {
   TYPE data;                                   //veri ogesi
   link* next;                                  //bir sonraki baglantiyi gosteren isaretci
   };
////////////////////////////////////////////////////////////
template<class TYPE>                            //"linklist<TYPE>" sinifi
class linklist                                  //baglantilar listesi
   {
   private:
      link<TYPE>* first;                        //ilk baglantiyi gosteren isaretci
   public:
      linklist()                                //argumansiz kurucu
         { first = NULL; }                      //ilk baglanti yok
      void additem(TYPE d);                     //veri ogesi ekle (bir baglanti)
      void display();                           //tum baglantilari goster
   };
//-----------------------------------------------------------
template<class TYPE>
void linklist<TYPE>::additem(TYPE d)            //veri ogesi ekle
   {
   link<TYPE>* newlink = new link<TYPE>;        //yeni baglanti kur
   newlink->data = d;                           //verisini ver
   newlink->next = first;                       //bir sonraki baglantiyi gosterir
   first = newlink;                             //ilk isaretci simdi bunu gosterir
   }
//-----------------------------------------------------------
template<class TYPE>
void linklist<TYPE>::display()                  //tum baglantilari goster
   {
   link<TYPE>* current = first;                 //isaretci ilk baglantiyi gostersin
   while( current != NULL )                     //son baglantida donguden cik
      {
```

```
        cout << endl << current->data;          //verileri goster
        current = current->next;                //bir sonraki baglantiya ilerle
        }
     }
////////////////////////////////////////////////////////////////
int main()
   {
   linklist<employee> lemp;                     //lemp, "linklist<employee>" sinifinin
   employee emptemp;                            //bir nesnesidir
   char ans;                                    //gecici personel kaydi
                                                //kullanicinin cevabi('y' ya da 'n')
   do
      {
      cin >> emptemp;                           //kullanicidan personel verisini al
      lemp.additem(emptemp);                    //bunu 'lemp' listesine ekle
      cout << "\nAdd another (y/n)? ";
      cin >> ans;
      } while(ans != 'n');                      //kullanici isini tamamladiginda
   lemp.display();                              //bagli listenin tamamini goster
   cout << endl;
   return 0;
   }
```

`main()`'de `lemp` adlı bir liste örneklenir. Daha sonra döngünün içinde, kullanıcıdan personel verisini girmesi istenir. Ardından, personel nesnesi listeye eklenir. Kullanıcı döngüyü durdurduğunda tüm personel verileri ekrana dökülür. Programın örnek bir etkileşimi şöyledir:

```
   Enter last name: Mendez
   Enter number: 1233
Add another(y/n)? y

   Enter last name: Smith
   Enter number: 2344
Add another(y/n)? y

   Enter last name: Chang
   Enter number: 3455
Add another(y/n)? n

   Name: Chang
   Number: 3455

   Name: Smith
   Number: 2344

   Name: Mendez
   Number: 1233
```

Dikkat ederseniz, `employee` tipinde nesneleri saklamak için `linklist` sınıfının hiçbir şekilde değiştirilmesi gerekmez. Şablon sınıflarının güzelliği işte burada: Şablon sınıfları sadece temel veri tipleriyle değil, aynı zamanda kullanıcı tarafından tanımlanan tiplerle de çalışırlar.

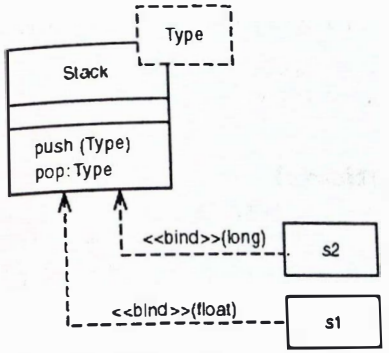
Neleri Saklayabilirsiniz?

Bir konuya dikkatinizi çekmiştik: Bir veri saklama şablon sınıfının üye fonksiyonlarındaki operatörleri kontrol ederek, belirli bir tipteki değişkenleri bu tür bir sınıf içinde saklayabilip

saklamayacağınızı anlayabilirsiniz. `TEMPLIST2` programındaki `linklist` sınıfı içinde bir karakter katarı (`string` sınıfı) saklamak mümkün müdür? Bu sınıf içindeki üye fonksiyonlar ekleme (<<) ve çıkarma (>>) operatörlerini kullanırlar. Bu operatörler, karakter katarlarıyla kusursuz çalışır. Bu nedenle, kendiniz de onaylayacağınız gibi, karakter katarlarını saklamak için bu sınıfı kullanmamak için hiçbir sebep yoktur. Ancak, depolama sınıfının üye fonksiyonunda, söz konusu veri tipi üzerinde işlem gören hiç operatör mevcut değilse bu tipi saklamak için bu sınıfı kullanamazsınız.

UML ve Şablonlar

Şablonlar (UML'de *parametreli sınıflar* olarak da adlandırılırlar), bir sınıf şeması içinde UML sınıf simgesinin bir varyasyonu olarak temsil edilirler. Şablon argümanlarının isimleri, sınıf dikdörtgeninin sağ üst köşesinden dikdörtgene giren bir noktalı dikdörtgen içine yerleştirilir. Şekil 14.3, bu bölümün başındaki `TEMPSTAK` programı için bir UML sınıf şemasını gösteriyor.



ŞEKİL 14.3: Bir UML sınıf şeması içindeki şablon.

Burada sadece tek bir şablon argümanı vardır: `Type`. `push()` ve `pop()` işlemleri, dönüş tipleri ve argüman tipleri ile gösterilir. (Dönüş tipinin fonksiyon isminin *peşinden* geldiğine ve noktalı virgül ile ayrıldığına dikkat edin.) Şablon argümanı genellikle işlem imzalarında kendini gösterir, `push()` ve `pop()`'ta `Type`'ın yaptığı gibi.

Bu şema ayrıca şablon sınıfından kopyalanan spesifik sınıfları da gösterir: `s1` ve `s2`.

Şablonları tarif etmesinin yanında Şekil 14.3, iki yeni UML kavramını da tanıtır: *bağımlılıklar* ve *klişeler*.

UML'de Bağımlılık

UML'de *bağımlılık* iki öğe arasındaki ilişkidir; öyle ki, bağımsız olanın içindeki bir değişiklik bağımlı olanın içinde de bir değişikliğe neden olabilir. Bağımlı olan, bağımsız olana bağlıdır veya bağımsız olanı kullanır. Bu nedenle, bağımlılık kimi zaman "*kullanıyor*" ilişkisi olarak da adlandırılır. Burada şablon sınıfı bağımsız öğedir; bu şablon sınıfından kopyalanan sınıflar ise bağımlı öğelerdir.

Bağımlılık, bağımsız öğeye işaret eden oklu kesikli çizgi ile gösterilir. Şekil 14.3'te örneklenen sınıflar **s1** ve **s2**, **Stack** sınıfına bağımlıdır; çünkü, eğer **Stack**'ın değişmesi gerekirse kopyalanan sınıflar da muhtemelen etkileneceklerdir.

Bağımlılık çok geniş bir kavramdır ve UML'de birçok duruma uygulanır. Önceden gördüğümüz bağlantı (association), genelleştirme ve diğer ilişkiler aslında bağımlılığın çeşitleridir. Yine de, bunlar UML şemalarında özel olarak gösterilecek kadar bir öneme sahiptirler.

En sık görülen bağımlılıklardan biri, bir sınıf bir başka sınıfı argüman olarak, işlemlerinin birinde kullandığı zaman ortaya çıkar.

UML'de Klişeler (Stereotypes)

Bir *klişe* (*stereotype*), bir UML öğesi hakkında ilave ayrıntılar belirtmenin yollarından biridir. Bir klişe, çift açılı parantez içindeki bir kelime ile simgelenir.

Söz gelişi, Şekil 14.3'teki kesikli çizgiler bağımlıkları simgeler, fakat bunların ne tür bir bağımlılık olduğunu belirtmez. **<<bind>>** klişesi ise bağımsız öğenin (şablon sınıfı), klişeyi takip eden parantezler içinde gösterilen parametreleri kullanarak bağımlı öğeyi (söz konusu sınıf) kopyaladığını açıkça belirtir. Yani, **Type**'ın **float** veya **long** ile değiştirileceğini dile getiriyor.

UML, dilin öğeleri olarak pek çok klişe tanımlar. Bunların her biri belirli bir UML öğesine karşılık gelir: kimileri sınıflara, kimileri bağımlılıklara vs. Siz de kendinize ait klişeler ekleyebilirsiniz.

Kural Dışı Durumlar (Exceptions)

Kural dışı durumlar, bu bölümün ikinci temel konusudur. Bunlar, C++ sınıfları tarafından üretilen hataları kontrol altına almak için sistematik, nesne yönelimli bir yöntem sunarlar. Kural dışı durumlar, programın çalışma zamanında (runtime) ortaya çıkan hatalardır. Bunlar çok çeşitli hata durumlarından kaynaklanır. Örneğin, belleği tüketmek, bir dosyayı açamamak, bir nesneye geçersiz bir başlangıç değeri atamaya çalışmak veya sınırların ötesine taşmış bir indeksi bir vektörde kullanmak, bu durumlardan bazılarıdır.

Kural Dışı Durumlara Neden İhtiyaç Duyarız?

Hataları kontrol altına almak için neden yeni bir mekanizmaya ihtiyaç duyarız? Şimdi, bu işlemin geçmişte nasıl ele alındığına bakalım. C dili programları genellikle, hatanın meydana geldiği fonksiyondan belirli bir değer döndürerek hata sinyali verirler. Örneğin, disk dosyası fonksiyonları bir hata sinyali vermek için genellikle **NULL** veya 0 döndürürler. Bu fonksiyonlardan birini her çağırışınızda döndürülen değeri kontrol edersiniz:

```
if( somefunc() == ERROR_RETURN_VALUE )
   //hataya el at veya hata isleme fonksiyonunu cagir
else
   //normal sekilde ilerle
if( anotherfunc() == NULL )
   //hataya el at veya hata isleme fonksiyonunu cagir
else
   //normal sekilde ilerle
if( thirdfunc() == 0 )
   //hataya el at veya hata isleme fonksiyonunu cagir
else
   //normal sekilde ilerle
```


Bu yöntemin bir problemi, bu tür bir fonksiyona yapılan çağrıların tümü program tarafından tek tek incelenmelidir. Fonksiyon çağrılarının her birini **if...else** ifadeleri ile çevrelemek ve hatayı kontrol altına alacak ifadeleri eklemek (veya hata yönetim rutinini çağırmak) çok fazla kod gerektirir. Ayrıca, listeyi dolambaçlı ve okunması güç hale sokar.

Sınıflar kullanıldığında problem daha da karmaşık hal alır; çünkü, bir fonksiyon açıkça çağrılmadan da hatalar ortaya çıkabilir. Örneğin, bir uygulamanın bir sınıfa ait nesneler tanımladığını varsayalım:

```
SomeClass obj1, obj2, obj3;
```

Uygulama, sınıfın kurucu fonksiyonunda bir hata olup olmadığını nasıl fark edebilir? Kurucu fonksiyon dahili olarak çağrılır; bu nedenle, kontrol edilecek bir dönüş değeri mevcut değildir.

Bir uygulama sınıf kütüphanelerini kullandığı zaman işler çok daha fazla karmaşıklaşır. Bir sınıf kütüphanesi ve bu sınıf kütüphanesinden yararlanan bir uygulama genellikle ayrı kişiler tarafından geliştirilmiştir: Sınıf kütüphanesi bir satıcı tarafından; uygulama ise, sınıf kütüphanesini satın alan bir programcı tarafından geliştirilmiştir. Bu durum, bir sınıfa ait üye fonksiyondan, bu fonksiyonu çağıran programa iletilecek hata değerlerini düzenlemeyi çok daha zor hale getirir. Bu konuya bölümün sonunda geri döneceğiz.

Eski zamandaki C programcıları, hataları yakalamanın bir başka yöntemini de hatırlayabilirler: **setjmp()** ve **longjmp()** fonksiyonlarının kombinasyonu. Buna rağmen, bu yöntem nesnelerin ortadan kaldırılışını düzgün olarak ele almadığı için nesne yönelimli bir ortam için uygun değildir.

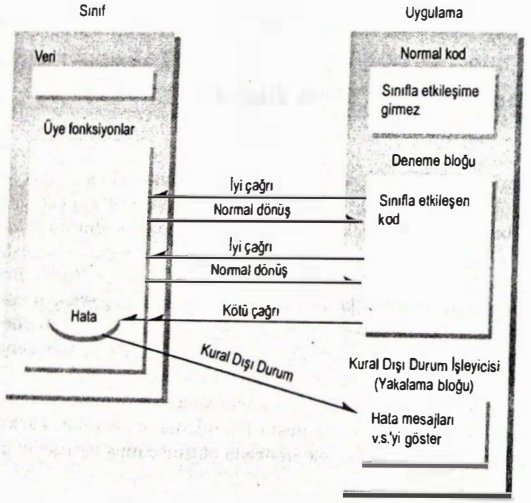
Kural Dışı Durumların Sözdizimi

Belirli bir sınıfın nesnelerini tanımlayan ve bu nesnelerle etkileşen bir uygulama hayal edin. Her zamanki gibi uygulamanın, sınıfın üye fonksiyonlarına çağrıda bulunması probleme neden olmaz. Yine de, kimi zaman uygulama, bir hatanın üye fonksiyon içinde tespit edilmesine neden olarak bir yanlışa yol açabilir. Bu üye fonksiyon sonra, bir hata ortaya çıktığını uygulamaya haber verir. Kural dışı bir durum kullanıldığında buna kural dışı bir durum *fırlatmak* (*throw*) deniyor. Hatayı kontrol altına almak için uygulamaya ayrı bir kod parçası yerleştirilir. Bu koda *kural dışı durum yöneticisi* (*exception handler*) veya *yakalama bloğu* (*catch block*) denir. Bu kod, üye fonksiyon tarafından fırlatılan kural dışı durumları *yakalar*. Uygulama içinde, sınıfın nesnelerini kullanan her kod bir *deneme bloğu* (*try block*) içine alınır. Deneme bloğu içinde üretilen hatalar yakalama bloğu içinde yakalanır. Sınıf ile etkileşim içinde olmayan kod, deneme bloğu içinde olmasa da olur. Bu düzenleme Şekil 14.4'te gösterilmiştir.

Kural dışı durum mekanizması üç yeni C++ anahtar kelimesi kullanır: **throw**, **catch** ve **try**. Ayrıca, **exception class** adında yeni bir tür unsur tanımlamamız gerekir. **XSYNTAX** çalışan bir program değil, sadece sözdizimini göstermek amacıyla oluşturulmuş bir iskelet programdır.

```
// xsyntax.cpp
// calisan bir program degildir
////////////////////////////////////////////////////////////
class AClass                          //bir sinif
   {
   public:
   class AnError                      //kural disi durum sinifi
      {
      };
```

```
    void Func()                         //bir uye fonksiyon
       {
       if( /*hata kosulu */ )
          throw AnError();              //kural disi durum firlat
       }
    };
//////////////////////////////////////////////////////////////
int main()                              //uygulama
   {
   try                                  //deneme blogu
      {
      AClass obj1;                      //AClass nesneleri ile etkilesim
      obj1.Func();                      //hataya neden olabilir
      }
   catch(AClass::AnError)               //kural disi durum yoneticisi
      {                                 //(yakalama blogu)
      //kullaniciya hata hakkinda bilgi ver vs.
      }
   return 0;
   }
```



ŞEKİL 14.4: Kural dışı durum mekanizması.

Hataların ortaya çıkabileceği herhangi bir sınıfı simgeleyen **AClass** adında bir sınıf ile başlanır. **AClass** sınıfının public bölümünde **AnError** adında bir kural dışı durum sınıfı tanımlanır. **AClass**'ın üye fonksiyonlarında hatalar kontrol edilir. Eğer bir hataya rastlanırsa, **throw** anahtar

kelimesi ve bunun peşinden hata sınıfının kurucu fonksiyonu kullanılarak bir kural dışı durum fırlatılır:

```
throw AnError();  //pesinden AnError sinifina ait bir kurucu fonksiyon
                  //gelen 'throw'
```

Program **main()** bölümünde, **AClass** ile etkileşim içinde olan ifadelerin tümü bir deneme bloğu içine alınır. Eğer bu ifadelerden herhangi biri, bir **AClass** üye fonksiyonu içinde hataya neden olursa, bir kural dışı durum fırlatılır ve kontrol, deneme bloğunun hemen peşi sıra gelen yakalama bloğuna geçer.

Basit Bir Kural Dışı Durum Örneği

Şimdi, kural dışı durumları kullanan çalışan bir programa bakalım. Bu örnek Bölüm 7'deki **STAKARAY** programından türetilmiştir. **STAKARAY**, tamsayı veri değerlerinin saklanabileceği bir yığın veri yapısı tanımlıyordu. Ne yazık ki, bu önceki örnek, sıkça ortaya çıkan iki hatayı tespit edemiyordu. Uygulama programı yığına haddinden fazla nesne yüklemeye kalkabiliyordu; böylece, dizinin kapasitesi aşılmış oluyordu. Ya da, program yığından haddinden fazla nesne almaya çalışıyordu; böylece, geçersiz veriler elde ediliyordu. **XSTAK** programında bu iki hatayı kontrol altına almak için bir kural dışı durum kullanılır.

```
// xstak.cpp
// kural disi durumlar
#include <iostream>
using namespace std;
const int MAX = 3;                  //yigin, 3 tamsayi tutar
//////////////////////////////////////////////////////////////
class Stack
   {
   private:
      int st[MAX];                  //tamsayilar dizisi
      int top;                      //yiginin en ustunun indeksi
   public:
      class Range                   //Stack icin kural disi durum sinifi
         {                          //dikkat: bos sinif govdesi
         };

      Stack()                       //kurucu fonksiyon
         { top = -1; }

      void push(int var)
         {
         if(top >= MAX-1)           //eger yigin doluysa,
            throw Range();          //kural disi durum olustur
         st[++top] = var;           //sayiyi yigina koy
         }
      int pop()
         {
         if(top < 0)                //eger yigin bossa,
            throw Range();          //kural disi durum olustur
         return st[top--];          //sayiyi yigindan al
         }
   };
//////////////////////////////////////////////////////////////
```

```
int main()
   {
   Stack s1;

   try
      {
      s1.push(11);
      s1.push(22);
      s1.push(33);
//    s1.push(44);                  //hooppp: yigin dolu
      cout << "1: " << s1.pop() << endl;
      cout << "2: " << s1.pop() << endl;
      cout << "3: " << s1.pop() << endl;
      cout << "4: " << s1.pop() << endl;   //hata: yigin bos
      }
   catch(Stack::Range)                    //kural disi durum isleyicisi
      {
      cout << "Exception: Stack Full or Empty " << endl;
      }
   cout << "Arrive here after catch (or normal exit)" << endl;
   return 0;
   }
```

Dikkat ederseniz, yığın küçük tutulmuştur, böylece yığına çok fazla öğe yerleştirerek bir kural dışı durumu tetiklemek daha kolay olur.

Şimdi, bu programın kural dışı durumlarla ilgili özelliklerini inceleyelim. Programda bunlardan dört tane mevcuttur. Sınıf spesifikasyonu içinde bir kural dışı durum sınıfı yer alır. Ayrıca, programda kural dışı durum fırlatan ifadeler de vardır. Programın **main()** bölümünde, kural dışı durumlara neden olabilecek bir kod bloğu (deneme bloğu) ve kural dışı durumları kontrol altına alan bir başka kod bloğu (yakalama bloğu) mevcuttur.

Kural Dışı Durum Sınıfını Açıkça Belirtmek

Program öncelikle, **Stack** sınıfı içinde bir kural dışı durum sınıfını açıkça belirtir:

```
class Range
   {           //dikkat: bos sinif govdesi
   };
```

Burada, fonksiyonun gövdesi boştur. Böylece, bu sınıfın nesnelerinin verileri ve üye fonksiyonları yoktur. Bu basit örnekte tüm ihtiyacımız olan bir sınıf ismi: **Range**. Bu isim, **throw** ifadesini yakalama bloğu ile birleştirir. (Daha sonra göreceğimiz gibi, sınıf gövdesinin her zaman boş olması gerekmez.)

Bir Kural Dışı Durum Fırlatmak

Stack sınıfında, yığın boşken uygulama yığından bir değer almaya çalışıyorsa veya yığın doluyken yığına bir değer yüklemeye çalışıyorsa bir kural dışı durum meydana gelir. Uygulamanın bir **Stack** nesnesini idare ederken bu tür bir yanlış yaptığını kendisine haber vermek için **Stack** sınıfının üye fonksiyonu, **if** ifadeleriyle bu koşulları kontrol eder ve eğer bir hata söz konusuysa bir kural dışı durum fırlatılır. **XSTAK** içinde kural dışı durum iki yerde fırlatılır; her ikisinde de aşağıdaki ifade kullanılır:

```
throw Range();
```


Bu ifadenin **Range()** bölümü **Range** sınıfının kurucu fonksiyonunu çağırır. Bu fonksiyon **Range** sınıfına ait bir nesne oluşturur. İfadenin **throw** bölümü, programın kontrolünü kural dışı durum yöneticisine aktarır (bunu az sonra inceleyeceğiz).

Deneme Bloğu

Bu kural dışı duruma neden olabilecek **main()**'deki tüm ifadeler – yani, **Stack** nesneleri üzerinde değişiklik yapan ifadeler – küme parantezi içine alınır ve **try** anahtar kelimesinin peşinde yer alır:

```
try
   {
   //kural disi duruma neden olabilecek nesneler uzerinde islem yapan kod
   }
```

Bu sadece, normal uygulama kodunun bir parçasından ibarettir. Kural dışı durumları kullanmıyor olsaydınız, yazmanız gereken kod bu olacaktı. Bu programdaki kodun tümünün deneme bloğu içinde bulunması gerekmez; sadece **Stack** sınıfı ile etkileşen kodun deneme bloğu içinde olması yeterlidir. Ayrıca, programınız içinde birçok deneme bloğu olabilir; böylece, **Stack** nesnelerine farklı yerlerden erişebilirsiniz.

Kural Dışı Durum Yöneticisi (Yakalama Bloğu)

Kural dışı durumu kontrol altına alan kod, kural dışı sınıf ismi parantez içinde olacak şekilde küme parantezleri içine alınır ve **catch** anahtar kelimesinden önce gelir. Kural dışı sınıf isminin, kendisinin içinde yer aldığı sınıfı içermesi gerekir. Bu, örnekte şu şekildedir: **Stack::Range**.

```
catch(Stack::Range)
   {
   //kural disi durumu yoneten kod
   }
```

Bu ifadeye *kural dışı durum yöneticisi* (*exception handler*) denir. Bunun deneme bloğunun hemen peşinden gelmesi gerekir. **XSTAK**'ta kural dışı durum yöneticisi, programın neden başarısız olduğunu kullanıcıya açıklamak için ekrana bir hata mesajı basar.

Kontrol, kural dışı durum yöneticisinin en sonuna kayar. Bu nedenle, işlemlere bu noktadan itibaren devam edebilirsiniz. Ya da, kural dışı durum yöneticisi, kontrolü başka yere aktarabilir veya (genellikle) programı sona erdirir.

Olaylar Zinciri

Şimdi, bir kural dışı durum meydana geldiğinde karşılaşılan olaylar zincirini özetleyelim:

1. Kod, deneme bloğu dışında normal biçimde çalışıyor.
2. Kontrol, deneme bloğuna geçiyor.
3. Deneme bloğu içindeki bir ifade, bir üye fonksiyon içinde hataya yol açıyor.
4. Üye fonksiyon bir kural dışı durum fırlatıyor.
5. Kontrol, deneme bloğunu takip eden kural dışı durum yöneticisine (yakalama bloğu) aktarılır.

Hepsi bu kadar. Sonuçta elde edilen kodun ne kadar net olduğuna dikkat edin. Deneme bloğu içindeki herhangi bir ifade bir kural dışı duruma neden olabilir. Fakat, her birinin döndürdüğü değeri kontrol etmek için kaygılanmamıza gerek yoktur, çünkü `try-throw-catch` düzenlemesi bunların tümünü otomatik olarak kontrol altına alır. Söz konusu örnekte, kural dışı duruma neden olan iki ifadeyi kasten oluşturduk. İlki,

```
s1.push(44);        //yigina haddinden fazla veri yukler
```

öncesinde gelen açıklama simgesini kaldırırsanız kural dışı duruma yol açar. İkincisi ise,

```
cout << "4: " << s1.pop() << endl;          //bos yigindan veri alir
```

ilk ifade açıklama olarak programdan çıkarılırsa bir kural dışı duruma neden oluyor. Her ikisini de deneyin. Her iki durumda da ekranda aynı hata mesajı gösterilecektir:

```
Stack Full or Empty
```

Birden Fazla Kural Dışı Durum

Bir sınıf tasarlayarak istediğiniz kadar çok sayıda kural dışı durum oluşturabilirsiniz. Bunun nasıl gerçekleştirildiğini göstermek için, dolu bir yığına veri yüklemeye çalışırken ve boş bir yığından veri almaya çalışırken ayrı ayrı kural dışı durumlar oluşturmak amacıyla **XSTAK** programını değiştireceğiz. **XSTAK2** programının listesi şöyledir:

```
// xstak2.cpp
// iki adet kural disi durum isleyicisi
#include <iostream>
using namespace std;
const int MAX = 3;                  //yigin 3 tamsayi tutar
////////////////////////////////////////////////////////////////
class Stack
   {
   private:
      int st[MAX];                  //yigin: tamsayi dizisi
      int top;                      //yiginin en ustunun indeksi
   public:
      class Full { };               //kural disi durum sinifi
      class Empty { };              //kural disi durum sinifi
//--------------------------------------------------------------
      Stack()                       //kurucu fonksiyon
         { top = -1; }
//--------------------------------------------------------------
      void push(int var)            //sayiyi yigina koy
         {
         if(top >= MAX-1)           //eger yigin doluysa,
            throw Full();           //Full adinda bir kural disi durum firlat
         st[++top] = var;
         }
//--------------------------------------------------------------
      int pop()                     //sayiyi yigindan al
         {
         if(top < 0)                //eger yigin bossa,
            throw Empty();          //Empty adinda bir kural disi durum firlat
         return st[top--];
```



```
         }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   Stack s1;

   try
      {
      s1.push(11);
      s1.push(22);
      s1.push(33);
//    s1.push(44);                  //hata: yigin dolu
      cout << "1: " << s1.pop() << endl;
      cout << "2: " << s1.pop() << endl;
      cout << "3: " << s1.pop() << endl;
      cout << "4: " << s1.pop() << endl;          //hata: yigin bos
      }
   catch(Stack::Full)
      {
      cout << "Exception: Stack Full" << endl;
      }
   catch(Stack::Empty)
      {
      cout << "Exception: Stack Empty" << endl;
      }
   return 0;
   }
```

XSTAK2'de iki adet kural dışı durum sınıfı belirtilir:

```
class Full { };
class Empty { };
```

Yığın zaten doluyken uygulama **push()** fonksiyonunu çağırırsa, aşağıdaki ifade çalıştırılır:

```
throw Full();
```

Şu ifade ise, yığın boşken **pop()** fonksiyonu çağrıldığında çalıştırılır:

```
throw Empty();
```

Her kural dışı durum için ayrı yakalama bloğu kullanılır:

```
try
   {
   //Stack nesneleri uzerinde islem yapan kod
   }
catch(Stack::Full)
   {
   //yiginin dolu olmasi durumunda kural disi durumu yoneten kod
   }
catch(Stack::Empty)
   {
   //yiginin bos olmasi durumunda kural disi durumu yoneten kod
   }
```

Belirli bir deneme bloğu ile kullanılan yakalama bloklarının tümü deneme bloğunu hemen peşi sıra takip etmelidir. Bu örnekte, yakalama bloklarının her biri yalnızca ekrana bir mesaj basarlar: "Stack Full" veya "Stack Empty". Belirli bir kural dışı durum için yalnızca bir yakalama bloğu çalıştırılır. Bir grup yakalama bloğu, veya *yakalama merdiveni*, kodun yalnızca ilgili bölümünü çalıştırarak sanki bir **switch** ifadesi gibi işlev görür. Bir kural dışı durum yönetilirken kontrol, tüm yakalama bloklarının peşinden gelen ifadeye aktarılır. (Bir **switch** ifadesinden farklı olarak yakalama bloklarının her birini **break** ile bitirmenize gerek yoktur. Bu bağlamda, yakalama blokları daha çok fonksiyonlar gibi davranırlar.)

Distance Sınıfı ve Kural Dışı Durumlar

Şimdi, kural dışı durumlarla ilgili bir başka örneğe göz atalım. Bu kez, önceki bölümlerden bildiğiniz ünlü **Distance** sınıfı üzerinde bir uygulama yapalım. Bir **Distance** nesnesi, ayak için bir tamsayı değerine ve inç için de bir kayan noktalı değere sahiptir. İnç değeri her zaman 12.0'den küçük olmalıdır. Önceki örneklerde bu sınıfla ilgili saptadığımız problem şuydu: Kullanıcı bir nesneye başlangıçta inç değeri olarak 12.0 veya 12.0'den büyük bir değer atanmışsa, sınıf kendisini bu durumdan koruyamıyordu. Aritmetik rutinleri (mesela, **operator+()**) inç'in 12.0'den küçük olacağını varsaydıkları için **Distance** sınıfı aritmetik işlem gerçekleştirmeye çalıştığında bu durum probleme yol açabiliyordu. Bu tür geçersiz değerler de ekranda gösterilebilir. Böylece, 7'-15" gibi değerler karşısında kullanıcının kafası allak bullak olabilir.

Şimdi, bu tür bir hatayı kontrol altına almak amacıyla bir kural dışı durum kullanarak, **XDIST**'te gösterildiği gibi, **Distance** sınıfını yeniden yazalım.

```
// xdist.cpp
// Distance sinifi ile kural disi durumlar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance                          //English Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      class InchesEx { };               //kural disi durum sinifi
//-----------------------------------------------------------
      Distance()                        //kurucu fonksiyon (arguman yok)
         { feet = 0; inches = 0.0; }
//-----------------------------------------------------------
      Distance(int ft, float in)        //kurucu fonksiyon (iki argumanli)
         {
         if(in >= 12.0)                 //inches limit disinda ise,
            throw InchesEx();           //kural disi durum firlat
         feet = ft;
         inches = in;
         }
//-----------------------------------------------------------
      void getdist()                    //mesafeyi kullanicidan al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         if(inches >= 12.0)             //inches limit disinda ise,
            throw InchesEx();           //kural disi durum olustur
```



```
         }
//-----------------------------------------------------------
      void showdist()                   //mesafeyi goster
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////
int main()
   {
   try
      {
      Distance dist1(17, 3.5);          //2 argumanli kurucu
      Distance dist2;                   //argumansiz kuruc
      dist2.getdist();                  //mesafeyi kullanicidan al
      cout << "\ndist1 = "; dist1.showdist();   //mesafeleri goster
      cout << "\ndist2 = "; dist2.showdist();
      }
   catch(Distance::InchesEx)            //kural disi durumlari yakala
      {
      cout << "\nInitialization error: "
           "inches value is too large.";
      }
   cout << endl;
   return 0;
   }
```

Distance sınıfına **InchesEx** olarak adlandırılan bir kural dışı durum sınıfı yerleştirilir. Sonra, kullanıcı inç verisine ilk değer olarak 12.0'ye eşit veya 12.0'den büyük bir değer atamaya çalışırsa, bir kural dışı durum fırlatılır. Bu iki yerde ortaya çıkar: Programcının başlangıç değerlerini temin ederken hata yapabileceği iki argümanlı kurucu fonksiyonda ve kullanıcının *Enter inches* yönlendirmesiyle hatalı bir değer girebileceği **getdist()** fonksiyonunda. Ayrıca, negatif değerleri ve diğer girdi hatalarını da kontrol edebiliriz.

main()'de **Distance** nesneleriyle gerçekleştirilen etkileşimin tümü bir deneme bloğu içine alınır; yakalama bloğu ise ekranda bir hata mesajı verir.

Elbette daha sofistike programlarda, bir kullanıcı hatasını (programcı hatasına karşın) daha farklı kontrol altına almak isteyebilirsiniz. Deneme bloğunun başına gidip kullanıcıya bir başka uzaklık değeri girmesi için bir şans daha vermek kullanıcı açısından çok daha tercih edilir olur.

Argümanlı Kural Dışı Durumlar

Uygulama, kural dışı duruma neyin neden olduğu konusunda daha ayrıntılı bilgiye ihtiyaç duyarsa ne olur? Örneğin, **XDIST** programında, hatalı inç değerinin ne olduğunu bilmek programcıya yardımcı olabilir. Aynı kural dışı durum **XDIST**'te olduğu gibi farklı üye fonksiyonlar tarafından fırlatılabiliyorsa, hatanın gerçek faili olan fonksiyonu bilmek fayda sağlar. Peki bu tür bilgileri, kural dışı durumun fırlatıldığı üye fonksiyondan hatayı yakalayan uygulamaya iletilmesi için bir yöntem bulabilir miyiz?

Kural dışı durum fırlatılırken, sadece kontrolü kural dışı durum işleyicisine devretmekle kalmadığımızı, aynı zamanda kurucusunu çağırdığımız kural dışı durum sınıfından bir nesne oluşturduğumuzu da hatırlarsanız bu soruya cevap verebilirsiniz. Örneğin **XDIST**'te

```
throw InchesEx();
```

ifadesi ile kural dışı bir durum fırlatıldığında **InchesEx** tipinden bir nesne oluşturulur. Bu kural dışı durum sınıfına veri üyeleri eklersek, nesneyi oluşturduğumuzda bu üyelere başlangıç değerleri atayabiliriz. Böylece kural dışı durum işleyicisi, kural dışı durumu yakaladığı zaman o nesnenin verilerini alabilir. Bu tıpkı, bir topun üzerine mesaj yazıp topu komşunuzun bahçesine atmaya benzer. Şimdi gelin **XDIST** programını değiştirerek bu anlattıklarımızı uygulayalım:

```
// xdist2.cpp
// argumanli kural disi durumlar
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                  //Ingiliz olculerine gore Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
//-------------------------------------------------------------
      class InchesEx                        //kural disi durum sinifi
         {
         public:
            string origin;                  //rutinin adi icin
            float iValue;                   //hatali inc degeri icin

            InchesEx(string or, float in)   //iki argumanli kurucu fonk.
               {
               origin = or;                 //karakter katarini tut
               iValue = in;                 //inc degerini tut
               }
         };                                 //kural disi durum sinifinin sonu
//-------------------------------------------------------------
      Distance()                            //kurucu fonk.(argumansiz)
         { feet = 0; inches = 0.0; }
//-------------------------------------------------------------
      Distance(int ft, float in)            //kurucu fonk.(iki argumanli)
         {
         if(in >= 12.0)
            throw InchesEx("2-arg constructor", in);
         feet = ft;
         inches = in;
         }
//-------------------------------------------------------------
      void getdist()                        //uzunluk degerini kullanicidan al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         if(inches >= 12.0)
            throw InchesEx("getdist() function", inches);
         }
//-------------------------------------------------------------
      void showdist()                       //mesafeyi goster
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
```



```
   try
      {
      Distance dist1(17, 3.5);
      Distance dist2;                   //iki argumanli kurucu fonk.
      dist2.getdist();                  //argumansiz kurucu fonk.
                                        //degeri al
                                        //uzakligi goster
      cout << "\ndist1 = "; dist1.showdist();
      cout << "\ndist2 = "; dist2.showdist();
      }
   catch(Distance::InchesEx ix)
      {                                 //kural disi durum yoneticisi
      cout << "\nInitialization error in " << ix.origin
           << ".\n   Inches value of " << ix.iValue
           << " is too large.";
      }
   cout << endl;
   return 0;
   }
```

Bir kural dışı durum fırlatırken veri aktarma işleminin üç aşaması vardır: Kural dışı durum sınıfı için veri üyelerini ve bir kurucuyu belirlemek, kural dışı durumu fırlatırken bu kurucuyu ilk kullanıma hazırlamak ve kural dışı durumu yakaladığımızda nesnenin verilerine ulaşmak. Gelin tüm bunları tek tek ele alalım:

Bir Kural Dışı Durum Sınıfında Verileri Belirlemek

Kural dışı durum sınıflarındaki verileri public olarak tanımlarsak, kural dışı durum yöneticisi bu verilere doğrudan ulaşabilir ve bu sayede işimiz kolaylaşır. **XDIST2**'de **InchesEx** adlı yeni kural dışı durum sınıfının spesifikasyonu şu şekildedir:

```
class InchesEx                          //kural disi durum sinifi
   {
   public:
      string origin;                    //rutinin adi icin
      float iValue;                     //hatali inches degeri icin

      InchesEx(string or, float in)     //iki argumanli kurucu
         {
         origin = or;                   //nesneye string koy
         iValue = in;                   //nesneye inches degeri koy
         }
   };
```

Bir **string** nesnesi için public değişkenler vardır. Bunlar, çağrılan üye fonksiyonun adını ve hatalı inç değerini (**float** tipi) tutar.

Kural Dışı Durum Nesnesini İlk Kullanıma Hazırlamak

Kural dışı bir durum fırlattığımızda verileri nasıl ilk kullanıma hazırlarız? **Stack** sınıfının iki argümanlı kurucusunda şöyle yapılır:

```
throw InchesEx("2-arg constructor", in);
```

Stack'in **getdist()** üye fonksiyonunda da şöyle denir:

```
throw InchesEx("getdist()function", inches);
```

Kural dışı durum fırlatıldığında, ilgili fonksiyon karakter katarı ve **inches** değerlerini ekranda gösterir. **karakter katarı**, kural dışı durumu hangi üye fonksiyonun fırlattığını; **inches** değeri de üye fonksiyonun yakaladığı hatalı ölçüm değerini verir. Bu ilave bilgiler, programcının hata kaynağını saptamasını kolaylaştırır.

Kural Dışı Durum Nesnesinden Veri Çıkarmak

Bir kural dışı durum yakaladığımız zaman bundan nasıl veri çıkarabiliriz? Bunun en basit yolu, burada bizim de yaptığımız gibi, veriyi kural dışı durum sınıfının public bir parçası haline getirmektir. Daha sonra, yakalama bloğunda **ix**'i yakaladığımız kural dışı durum nesnesinin adı olarak bildirebiliriz. Bu ismi kullanarak nesnenin verisine normal yolla, yani nokta operatörü ile ulaşabiliriz:

```
catch(Distance::InchesEx ix)
   {
   //'ix.origin' ve 'ix.iValue'ya dogrudan erisim.
   }
```

Ardından, **ix.origin** ve **ix.iValue** değerlerini ekrana çıkarabiliriz. **XDIST2**'de kullanıcının **inches** için limit dışı değer girdiği örnek bir çalışmanın çıktısı şöyledir:

```
Enter feet: 7
Enter inches: 13.5

Initialization error in getdist() function.
   Inches value of 13.5 is too large.
```

Benzer şekilde, programcı **main()**'deki **dist1** tanımını şu şekilde değiştirirse

```
Distance dist1(17, 22.25);
```

ortaya çıkan kural dışı durum şu hata mesajına neden olacaktır:

```
Initialization error in 2-arg constructor.
   Inches value of 22.25 is too large.
```

Elbette, kural dışı durum argümanlarını istediğimiz gibi kullanabiliriz. Ancak bunlar genelde kural dışı durumu ortaya çıkaran hatayı teşhis etmemize yardımcı olan bilgiler taşırlar.

bad_alloc Sınıfı

Standart C++'ta bazı hazır kural dışı durum sınıfları vardır. Bunlar arasında muhtemelen en yaygın kullanılanı **bad_alloc**'tur. **bad_alloc**, **new** ile bellek alanı ayırmaya çalışırken hata olması durumunda fırlatılır. (C++'ın eski versiyonlarında bu kural dışı duruma **xalloc** deniyordu. Kitabımız basıma hazırlanırken, Microsoft Visual C++'ta hala bu eski yaklaşım geçerliydi.) Uygun deneme ve yakalama blokları oluşturursanız, **bad_alloc**'u çok az çabayla kullanabilirsiniz. Kullanımı örneklemek üzere **BADALLOC** adlı kısa bir program parçasına bakıyoruz:

```
// badalloc.cpp
// bad_alloc kural disi durumu
#include <iostream>
```



```
using namespace std;

int main()
   {
   const unsigned long SIZE = 10000;          //bellek buyuklugu
   char* ptr;                                 //bellegi gosteren isaretci

   try
      {
      ptr = new char[SIZE];
      }                                       //SIZE kadar byte ayir
   catch(bad_alloc)
      {                                       //kural disi durum yoneticisi
      cout << "\nbad_alloc exception: can't allocate memory.\n";
      return(1);
      }
   delete[] ptr;                              //ayrilmis bellegi serbest birak
   cout << "\nMemory use is successful.\n";
   return 0;
   }
```

new kullanan tüm ifadeleri bir deneme bloğu içine koyun. Ardından gelen yakalama bloğu, kural dışı durumu kontrol altına alır; bunu yaparken de çoğunlukla bir hata mesajı gösterir ve programı sonlandırır.

Kural Dışı Durumlarla İlgili Notlar

Kural dışı durumları kullanmanın sadece en basit ve en yaygın yöntemlerini göstermekle yetindik. Daha fazla ayrıntıya girmeyeceğiz. Kural dışı durumların kullanımıyla ilgili bazı noktalara göz gezdirerek bu bölümü tamamlayacağız.

Kümelenmiş Fonksiyonlar

Kural dışı duruma neden olan ifadenin doğrudan deneme bloğunun içinde yer alması gerekmez; bu ifade deneme bloğundaki bir ifadenin çağırdığı bir fonksiyonda da yer alabilir. (Hatta, bir kaç seviye kümelenmiş fonksiyonların en içindeki bir ifade de kural dışı duruma neden olabilir.) Bu nedenle, programın en üst seviyesine bir deneme bloğu koymuş olmanız yeterlidir. Daha alt seviyedeki fonksiyonları o kadar karmaşık hale getirmeye gerek kalmaz. Yeter ki, bu fonksiyonları doğrudan ya da dolaylı olarak çağıran fonksiyonlar, deneme bloğunda yer alsın. (Ancak bazen iç içe fonksiyonların orta seviyelerindeki fonksiyonların da kural dışı durum mesajlarına kendi tanımlayıcı verilerini eklemeleri ve bunu bir üst seviyeye göndermeleri gerekebilir.)

Kural Dışı Durumlar ve Sınıf Kütüphaneleri

Kural dışı durumlar tarafından çözülen önemli bir problem, sınıf kütüphanelerindeki hatalardır. Bir kütüphane rutini bir hatayı keşfedebilir ama böyle bir durumda ne yapması gerektiğini genelde bilmez. Ne de olsa, kütüphane fonksiyonunu yazan kişi de yazım zamanı da farklıdır. İşte böyle durumlarda kütüphane fonksiyonu hatayı, kendisini çağıran koda bildirerek "Burada bir hata oldu. Hata konusunda ne yapmak istediğini bilmiyorum ama böyle bir hata olduğunu bil" anlamına bir mesaj gönderme ihtiyacı hisseder. Kütüphane fonksiyonunu çağıran program da o hatayla ilgili olarak nasıl uygun görüyorsa o şekilde işlem yapar.

Kural dışı durum mekanizması bu yeteneği sağlar. Çünkü kural dışı durumlar, bir yakalama bloğu bulunana kadar iç içe fonksiyonların en üst düzeyine doğru aktarılırlar. Kural dışı durumu ortaya çıkarak ifade bir kütüphane fonksiyonunda yer alabilir. Ancak bu durumda yine de yakalama bloğu, hatayı nasıl değerlendireceğini bilen asıl programda olabilir.

Bir sınıf kütüphanesi yazıyorsanız, bunu kullanan programlar açısından problem oluşturacak her türlü hatanın kural dışı durum fırlatmasını sağlamalısınız. Bir sınıf kütüphanesi kullanan bir program yazıyorsanız, kütüphanenin fırlatabileceği hata durumlarını deneyecek ve yakalayacak bloklar oluşturmalısınız.

Her Duruma Uymaz

Kural dışı durumlar tüm hata türlerine uymaz. Çünkü bunlar, program boyutu ve (kural dışı durum medyana geldiğinde de) zaman üzerinde bazı kısıtlamalar getirir. Örneğin, kural dışı durumlar muhtemelen kullanıcının giriş hataları için (nümerik veri yerine harf girmek gibi) kullanılmamalıdır. Bu tür hatalar zaten program tarafından kolaylıkla yakalanabilir. Böyle hallerde, kural dışı durumlardan yararlanmak yerine normal karar yapılarını ve döngüleri kullanarak kullanıcının girdiği veriler kontrol edilmeli ve gerekirse yeniden girdi istenmelidir.

Yok Edici Fonksiyonlar Otomatik Olarak Çağrılır

Kural dışı durum mekanizması, şaşırtıcı derecede sofistikedir. Kural dışı bir durum fırlatıldığında, deneme bloğunda o noktaya kadar geçen kod tarafından oluşturulan her hangi bir nesne varsa o nesne için bir yok edici fonksiyon otomatik olarak çağrılır. Bu, gereklidir; çünkü uygulama hataya hangi ifadenin neden olduğunu bilemez, hatadan kurtulmaya çalışmaktadır ve (en iyi ihtimalle) deneme bloğunun başına dönüp yeniden işe başlamak isteyecektir. Kural dışı durum mekanizması, deneme bloğundaki kodun, nesnelerin varlığı açısından "reset" edilmesini garanti altına alır.

Kural Dışı Durumların Yönetilmesi

Kural dışı bir durumu yakaladığınızda bazen uygulamanızı sona erdirmek istersiniz. Kural dışı durum mekanizması, programı sona erdirmeden önce hatanın kaynağını kullanıcıya bildirme ve eğer gerekiyorsa bazı temizlik işlemleri yapma imkanı verir. Deneme bloğunda oluşturulan nesneler için yok edici fonksiyonları çalıştırarak temizlik işlemini kolaylaştırır. Böylece, deneme bloğundaki nesneler bellek gibi sistem kaynakları kullanıyorsa bu kaynakları serbest bırakma imkanı elde edersiniz.

Diğer bazı durumlarda programı sona erdirmeyi tercih etmeyebilirsiniz. Belki de programınız hatanın nedenini bulabilir ve bu durumu ortadan kaldırabilir. Belki de kullanıcıdan daha farklı veri girmesi istenerek problem çözülebilir. Böyle durumlarda, deneme ve yakalama blokları tipik olarak bir döngünün içine yerleştirilir. Bunun amacı, kontrolün, (kural dışı durum mekanizmasının ilk haline getirmeye çalıştığı) döngü başına yeniden iade edilmesidir.

Fırlatılan kural dışı durumla eşleşen bir kural dışı durum yönetim rutini yoksa, program sessiz sedasız işletim sistemi tarafından sonlandırılır.

Özet

Şablonlar, farklı veri tipleri üzerinde çalışabilmeniz için fonksiyon ya da sınıf aileleri oluşturmanızı sağlarlar. Tıpatıp aynı işlemleri farklı veri tipleri üzerinde yapmak için farklı fonksiyonlar yazmanız gerektiğini fark ederseniz, fonksiyon şablonlarını kullanmayı alternatif olarak değerlendirmeniz gerekir. Benzer şekilde, sadece üzerinde işlem yaptıkları veri tipleri bakımından ayrılan, onun dışında tıpatıp birbirinin aynı olan sınıf spesifikasyonları yazıyorsanız, burada da sınıf şablonları kullanmayı dikkate almalısınız. Şablon kullanırsanız, zamandan tasarruf edersiniz ve daha güçlü, daha sağlam, bakımı ve (şablonları anladıktan sonra) anlaması daha kolay kod yazma imkanına kavuşursunuz.

Kural dışı durumlar, C++ hatalarını sistematik ve nesne yönelimli programlama mantığına uygun bir şekilde işlemeye yarayan bir mekanizma oluşturur. Kural dışı bir durum genellikle bir sınıfın nesneleri üzerinde işlem yapan bir deneme bloğundaki hatalı bir ifadeden kaynaklanır. Sınıf üyesi fonksiyon bu hatayı yakalar ve kural dışı durum oluşturur. Bu kural dışı durum, program tarafından, deneme bloğunun peşinden gelen "kural dışı durum yöneticisi" kodundaki sınıf kullanılarak yakalanır.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Şablonlar, _____ bir aile oluşturmanın kolay bir yoludur.
   a. değişkenlerden
   b. fonksiyonlardan
   c. sınıflardan
   d. programlardan
2. Şablon argümanından önce şu anahtar kelime kullanılır: _____.
3. Doğru/Yanlış: Kullanıcı girdisine bağlı olarak şablonlar bir fonksiyonun farklı versiyonlarını oluşturur.
4. Her zaman argümanının iki katını döndüren bir fonksiyonun şablonunu yazın.
5. Bir şablon sınıfı
   a. farklı konteynerlerde saklanmak üzere tasarlanmıştır.
   b. değişik veri tipleriyle çalışır.
   c. birbirinin tıpatıp aynı olması gereken nesneler üretir.
   d. Farklı sayılarda üye fonksiyonları olan sınıflar üretir.
6. Doğru/Yanlış: Şablon argümanının sayısı birden fazla olabilir.
7. Bir şablondan gerçek bir fonksiyon oluşturmaya "fonksiyonun _____" denir.
8. Bir şablon fonksiyonun gerçek kodu ne zaman oluşturulur?
   a. Fonksiyon deklarasyonu kaynak kodunda yer aldığında.
   b. Fonksiyon tanımı kaynak kodunda yer aldığında.
   c. Fonksiyonu çağıran bir ifade kaynak kodunda yer aldığında.
   d. Fonksiyon, program çalışması sırasında yürütüldüğünde.
9. Şablonlarda kilit önem taşıtan kavram, bir _____, yerine _____ ifade eden bir isim kullanmaktır.
10. Şablonlar sıklıkla _____ sınıflar için kullanılırlar.
11. Kural dışı bir durum tipik olarak şundan kaynaklanır:
    a. Uygulamayı yazan programcıdan.
    b. Sınıf üye fonksiyonlarını yazan sınıfı oluşturandan.
    c. Programın çalışması sırasında ortaya çıkan bir hatadan.
    d. Programı sonlandıran bir işletim sistemi arızasından.
12. Kural dışı durumlarla kullanılan C++ anahtar kelimeleri _____, _____ ve _____'dir.

13. Gövdesi boş olan `BoundsError` sınıfını kullanarak kural dışı bir durum oluşturan bir ifade yazın.
14. Doğru/Yanlış: Kural dışı bir duruma neden olabilecek ifadeler bir yakalama bloğunun parçası olmalıdır.
15. Kural dışı durumlar nereden nereye fırlatılırlar?
    a. Yakalama bloğundan deneme bloğuna.
    b. Yakalama ifadesinden deneme bloğuna.
    c. Hatanın oluştuğu noktadan bir yakalama bloğuna.
    d. Bir kural dışı durum fırlatma ifadesinden bir yakalama bloğuna.
16. Hata kodu ile hata ismini saklayacak şekilde bir kural dışı durum sınıfının spesifikasyonunu yazın. Bunu yaparken kurucu fonksiyon da kullanın.
17. Doğru/Yanlış: Kural dışı bir durum fırlatan bir ifadenin bir deneme bloğunda yer alması gerekli değildir.
18. Aşağıdaki durumların hangisinde tipik olarak kural dışı bir durum fırlatılır?
    a. Aşırı miktarda veri zaten bir dizide taşmaya tehlikesi doğurur.
    b. Kullanıcı programı sonlandırmak için Ctrl+C tuş kombinasyonuna basar.
    c. Elektrik kesilmesi sonucu sistem kapanır.
    d. `new` anahtar kelimesi talep edilen bellek alanını alamaz.
19. Kural dışı durum iletisinde ilave bilgiler nereye konulabilir?
    a. `throw` anahtar kelimesine
    b. hataya neden olan fonksiyona
    c. yakalama bloğuna
    d. kural dışı durum sınıfının bir nesnesine
20. Doğru/Yanlış: Bir program, kural dışı bir durum fırlatıldıktan sonra çalışmaya devam edebilir.
21. Bağımlılıklardan söz ederken, şablon sınıfı _____ öğe, örneklenmiş sınıf da _____ öğedir.
22. Bir şablon sınıfı UML'de nasıl gösterilir?
    a. İlaveleri olan sıradan bir sınıf olarak.
    b. Kesikli çizgi ile.
    c. Çevresi kesikli çizgiyle belirlenmiş bir dikdörtgen şeklinde.
    d. Yukarıdakilerden hiç biri.
23. Doğru/Yanlış: Bağımlılık (dependency) bir tür bağlantıdır (association).
24. Bir klişe (stereotype) bir UML öğesi hakkında _____ verir.

Alıştırmalar

Yıldızla işaretli alıştırmaların cevapları Ek G'dedir.

1. Bir dizideki elemanların ortalamasını döndüren bir şablon fonksiyonu yazın. Fonksiyonun argümanları; dizi ismi ile dizinin boyutu (`int` tipi) olsun. `main()` programda, fonksiyonu `int`, `long`, `double` ve `char` tipinden diziler üzerinde kullanın.*
2. Kuyruk (queue) bir veri saklama sistemidir. Yığına benzer ama ondan farklı olarak son girenin ilk çıktığı değil de ilk girenin ilk çıktığı bir düzendir. Tıpkı bankadaki bir işlem bankosunun önünde oluşan kuyruk gibi: 1, 2, 3 sayılarını bu sırada kuyruğa sokarsanız, çıkıştaki sıra da 1, 2, 3 olur.*

Yığını uygularken, kullanılan dizine tek bir indeks olması yeterlidir (Bölüm 7'deki `STAKARAY` programına bakın). Kuyrukta ise hem yeni öğelerin eklendiği arka uca hem de eski isimlerin kuyruğu terk ettiği baş ucuna işaret eden indekslere ihtiyaç vardır. Öğeler kuyruğa girip çıktıkça kuyruğun başı ile sonu hareket eder. Kuyruğun başı ya da sonu, kullanılan dizinin sınırlarına ulaşırsa, o indeks çevrilip yeniden başa getirilir. Kuyruk sınıfı için bir şablon yazın. Kuyruğu kullanan programcının, kuyruğun kapasitesini aşmak ya da boş bir kuyruktan unsur çıkarmaya çalışmak gibi hatalar yapmayacağını varsayın. Farklı veri tipleri için çeşitli kuyruklar tanımlayın ve bunlara veri koyup bunlardan veri çıkarın.*

3. Yukarıdaki 2. alıştırmada kullandığınız kuyruk şablonuna kural dışı durumlar ekleyin. İlk kural dışı durum, dizinin kapasitesi aşıldığında fırlatılsın. Diğeri de boş bir kuyruktan eleman çıkarılmaya çalışıldığında tetiklensin. Örnek bir çözüm şöyle olabilir: Kuyruğa ilave bir veri öğesi ekleyin ve bu öğe kuyruktaki öğelerin sayısını tutsun. Yeni bir öğe eklendiğinde ya da eksildiğinde bu sayı da ona göre değişsin. Sayı kuyruğun kapasitesini aşarsa ya da negatif olursa kural dışı bir durum fırlatın.*
   Bu programın `main()` kısmını etkileşimli yaparak kullanıcının kuyruğa veri sokup kuyruktan veri çıkarmasını sağlayabilirsiniz. Böylece kuyruğu denemeniz daha kolay olur. Kural dışı durum fırlatılınca, program, kuyruğun içeriği bozulmaksızın kullanıcının hata durumundan kurtulabilmesine izin vermelidir.
4. Kendisine gönderilen iki argümanın değerlerini takas eden `swap()` adlı bir fonksiyon yazın. (Bu argümanları muhtemelen referans olarak geçmeyi tercih edeceksinizdir.) Fonksiyonu, tüm nümerik veri tipleri (`char`, `int`, `float` vb.) ile kullanılabilecek şekilde bir şablona çevirin. Fonksiyonu çeşitli veri tipleri üzerinde denemek için bir `main()` programı yazın.
5. Bir dizideki en büyük elemanın değerini döndüren `amax()` adlı bir fonksiyon yazın. Fonksiyonun argümanları; dizinin adresi ve büyüklüğü olmalıdır. Fonksiyonu, herhangi bir nümerik veri tipinden dizi ile çalışabilecek bir fonksiyon haline getirin. Fonksiyonu farklı veri tiplerinden diziler üzerinde deneyen bir `main()` programı yazın.
6. Bölüm 8'deki `ARROVER3` programında kullanılan `safearay` sınıfı ile başlayın. Bu sınıfı, her tür veri saklayabilecek bir şablon haline getirin. `main()`'de en az iki farklı tipte güvenli diziler oluşturun ve bunlara değerler koyun.
7. Bölüm 8'te 7. alıştırmadaki `frac` sınıfı ve kesirler üzerinde dört işlem yapabilen hesap makinesi programını alın. `frac` sınıfını, pay ve paydada farklı veri tipleri kullanılarak örneklenebilecek şekilde bir şablona dönüştürün. Bu veri tiplerinin tamsayı tipleri olması gerektiği için `char`, `short`, `int` ve `long` ile kısıtlısınız. (kendinize ait bir `int` tipi geliştirirseniz o başka...) `main()`'de `frac<char>` sınıfının kopyasını oluşturun ve bunu dört işlem hesap makinesinde kullanın. `frac<char>` sınıfı `frac<int>` sınıfından daha az yer kaplayacaktır ama büyük kesirler üzerinde işlem yapamayacaktır.
8. Bölüm 8'deki `ARROVER3` programına, limit dışına taşan bir indeksin kural dışı bir durum fırlatmasını sağlayacak şekilde bir kural dışı durum sınıfı ekleyin. Yakalama bloğu, kullanıcıya bir hata mesajı gösterebilir.
9. (`ARROVER3`'ten uyarladığımız ve) 8. alıştırmada kullandığımız kural dışı durum sınıfını değiştirerek, yakalama bloğundaki hata mesajının bu duruma neden olan indeks değerini göstermesini sağlayın.
10. Kural dışı durumların ne zaman kullanılması gerektiğine dair farklı düşünceler vardır. "Akışlar ve Dosyalar" adlı Bölüm 12'deki `ENGLERR` programına dönün. Burada kullanıcının yaptığı hatalar kural dışı durumlar oluşturmalı mıdır? Bu alıştırmada öyle olması

gerektiğini varsayalım. Programdaki `Distance` sınıfına bir kural dışı durum sınıfı ekleyin. (Bu bölümdeki `XDIST` ve `XDIST2` örneklerine de bakın.) `ENGLERR`'in hata mesajı verdiği her yerde kural dışı durum fırlatın. Kural dışı durum kurucusunun bir argümanını kullanarak hatanın nerede olduğunu ve hatanın spesifik nedenini bildirin (`inches` için sayı girilmemiş, `inches` limit dışına taşmış, gibi...) `isint()` fonksiyonunda hata olduğu zaman da kural dışı durum fırlatın (hiç bir şey girilmemiş, hane sayısı limiti aşılmış, rakam olmayan karakterler girilmiş, tam sayı limit dışına taşmış gibi.) Soru: `isint()` kural dışı durum fırlatmaya başladığı andan itibaren artık bağımsız bir fonksiyon olarak kalabilir mi?

Hem deneme bloğunu hem de yakalama bloğunu `do` döngüsü içine yerleştirebilirsiniz. Böylece kural dışı bir durum fırlattıktan sonra yeniden döngünün başına giderek kullanıcıdan yeni girdi isteyebilirsiniz. Ayrıca, iki argümanlı kurucu fonksiyonda programcının, `Distance` değerine, limit dışına taşan bir `inches` üyesi ile ilk değeri atanması durumunda fırlatılacak kural dışı bir durum da tanımlayabilirsiniz.

11. Bölüm 8'deki `STRPLUS` programı ile başlayın. Bu sınıfa kural dışı bir durum ekleyin ve başlangıç karakter katarı aşırı uzun gelirse tek argümanlı kurucu fonksiyonda kural dışı durum fırlatılmasını sağlayın. Bir başka kural dışı durum da aşırı yüklenmiş + operatöründe olsun; iki karakter katarının birbirine eklenmesi sonucunda ortaya çıkan karakter katarı aşırı uzun olursa kural dışı durum fırlatılsın. Hatalardan hangisinin ortaya çıktığını belirtin.
12. Bazen kural dışı durumları kullanmanın en kolay yolu, kural dışı durum sınıfının üye olduğu yeni bir sınıf oluşturmaktır. Bunu, dosya hatalarından kaynaklanan kural dışı durumları yöneten bir sınıf ile deneyin. Dosyadan okumak ve dosyaya yazmak için üye fonksiyonları ve kural dışı durum yönetici sınıfı olan `dofile` adlı bir sınıf tanımlayın. Sınıfın kurucu fonksiyonu, dosya ismini argüman olarak almalı ve bu adla bir dosya açmalıdır. Dosya işaretçisini dosyanın başına götürecek bir üye fonksiyon da yazabilirsiniz. Bölüm 12'deki `REWERR` programını model alarak, aynı işlevlere sahip ancak `dofile` sınıfının üyelerini çağırarak çalışan bir `main()` program yazın.

ONBEŞİNCİ BÖLÜM

15

STANDART ŞABLON KÜTÜPHANESİ

STL'e Giriş

Algoritmalar

Sekans Konteynerler

İteratörler

Özelleştirilmiş İteratörler

Birleşik Konteynerler

Kullanıcı Tarafından Tanımlanan Nesneleri Depolamak

Fonksiyon Nesneler

Bilgisayar programlarının çoğu verileri işlemek için mevcuttur. Veriler, gerçek dünyaya ait çok çeşitli bilgileri simgeleyebilir: Personel kayıtları, envanterler, metin belgeleri, bilimsel deneylerin sonuçları vs. Neyi simgelerse simgelesin, veri bellekte saklanır ve benzer yöntemlerle yönetilir. Üniversitelerin bilgisayar mühendisliği programları genellikle "Veri Yapıları ve Algoritmalar" adında bir ders içerir. Burada *veri yapıları* terimi verileri bellekte saklamak için kullanılan çeşitli yöntemleri ifade eder; *algoritmalar* terimi ise verilerin nasıl yönetileceği ile ilgilidir.

C++ sınıfları, bir veri yapıları kütüphanesi oluşturmak için kusursuz bir mekanizma sağlar. Geçmişte, verilerin saklanmasını ve işlenmesini kontrol altına almak amacıyla derleyici satıcıları ve üçüncü taraf geliştiriciler *konteyner sınıflar* (*container classes*) kütüphanesini piyasaya sundular. Ancak, şimdilerde Standart C++ kendisine ait hazır konteyner sınıf kütüphanesini içerir. Bu, Standart Şablon Kütüphanesi (Standard Template Library - STL) olarak adlandırılır ve Hewlett Packard'dan Alexander Stepanov ve Meng Lee tarafından geliştirilmiştir. STL, Standart C++ sınıf kütüphanesinin bir parçasıdır ve verileri saklamak ve işlemek için standart bir yöntem olarak kullanılabilir.

Bu bölümde STL ve STL'in nasıl kullanıldığı ele alınıyor. STL geniş ve karmaşıktır. Bu nedenle, ne pahasına olursa olsun STL hakkında her şeyi ele almayacağız; aksi halde bu, büyük bir kitap yazmayı gerektirirdi. (STL hakkında birçok kitap piyasada mevcuttur.) Bu bölümde STL'e bir giriş yapacağız ve en çok kullanılan algoritmalar ve konteynerlerle ilgili örnekler vereceğiz.

STL'e Giriş

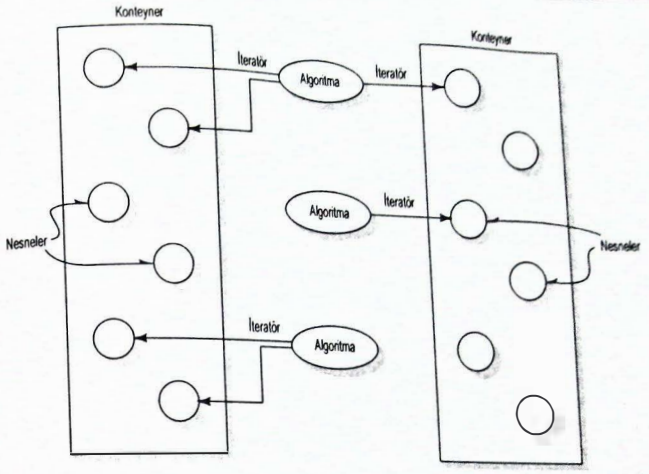
STL birkaç çeşit unsur içerir. Bunlardan en önemli üçü konteynerler (container), algoritmalar ve iteratörlerdir (iterator).

Konteyner, saklanmış verilerin bellekte organize edilme yöntemlerinden biridir. Önceki bölümlerde iki çeşit konteyneri incelemiştik: Yığınlar ve bağlı listeler. Bir başka konteyner olan dizi ise o kadar yaygın kullanıma sahiptir ki bu, C++'a (ve diğer bilgisayar dillerine) standart olarak dahil edilmiştir. Bununla birlikte, başka birçok konteyner daha mevcuttur. STL, bunlardan en kullanışlı olanlarını içerir. STL konteynerleri şablon sınıfları ile gerçeklenir, böylece farklı türden verileri tutmak için kolaylıkla adapte edilebilir.

STL'deki *algoritmalar* konteynerlerdeki verileri çeşitli yöntemlerle işlemek için konteynerlere uygulanan prosedürlerdir. Örneğin, sıralamak, kopyalamak, aramak ve birleştirmek için algoritmalar vardır. Algoritmalar şablon fonksiyonları ile simgelenirler. Bu fonksiyonlar, konteyner sınıflarının üye fonksiyonları değildir. Daha doğrusu, bunlar tek başına mevcut olan fonksiyonlardır. Gerçekten, STL'in en çarpıcı özelliklerinden biri, algoritmalarının çok genel oluşudur. Bu algoritmaları sadece STL konteynerlerinde kullanmayıp, aynı zamanda sıradan C++ dizilerinde ve kendi kendinize geliştirdiğiniz konteynerler üzerinde de kullanabilirsiniz. (Konteynerler ayrıca daha spesifik görevler için de üye fonksiyonlar içerirler.)

İteratörler işaretçi kavramının genelleştirilmiş bir şeklidir: Konteyner içindeki elemanlara işaret ederler. Bir işaretçiyi arlırabildiğiniz gibi bir iteratörü da artırabilirsiniz. Böylece, iteratör sırayla konteyner içindeki bir sonraki elemanı gösterir. İteratörler STL'in en temel parçasıdır, çünkü iteratörler, algoritmalar ile kontyenerleri birleştirirler. Bunları yazılım dünyasındaki kablolar gibi düşünün (müzik seti bileşenlerini birbirine bağlayan veya bir bilgisayarı çevre birimlere bağlayan kablolar gibi).

Şekil 15.1'de STL'in üç ana bileşeni gösteriliyor. Bu bölümde konteyner, algoritma ve iteratörleri biraz daha ayrıntılı ele alacağız. Sonraki bölümlerde bu kavramları program örnekleriyle birlikte daha da dikkatli inceleyeceğiz.



ŞEKİL 15.1: Konteynerler, algoritmalar ve iteratörler.

Konteynerler

Konteyner (container), bir çeşit veri saklama yöntemidir. Verilerin hazır tiplerden (`int` ve `float` gibi) veya sınıf nesnelerinden oluşup oluşmadığı önemli değildir. STL, temel nitelikte yedi çeşit konteynere imkan verir. Ayrıca, temel nitelikteki çeşitlerden türetilen üç konteyner tipi daha vardır. İlaveten, siz de temel niteliğe sahip konteyner çeşitlerine dayanan kendi konteynerlerinizi tasarlayabilirsiniz. Niye bu kadar çok çeşitte konteynere ihtiyaç duyduğumuzu merak ediyor olabilirsiniz. Veri saklama gerektiren durumların tümünde neden C++ dizilerini kullanmayalım ki? Yanıt, verimliliktir. Diziler bir çok durumda kullanışsız ve yavaş kalır.

STL'deki konteynerler iki ana kategoriye ayrılır: *Sekans* (sequence) ve *birleşik* (associative). Sekans konteynerler *vektör* (*vector*), liste (*list*) ve *çift uçlu kuyruktur* (deque). Birleşik konteynerler ise *çoklu küme*, *eşleme* (*map*) ve *çoklu eşlemedir* (*multimap*). İlaveten, birkaç özelleştirilmiş konteyner de sekans konteynerlerden türetilir. Bunlar *yığın* (*stack*), *kuyruk* (*queue*) ve *öncelik kuyruğudur* (*priority queue*). Bu kategorilere sırayla göz atacağız.

Sekans Konteynerler

Bir sekans konteyner, tıpkı bir cadde üzerindeki evler gibi bir sıra şeklinde göz önünde canlandırılabilen elemanları saklar. Her eleman sıradaki konumu itibariyle diğer elemanlarla ilişkilidir. Her eleman (uçtakiler hariç) belirli bir elemanın peşinden gelir ve bir başka eleman tarafından takip edilir. Sıradan bir C++ dizisi sekans konteyner örneklerinden biridir.

C++ dizilerinde karşılaşılan bir problem, dizinin büyüklüğünü derleme zamanında, yani kaynak kod içinde, belirtmeniz gerektiğidir. Ne yazık ki, programı yazdığınız esnada genellikle dizide ne kadar veri saklanacağını bilmezsiniz. Bu nedenle, en çok veri miktarı olarak tahmin

ettiğiniz verileri tutabilecek kadar büyük bir dizi belirtmeniz gereklidir. Program çalıştığı sırada ya diziyi dolduramadığınız için bellekten yer harcamış olacaksınız, ya da yer yetmediği için bir hata mesajı alacaksınız (hatta programınızı uçurabilirsiniz bile). STL bu tür zorlukları önlemek için *vektör* konteynerini sunar.

İşte dizilerle ilgili bir başka problem daha. Diyelim ki, personel kayıtlarını saklıyorsunuz ve bu kayıtları personelin soyadına göre alfabetik olarak düzenlediniz. Şimdi ismi L harfi ile başlayan yeni bir personeli eklemek isterseniz, bu yeni kayıta yer açmak için M'den Z'ye kadar tüm kayıtları kaydırmanız gereklidir. Bu işlem çok zaman alıcı olabilir. STL bu problemi çözmek için bağlı liste kavramına dayanan *liste* konteynerini sunar. Bölüm 10'da "İşaretçiler" bahsindeki `LINKLIST` örneğinden hatırlarsanız, bir bağlı listeye yeni öğe eklemek, birkaç işaretçiyi yeniden düzenleyerek yapılan kolay bir işlemdir.

Sekans konteynerlerin üçüncüsü, bir yığın ve kuyruğun (queue) birleşimi gibi düşünülebilen bir *çift uçlu kuyruktur* (*deque*). Yığın, önceki örneklerden hatırlayabileceğiniz gibi, son giren ilk çıkar prensibine dayanır. Giriş ve çıkışların her ikisi de yığının tepesinden yapılır. Öte yandan kuyruk, ilk giren ilk çıkar prensibini kullanır: Veriler, tıpkı bir banka kuyruğundaki müşteriler gibi, önceden girerler ve arkadan çıkarlar. Çift uçlu kuyruk ise, bu iki yöntemi birleştirir. Böylece, her iki uçtan da ekleme ve çıkarma yapabilirsiniz. Deque kelimesi Double-Ended QUEue'dan (Çift Uçlu Kuyruk) türetilmiştir. Bu, çok yönlü bir mekanizmadır. Öyle ki, kendi başına kullanışlı olmasının yanı sıra yığınlar ve kuyruklar için bir temel olarak da kullanılabilir (bunu daha sonra göreceksiniz).

Tablo 15.1 STL sekans konteynerlerinin özeliklerini özetler. Tablo, karşılaştırma amacıyla sıradan C++ dizisini de içerir.

TABLO 15.1: Temel Sekans Konteynerler

Konteyner	Özellik	Avantaj ve Dezavantaj
sıradan C++ dizisi	Sabit boyut	Hızlı rasgele erişim (indeks numarası vasıtasıyla)
		Ortaya eklemek ve ortadan çıkarmak yavaş
		Çalışma zamanında boyut değiştirilemez
vektör	Yeniden konumlandırabilir, Genişletilebilir dizi	Hızlı rasgele erişim (indeks numarası vasıtasıyla)
		Ortaya eklemek ve ortadan çıkarmak yavaş
		Uçlara eklemek veya uçlardan çıkarmak hızlı
liste	Çift bağlantılı bağlı liste	Her konumda eklemek ve çıkarmak hızlı
	Rasgele erişim yavaş	Her iki uca hızlı erişim
çift uçlu kuyruk	Vektör gibi, fakat her iki uçtan erişilebilir.	Hızlı rasgele erişim (indeks numarası vasıtasıyla)
		Ortaya eklemek ve ortadan çıkarmak yavaş
		Başından veya sonundan hızlı ekleme ve çıkarma

Bir STL nesnesine başlangıç değeri atamak kolaydır. Öncelikle, ilgili başlık dosyasını dahil etmelisiniz. Sonra, saklanacak nesnelerin türünün parametre olarak aktarıldığı bir şablon biçimi kullanırsınız. Örnekler şöyle olabilir:

```
vector<int> aVect;          //bir tamsayilar vektoru tanimla
```

veya


```
list<airtime> departure_list;    //bir "airtime" listesi tanimla
```

Dikkat ederseniz, STL konteynerlerinin boyutunu açıkça belirtmenize gerek yoktur. Konteynerler tüm bellek ayırma işlemlerini kendileri kontrol ederler.

Birleşik Konteynerler

Birleşik konteynerler sıralı değildir; bunun yerine bu tip konteynerler verilere erişmek için bir *anahtar* kullanır. Genellikle bir sayı veya karakter katarı olan anahtarlar, saklanan elemanları belirli bir sırada tutmak için konteyner tarafından otomatik olarak kullanılırlar. Bu tıpkı sıradan bir İngilizce sözlüğe benzer. Verilere, alfabetik sırayla düzenlenmiş kelimelere bakarak erişirsiniz. Bir anahtar değer ile başlarsınız (sözlük örneğini kullanmak amacıyla diyelim ki *aardvark* kelimesi ile başlıyorsunuz). Konteyner bu anahtarı, elemanın bellekteki konumuna dönüştürür. Eğer anahtarı biliyorsanız, ilgili değere çabucak erişebilirsiniz.

STL'de iki tür birleşik konteyner mevcuttur: *kümeler* (*set*) ve *eşlemeler* (*map*). Bunların her ikisi de, hızlı arama, ekleme ve çıkarma özelliğine sahip *ağaç* (tree) denilen bir yapı içinde verileri saklarlar. Kümeler ve eşlemeler, bu nedenle, çok çeşitli uygulamalar için uygun çok yönlü genel veri yapılarıdır. Ancak, bunları sıralamak ve rasgele (random) erişim gerektiren diğer işlemleri gerçekleştirmek verimli bir yol değildir.

Kümeler, eşlemelere kıyasla daha basittir ve daha yaygın olarak kullanılırlar. Bir küme, *anahtarlar* içeren bir takım öğeler saklar. Anahtarlar bu öğeleri sıralamak için kullanılan niteliklerdir. Örneğin, bir küme, isim nitelikleri anahtar olarak kullanılarak alfabetik olarak sıralanmış `person` sınıfının nesnelerini saklayabilir. Bu durumda, istenilen `person` nesnesini, belirtilen isme sahip nesneyi arayarak kolaylıkla saptayabilirsiniz. Eğer bir küme temel tipte (mesela, `int`) değerleri saklıyorsa anahtar, saklanan öğenin bütünüdür. Bazı yazarlar bir küme içinde saklanan nesnenin tümünden *anahtar* olarak söz ediyorlar, fakat biz, anahtarı sıralamak için kullanılan niteliğin öğenin tümü olması gerekmediğini vurgulamak için bunu *anahtar nesnesi* olarak adlandıracağız.

Bir eşleme, nesne çiftlerini saklar: Bir anahtar nesnesi ve bir değer nesnesi. Bir eşleme, genellikle diziye benzer bir konteyner gibi kullanılır. Aradaki fark, eşlemenin elemanlarına indeks numaraları ile erişmek yerine rasgele seçilmiş bir tipte olabilen indekslerle erişilmesidir. Yani, anahtar nesnesi indeks olarak görev yapar; değer nesnesi ise bu indeksteki değerdir.

Map ve *set* konteynerleri, belirli bir değere ait sadece bir anahtarın saklanmasına imkan verirler. Bunları, diyelim ki, tek personel numarasına göre düzenlenen çalışanlar listesinde kullanmak çok mantıklıdır. Öte yandan, *multimap* ve *multiset* konteynerleri birden fazla anahtara imkan verirler. Örneğin, bir İngilizce sözlükte "set" kelimesinin birkaç karşılığı olabilir.

Tablo 15.2'de, STL'de mevcut olan birleşik konteynerler özetleniyor.

TABLO 15.2: Temel Birleşik Konteynerler

Konteyner	Özellik
set	Sadece anahtar nesneleri saklar
	Her değer için yalnızca tek anahtara izin verilir
multiset	Sadece anahtar nesneleri saklar
	Birden fazla anahtar değerine izin verilir

TABLO 15.2: Temel Birleşik Konteynerler

Konteyner	Özellik
map	Anahtar nesnesini değer nesnesi ile ilişkilendirir
	Her değer için yalnızca tek anahtara izin verilir
multimap	Anahtar nesnesini değer nesnesi ile ilişkilendirir
	Birden fazla anahtar değerine izin verilir

Birleşik konteynerleri tanımlamak tıpkı sekans konteynerleri tanımlamaya benzer:

```
set<int>intSet;              //tamsayilar kumesi tanimlar
```

veya

```
multiset<employee>machinists; //personelle ilgili bir coklu kume tanimlar
```

Üye Fonksiyonlar

Sıralama ve arama gibi karmaşık işlemleri yürüten algoritmalar, STL'in ağır toplarıdır. Ancak, konteynerler, belirli bir tipteki konteynere özgü daha basit görevleri gerçekleştirmek için ayrıca üye fonksiyonlara da gereksinim duyarlar.

Tablo 15.3, isimleri ve amaçları birçok konteyner sınıfı için ortak olan ve sıkça kullanılan bazı üye fonksiyonları listeliyor.

TABLO 15.3:Tüm Konteynerlerde Ortak Olan Bazı Üye Fonksiyonlar

İsim	Amaç
`size()`	Konteyner içindeki öğelerin sayısını döndürür
`empty()`	Konteyner boşsa, true döndürür
`max_size()`	Olası en büyük konteynerin boyutunu döndürür
`begin()`	Konteynerin başlangıcına işaret eden bir iteratör döndürür; konteyner içinde ileriye doğru iterasyonla ilerlemek için kullanılır
`end()`	Konteynerin son konumunu geçen yeri gösteren bir iteratör döndürür; ileriye yönelik iterasyonu durdurmak için kullanılır
`rbegin()`	Konteynerin sonuna işaret eden bir ters iteratör döndürür; konteyner içinde geriye doğru iterasyon yapmak için kullanılır
`rend()`	Konteynerin başlangıcına işaret eden bir iteratör döndürür; geriye doğru iterasyonu durdurmak için kullanılır

Diğer birçok üye fonksiyon sadece belirli konteynerlerde veya belirli konteyner kategorilerinde karşınıza çıkar. Konu içinde ilerledikçe bunlar hakkında daha fazla bilgi sahibi olacaksınız. "STL Algoritmaları ve Üye Fonksiyonları" adlı Ek F'de, STL üye fonksiyonlarını ve hangi konteyner için hangi üye fonksiyonun mevcut olduğunu gösteren bir tablo yer alıyor.

Konteyner Adaptörleri

Konteyner adaptörleri denilen bir mekanizma kullanarak önceden bahsedilen normal fonksiyonlardan özel amaçlı konteynerler tanımlanması mümkündür. Bu özel amaçlı konteynerler, daha genel konteynerlerden daha basit bir arayüze sahiptirler. STL'de konteyner adaptörleri ile gerçeklenen özel konteynerler *yığın*, *kuyruk* ve *öncelik kuyruklarıdır*. Önceden vurguladığımız gibi, yığın, yığının tepesine veri öğesi yerleştirmek veya tepesinden bir veri öğesi almak için erişimleri kısıtlar. Kuyrukta, öğeleri kuyruğun bir ucundan ekleyip diğer ucundan çıkarırsınız. Öncelik kuyruklarında verileri önden rasgele sırayla eklersiniz; fakat verileri diğer uçtan alırken her zaman saklı olan *en büyük öğeyi* alırsınız: Öncelik kuyruğu, verileri sizin için otomatik olarak sıralar.

Yığın, kuyruk ve öncelik kuyrukları, farklı sekans konteynerlerinden oluşturulabilirler. Gerçi, bu işlem için genellikle çift uçlu kuyruklar kullanılır. Tablo 15.4, soyut veri tiplerini ve bunların gerçeklenmesinde kullanılabilecek sekans konteynerlerini gösteriyor.

TABLO 15.4:Adaptör Tabanlı Konteynerler

Konteyner	Gerçeklenme Yöntemi	Özellik
yığın	Vektör, liste veya çift uçlu kuyruk olarak gerçeklenebilir	Sadece tek uçtan ekleme ve çıkarma
kuyruk	Liste veya çift uçlu kuyruk olarak gerçeklenebilir	Bir uçtan ekleme ve diğer uçtan çıkarma
öncelik kuyruğu	Vektör veya çift uçlu kuyruk olarak gerçeklenebilir	Bir uçtan rasgele sırayla ekleme, diğer uçtan sıralı halde çıkarma

Bu sınıfları ilk kullanıma hazırlamak için şablon içinde şablon kullanılır. Örneğin, aşağıda çift uçlu kuyruk sınıfından örneklenen `int` tipini tutan bir yığın nesnesi yer alıyor:

```
stack< deque<int> > aStak;
```

Bu biçimlendirme ile dikkat etmeniz gereken bir ayrıntı, iki adet açılı kapanış parantezinin arasına boşluk eklemeniz gerektiğidir. Şu şekilde yazamazsınız:

```
stack<deque<int>> astak;      //hatali soz dizimi
```

Çünkü derleyici, >> karakter çiftini bir operatör olarak yorumlayacaktır.

Algoritmalar

Bir algoritma, bir konteyner (veya konteynerler) içindeki öğelere bir şeyler yapan bir fonksiyondur. Önceden vurguladığımız gibi, STL'deki algoritmalar, önceki konteyner kütüphanelerinde olduğu gibi konteyner sınıflarının üye fonksiyonu değildir, hatta `friend`'i bile değildir; bu algoritmaların tümü kendi başına bir şablon fonksiyonudur. Algoritmaları standart C++ dizileriyle veya kendi kendinize tasarladığınız konteyner sınıfları ile birlikte (sınıfın belirli temel fonksiyonları içermesi şartıyla) kullanabilirsiniz.

Tablo 15.5'te birkaç tane tipik algoritma gösterilmiştir. Diğerlerini konu içinde ilerledikçe inceleyeceğiz. Ek F'de pek çok STL algoritmasını listeleyen bir tablo yer alıyor.

TABLO 15.5:Bazı Tipik STL Algoritmaları

Algoritma	Amaç
`find`	Belirli bir değere eşdeğer ilk elemanı döndürür
`count`	Belirli değere sahip olan elemanların sayısını sayar
`equal`	İki konteynerin içeriğini karşılaştırır; karşılık gelen tüm değeler eşitse true döndürür
`search`	Bir konteyner içindeki değerler sekansını karşılık gelen sekansın aynını bir başka konteyner içinde arar
`copy`	Bir konteynerden diğerine bir değerler sekansını kopyalar
`swap`	Bir konumdaki değeri bir başka konumdaki ile değiştirir
`iter_swap`	Bir konumdaki değerleri bir başka konumdaki bir değerler sekansıyla değiştirir
`fill`	Bir değeri bir konum sekansına kopyalar
`sort`	Bir konteyner içindeki değerleri belirtilen bir sıraya göre sıralar
`merge`	Sıralı iki eleman aralığını, daha büyük sıralı bir aralık elde etmek için birleştirir
`accumulate`	Belirli bir aralıktaki elemanların toplamını döndürür
`for_each`	Konteyner içindeki her eleman için açıkça belirtilen bir fonksiyonu çalıştırır

Varsayalım, `int` tipinde, içinde veriler de olacak şekilde bir dizi tanımlıyorsunuz:

```
int arr[8] = {42, 31, 7, 80, 2, 26, 19, 75};
```

Sonra bu diziyi sıralamak için STL `sort()` algoritmasını şu şekilde yazarak kullanabilirsiniz:

```
sort(arr, arr+8);
```

Burada `arr`, dizinin başlangıç adresi; `arr+8` ise en sondan bir sonraki adrestir (dizinin en sonundan bir öğe uzaktaki adrestir).

İteratörler

İteratörler, bir konteyner içindeki veri öğelerine (bunlar genellikle *eleman* olarak adlandırılır) tek tek erişmek için kullanılan işaretçiye benzer unsurlardır. İteratörler genellikle elemandan elemana bir sıra dahilinde ilerlerler; bu işlem konteyner boyunca *iterasyon yapmak* olarak adlandırılır. İteratörleri `++` operatörü ile artırabilirsiniz; böylece, bir sonraki elemanı göstermelerini sağlamış olursunuz. İteratörlerin işaret ettiği elemanın değerini elde etmek için iteratörlere `*` operatörü ile dereferans yapabilirsiniz. STL'de bir iteratör, iteratör sınıfının bir nesnesi ile simgelenir.

Farklı konteyner sınıfları, farklı konteyner tipleri ile kullanılmalıdır. İteratörlerin başlıca üç sınıfı mevcuttur: ileri, iki yönlü ve rasgele erişim. *İleri iteratör*, konteyner içinde her seferinde bir öğe ilerleyerek sadece ileri doğru hareket edebilir. İteratörün `++` operatörü bunu gerçekleştirir. Geriye doğru hareket edemez ve konteynerin ortalarındaki rasgele bir konuma gitmesi de ayarlanamaz. *İki yönlü bir iteratör* ileri hareket edebildiği gibi geriye doğru da gidebilir; yani, iki yönlü iteratör için hem `++` hem de `--` operatörleri tanımlıdır. *Rasgele erişim iteratörü*, geri ve ileri gidebildiği gibi, buna ek olarak rasgele bir konuma da zıplayabilir. Bu iteratöre, örneğin, "27. konuma git" diyebilirsiniz.

Ayrıca özelleştirilmiş türde iki iteratör daha mevcuttur: *Giriş iteratörü*, ardarda gelen veri öğelerini bir konteynerin içine okumak için bir girdi aygıtına (`cin` veya bir dosya) "işaret" edebilir; bir *çıkış iteratörü* da bir çıktı aygıtına (`cout` veya bir dosya) "işaret" edebilir ve bir konteynerdeki bilgileri aygıta yazabilir.

İleri, iki yönlü ve rasgele erişim iteratörleri saklanabiliyor olmalarına rağmen giriş ve çıkış iteratörlerinin değerleri saklanamaz. Bu mantıklıdır: İlk üç operatör bellekteki konumlara işaret eder. Oysa giriş ve çıkış iteratörleri, saklı "işaretçi" değerinin hiçbir anlamı olmayan I/O aygıtlarına işaret ederler. Bu farklı türde iteratörlerin özelikleri Tablo 15.6'da gösterilmiştir.

TABLO 15.6: İteratörlerin Özellikleri

İteratör Tipi	Okur/Yazar	İteratör Kaydedilebilir	Yön	Erişim
Rasgele erişim	Okur ve yazar	Evet	İleri ve geri	Rasgele
İki yönlü	Okur ve yazar	Evet	İleri ve geri	Doğrusal
İleri	Okur ve yazar	Evet	Sadece ileri	Doğrusal
Çıkış	Sadece yazar	Hayır	Sadece ileri	Doğrusal
Giriş	Sadece okur	Hayır	Sadece ileri	Doğrusal

STL ile Potansiyel Problemler

STL'in şablon sınıflarının karmaşıklığı, derleyiciler üzerinde baskı kurar; derleyicilerin tümü iyi tepki vermez. Şimdi, bazı potansiyel problemlere göz atalım.

Öncelikle, hataları bulmak kimi zaman güç olur,çünkü hatalar aslında sınıfı kullanan kodun içindeyken, derleyici bu hataları başlık dosyasının derinlerindeymiş gibi rapor edebilir. Suçluyu bulmak için her seferinde kodun tek satırını açıklama şeklinde koddan çıkarmak gibi kaba kuvvete dayalı yöntemlere başvurmak zorunda kalabilirsiniz.

Başlık dosyalarının önceden derlenmesi, bu özelliği destekleyen derleyicilerde derleme süresini heyecanlandırıcı ölçüde hızlandırmasına rağmen STL ile ilgili problemlere neden olabilir. Eğer işler çalışmıyor gibi görünüyorsa, önceden derlenmiş başlıkları kapatmayı deneyin.

STL, sahte derleyici uyarıları üretebilir. Bunların en gözdesi "Conversion may lose significant digits" ("Dönüşüm, anlamlı basamakların kaybolmasına neden olabilir") uyarısıdır. Bu uyarılar zararsız görünür; dikkate alınmayabilir ve kapatılabilirler.

Bu küçük şikayetler bir yana, STL şaşırtıcı ölçüde dengeli ve çok yönlü bir sistemdir. STL sayesinde, hatalar sadece programın çalışması sırasında değil de derleme sürecinde de yakalanabilir. Farklı algoritmalar ve konteynerler çok tutarlı bir arayüz sergilerler; bir konteyner veya algoritmayla çalışan (uygun olarak kullanıldığını varsayarsak) genellikle bir başkasıyla da çalışır.

Bu hızlı özet muhtemelen sizi yanıtlardan daha çok sorularla baş başa bırakır. Bu bölümün kalan kısmı, işleri biraz daha netleştirmek için STL'in işleyişi hakkında yeterince spesifik ayrıntıları size sunacaktır.

Algoritmalar

STL algoritmaları işlemlerini toplanmış veriler üzerinde gerçekleştirirler. Bu algoritmalar STL konteynerleriyle çalışacak şekilde tasarlanmıştır, fakat bunların hoş yanlarından biri, bu algoritmaları sıradan C++ dizilerine de uygulayabiliyor olmanızdır. Bu, dizilerle çalışırken sizi hatırı

sayılır bir yükten kurtarabilir. Bunlar ayrıca, algoritmaları öğrenmek için kolay bir yöntem de sunarlar (konteynerlerle karmaşık bir hale getirmeksizin). Bu bölümde bazı tipik algoritmaların nasıl kullanıldığını inceleyeceğiz. (Algoritmaların Ek F'de listelendiğini hatırınızdan çıkarmayın.)

find() Algoritması

`find()` algoritması, bir konteyner içinde, belirtilen değere sahip ilk elemanı arar. `FIND` örnek programı, bir değeri bir tamsayılar dizisinde bulmaya çalıştığımızda bunun nasıl gerçekleştirebileceğimizi gösteriyor.

```
// find.cpp
// belirtilen degere sahip ilk nesneyi bulur
#include <iostream>
#include <algorithm>                    //find() icin
using namespace std;

int arr[] = { 11, 22, 33, 44, 55, 66, 77, 88 };

int main()
   {
   int* ptr;
   ptr = find(arr, arr+8,33);           //ilk 33'u bul
   cout << "First object with value 33 found at offset "
        << (ptr-arr)<< endl;
   return 0;
   }
```

Bu programın çıktısı şöyledir:

```
First object with value 33 found at offset 2.
```

Her zamanki gibi, dizinin ilk elemanı 0 olarak numaralandırılır. Bu nedenle, 33 sayısı 3. ötelemede değil, 2. ötelemede bulunur.

Başlık Dosyaları

Bu programa `ALGORITHM` başlık dosyasını dahil ettik. Dikkat ederseniz, Standart C++ kütüphanesindeki diğer başlık dosyaları gibi, burada da dosya uzantısı (`.H` gibi) kullanmadık. Bu dosya STL algoritmalarının deklarasyonlarını içerir. Diğer başlık dosyaları konteynerler ve diğer amaçlar için kullanılır. Eğer STL'in eski bir versiyonunu kullanıyorsanız, başlık dosyalarınızı biraz farklı bir isimle (mesela `ALGO.H` gibi) dahil etmeniz gerekebilir.

Aralıklar

`find()`'ın ilk iki parametresi, incelenecek elemanların aralıklarını belirtir. Bu değerler iteratörler tarafından belirtilir. Bu örnekte, iteratörlerin özel bir hali olan normal C++ işaretçi değerleri kullanılmıştır.

İlk parametre, incelenecek ilk değerin iteratörüdür (veya bu örnekte, işaretçisidir). İkinci parametre, incelenecek son elemanın bir sonrasında yer alan konumun iteratörüdür.

Burada 8 eleman olduğu için söz konusu olan değer, ilk değer artı 8'dir. Buna *sonun ötesindeki* değer denir; incelenecek olan aralığın hemen sonrasındaki elemana işaret eder.

Kullandığımız söz dizimi, `for` döngülerdeki normal C++ ifadelerini hatırlatmaktadır:

```
for(int j=0; j<8; j++)                  // 0'dan 7'ye
   {
   if(arr[j] == 33)
      {
      cout << "First object with value 33 found at offset "
           << j << endl;
      break;
      }
   }
```

`FIND` örneğinde, `find()` algoritması, sizi bu döngüyü yazma zahmetinden kurtarır. Daha karmaşık durumlarda algoritmalar çok daha karmaşık kodlar yazmaktan sizi kurtarabilir.

count() Algoritması

Şimdi de bir konteynerde kaç elemanın belli bir değere sahip olduğunu sayan ve bu sayıyı döndüren `count()` adlı başka bir algoritmaya bakalım. Algoritmamızı `COUNT` örneğinde uyguluyoruz:

```
// count.cpp
// belirli bir degere sahip elemanlari sayar
#include <iostream>
#include <algorithm>
using namespace std;                    //count() icin

int arr[] = { 33, 22, 33, 44, 33, 55, 66, 77 };

int main()
   {
   int n = count(arr, arr+8, 33);       //33'lerin sayisini bul
   cout << "There are " << n << " 33's in arr." << endl;
   return 0;
   }
```

Programın çıktısı şöyle olacaktır:

```
There are 3 33's in arr.
```

sort() Algoritması

`sort()` algoritmasının işlevini tahmin edebilirsiniz. Bu algoritmayı da `SORT` adını verdiğimiz aşağıdaki programda bir diziye uyguluyoruz.

```
// sort.cpp
// bir tamsayi dizisini siralar
#include <iostream>
#include <algorithm>
using namespace std;
                                        //sayi dizisi
int arr[] = { 45, 2, 22, -17, 0, -30, 25, 55 };

int main()
   {
```

```
    sort(arr, arr+8);           //sayilari sirala
    for(int j=0; j<8; j++)      //sirali diziyi goster
       cout << arr[j] << ' ';
    cout << endl;
    return 0;
    }
```

Program, şu çıktıyı verir:

```
-30, -17, 0, 2, 22, 25, 45, 55
```

Bu algoritmanın başka varyasyonlarına daha sonra döneceğiz.

search() Algoritması

Bazı algoritmalar aynı anda iki konteyner üzerinde işlem yapar. Örneğin, **find()** algoritması tek bir konteynerin içinde belli bir değeri ararken, **search()** algoritması bir başka konteynerin içindeki bir konteyner tarafından belirlenen bir değer sekansını arar. Bunu aşağıdaki **SEARCH** örneğinde görüyoruz:

```
// search.cpp
// Bir konteynerdeki deger sekansini bir baska konteyner icinde arar
#include <iostream>
#include <algorithm>
using namespace std;

int source[] = { 11, 44, 33, 11, 22, 33, 11, 22, 44 };
int pattern[] = { 11, 22, 33 };

int main()
   {
   int* ptr;
   ptr = search(source, source+9, pattern, pattern+3);
   if(ptr == source+9)                    //sonu gecmissek
      cout << "No match found\n";
   else
      cout << "Match at " << (ptr - source)<< endl;
   return 0;
   }
```

Algoritma, dizi kaynağı içinde dizi modeli (pattern) tarafından belirlenen 11, 22, 33 sayı sekansını arar. İnceleyerek görebileceğiniz gibi bu sekans, kaynağın dördüncü elemanında (sıfırdan saymaya başladığımız için bu eleman 3'tür) bulunur. Programın çıktısı şu şekildedir:

```
Match at 3
```

Eğer iteratör değeri **ptr**, kaynağın sonundan bir sonraki konuma gelirse, aradığımızı bulamadık demektir.

search() gibi algoritmaların argümanlarının aynı tip konteyner olması gerekmez. Örneğin, kaynak bir STL vektöründe, model (pattern) de bir dizide yer alabilir. Böylesi bir genelleme yeteneği STL'in çok güçlü bir özelliğini oluşturur.

merge() Algoritması

Şimdi de üç konteynerle çalışan, iki kaynak konteynerinden aldığı elemanları hedef konteyner üzerinde birleştiren bir algoritmaya bakalım. Algoritmayı, aşağıdaki **MERGE** örneğinde izliyoruz:

```
// merge.cpp
// iki konteyneri alip ucuncusu uzerinde birlestirir
#include <iostream>
#include <algorithm>
using namespace std;                              //merge() icin

int src1[] = { 2, 3, 4, 6, 8 };
int src2[] = { 1, 3, 5 };
int dest[8];

int main()
   {
   merge(src1, src1+5, src2, src2+3, dest);       //src1 ve src2'yi
   for(int j=0; j<8; j++)                         //birlestirip dest'e yaz
      cout << dest[j] << ' ';                     //dest'i goster
   cout << endl;
   return 0;
   }
```

Hedef konteynerin içeriğini gösteren çıktı şu şekilde olacaktır:

```
1 2 3 3 4 5 6 8
```

Gördüğünüz gibi, birleştirme işlemi sırayı bozmaz; bir elemanı birinci konteynerden, diğerini ikinciden alarak çalışır.

Fonksiyon Nesneleri

Bazı algoritmalar *fonksiyon nesnelerini* argüman olarak alabilirler. Bir fonksiyon nesnesi, kullanıcı gözünde bir şablon fonksiyonuna çok benzer. Ancak aslında aşırı yüklenmiş () operatöründen ibaret tek bir üye fonksiyonu olan bir şablon sınıfının nesnesidir. Bu ifade biraz esrarengiz gelmiş olabilir ama kavramın kullanımı kolaydır.

Diyelim ki, bir sayı dizisini artan değil de azalan sıraya sokmak istiyorsunuz. **SORTEMP** programı bunun nasıl yapıldığını gösterir:

```
// sortemp.cpp
// double dizisini azalan siraya sokar
// greater<>() fonksiyon nesnesini kullanir
#include <iostream>
#include <algorithm>                      //sort() icin
#include <functional>                     //greater<> icin
using namespace std;
                                          //double dizisi
double fdata[] = { 19.2, 87.4, 33.6, 55.0, 11.5, 42.2 };

int main()
                                          //double'lari sirala
   {
   sort( fdata, fdata+6, greater<double>() );

   for(int j=0; j<6; j++)                 //sirali double'lari goster
```

```
        cout << fdata[j] << ' ';
    cout << endl;
    return 0;
    }
```

`sort()` algoritması genelde sayıları artan sıraya sokar. Ama `greater<>()` fonksiyon nesnesi, yani `sort()`'un üçüncü argümanı kullanılarak, sıralama düzeni ters çevrilir. Programın çıktısı şöyle olacaktır:

```
87.4 55 42.2 33.6 19.2 11.5
```

Karşılaştırmaların yanında aritmetik ve mantıksal işlemler için de fonksiyon nesneleri vardır. Fonksiyon nesnelerine bu bölümün son kısmında daha yakından eğileceğiz.

Fonksiyon Nesneleri Yerine Kullanıcı Tarafından Yazılan Fonksiyonlar

Fonksiyon nesneleri sadece temel C++ tipleri ve ilgili operatörlerin (+, <,== vs) tanımlı olduğu sınıflar üzerinde işlem görürler. Eğer böyle bir durumun söz konusu olmadığı değerler üzerinde çalışıyorsanız, kullanıcı tarafından tanımlanan bir fonksiyonu bir fonksiyon nesnesinin yerine yerleştirebilirsiniz. Örneğin, < operatörü sıradan `char*` karakter katarları için tanımlı değildir, fakat bu karşılaştırmayı gerçekleştiren bir fonksiyon yazabiliriz ve bu fonksiyonun adresini (ismini) fonksiyon nesnesinin yerine kullanabiliriz. `SORTCOM` örneği, `char*` karakter kataı dizilerinin nasıl sıralanabileceğini gösteriyor:

```
// sortcom.cpp
// kullanici tarafindan yazilan bir karsilastirma fonksiyonu
// karakter katari dizilerini siralar
#include <iostream>
#include <string>                       //strcmp() icin
#include <algorithm>
using namespace std;
                                        //karakter katari dizileri
char* names[] = { "George", "Penny", "Estelle",
                  "Don", "Mike", "Bob" };

bool alpha_comp(char*, char*);          //deklarasyon

int main()
   {
   sort(names, names+6, alpha_comp);    //karakter katarlarini sirala

   for(int j=0; j<6; j++)               //sirali katarlari ekranda goster
      cout << names[j] << endl;
   return 0;
   }

bool alpha_comp(char* s1, char* s2)     //eger s1<s2 ise, true dondur
   {
   return ( strcmp(s1, s2)<0 )? true : false;
   }
```

`sort()` algoritmasının üçüncü argümanı, `alpha_comp()` fonksiyonun adresidir. `alpha_comp()` iki `char*` karakter katarını karşılaştırır ve ilki leksikografik olarak (yani, alfabetik

sıraya göre) diğerinden önce geliyorsa true, diğer durumlarda false değeri döndürür. Burada C kütüphane fonksiyonu `strcmp()` kullanılmıştır. `strcmp()`, ilk argüman ikinciden küçükse sıfırdan küçük bir değer döndürür. Programın çıktısı, tahmin edebileceğiniz gibidir:

```
Bob
Don
Estelle
George
Mike
Penny
```

Aslında metin üzerinde işlem yapabilmek için kendi fonksiyon nesnelerinizi yazmanız gerekmez. Standart kütüphanedeki karakter kataı sınıfını kullanırsanız, kütüphanede tanımlı `less<>()` ve `greater<>()` gibi fonksiyon nesnelerinden yararlanabilirsiniz.

Algoritmalara _if Eklemek

Bazı algoritmaların `_if` ile biten versiyonları vardır. Bu algoritmalar, *önerme* (*predicate*) adı verilen ek bir parametre alırlar. Söz konusu ilave parametre, bir fonksiyon nesnesi ya da fonksiyondur. Örneğin, `find()` algoritması belli bir değere eşit olan tüm elemanları bulur. Arzu ettiğimiz başka bir özelliğe sahip nesneleri bulmak için `find_if()` algoritması ile çalışan bir fonksiyon da yazabiliriz.

Örneğimizde karakter kataı nesneleri kullanılmaktadır. `find_if()` algoritmasına, kullanıcı tarafından yazılmış bir `isDon()` fonksiyonu sağlıyoruz. Bu fonksiyon, karakter kataı nesnelerinden oluşan bir dizinin elemanının "Don" değerine eşit olup olmadığını kontrol eder. `FIND_IF`'in kaynak koduna bakalım:

```
// find_if.cpp
// karakter katari dizisinde "Don"a esit olan ilk ismi arar
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
//-------------------------------------------------------------
bool isDon(string name)             //name=="Don" ise true degeri dondur
   {
   return name=="Don";
   }
//-------------------------------------------------------------
string names[] = { "George", "Estelle", "Don", "Mike", "Bob" };

int main()
   {
   string* ptr;
   ptr = find_if( names, names+5, isDon);

   if(ptr==names+5)
      cout << "Don is not on the list.\n";
   else
      cout << "Don is element "
           << (ptr-names)
           << " on the list.\n";
   return 0;
   }
```

Dizideki isimlerden biri gerçekten de "Don" olduğundan, programın çıktısı şu şekilde gerçekleşecektir:

```
Don is element 2 on the list.
```

`isDon()` fonksiyonunun adresi, `find_if()`'e geçilen üçüncü argümandır. İlk ve ikinci argümanlar ise, her zaman olduğu gibi, dizinin ilk ve "sondan bir sonraki" adresleridir.

Burada `find_if()` algoritması, `isDon()` fonksiyonunu, aralık içindeki tüm elemanlara uygular. Şayet `isDon()` herhangi bir eleman için true değeri döndürüyorsa, o zaman `find_if()` de söz konusu elemanın işaretçisinin (iteratörünün) değerini döndürür. `isDon()` false döndürüyorsa, `find_if()` de dizinin sondan bir sonraki adresine bir işaretçi döndürecektir.

`count()`, `replace()` ve `remove()` gibi muhtelif diğer algoritmaların da `_if` versiyonları vardır.

for_each() Algoritması

`for_each()` algoritması bir konteyner içindeki her öğe için bir şeyler yapmanıza imkan verir. Bu "bir şeyler"in ne olduğunu saptamak için kendi fonksiyonunuzu yazarsınız. Fonksiyonunuz konteyner içindeki elemanları değiştirmez ama, bu elemanların değerlerini kullanabilir veya ekranda gösterebilir.

Şimdi, bir dizi içindeki tüm değerleri inç'ten santimetreye çevirmek ve ekranda göstermek için `for_each()`'in kullanıldığı bir örnek verelim. Bir değeri 2.54 ile çarpan `in_to_cm()` adında bir fonksiyon yazıyoruz ve bu fonksiyonun adresini `for_each()`'e üçüncü argüman olarak kullanalım. `FOR_EACH` için program listesi şöyledir:

```
// for_each.cpp
// inc degerindeki dizi elemanlarinin cm olarak ciktisi almak icin
// for_each() kullanir
#include <iostream>
#include <algorithm>
using namespace std;

void in_to_cm(double);          //deklarasyon

int main()
   {                            //inc degerleri dizisi
   double inches [] = { 3.5, 6.2, 1.0, 12.75, 4.33 };
                                //cm olarak cikti
   for_each(inches, inches+5, in_to_cm);
      cout << endl;
   return 0;
   }
void in_to_cm(double in)        //cm'ye cevirip cm olarak ekranda gosterir
   {
   cout << (in * 2.54) << ' ';
   }
```

Çıktı şu şekilde görünür:

```
8.89 15.748 2.54 32.385 10.9982
```


transform() Algoritması

`transform()` algoritması, bir konteyner içindeki öğelerin tümü üzerinde bir işlem gerçekleştirir ve elde edilen sonuç değerleri bir başka konteyner içine (veya aynı konteynere) yerleştirir. Yine, kullanıcı tarafından yazılan bir fonksiyon, öğelerin her biri üzerinde gerçekleştirilecek işlemleri saptar. Bu fonksiyonun döndürdüğü tip, hedef konteynerinki ile aynı olmalıdır. Örneğimiz `FOR_EACH`'e benzer; ancak, şimdiki `in_to_cm()` fonksiyonumuz dönüştürülmüş değerleri ekranda göstermek yerine, santimetre değerlerini farklı bir diziye, `centi[]`'ye yerleştirir. Ana program sonra, `centi[]`'nin içeriğini ekranda gösterir. `TRANSFO` programının listesi şöyledir:

```
// transfo.cpp
// inc degerlerinin dizisini cm'ye cevirmek icin transform()'u kullanir
#include <iostream>
#include <algorithm>
using namespace std;

int main()
   {                                //inc degerleri dizisi
   double inches[] = { 3.5, 6.2, 1.0, 12.75, 4.33 };
   double centi[5];
   double in_to_cm(double);         //prototip
                                    //centi[] dizisine aktar
   transform(inches, inches+5, centi, in_to_cm);

   for(int j=0; j<5; j++)           //centi[] dizisini ekranda goster
      cout << centi[j] << ' ';
   cout << endl;
   return 0;
   }

double in_to_cm(double in)          //incleri cm'ye cevir
   {
   return (in * 2.54);              //sonucu dondur
   }
```

Programın çıktısı, `FOR_EACH` programının çıktısı ile aynıdır.

STL'deki algoritmaların yalnızca birkaç tanesine göz gezdirdik. Birçok algoritma daha vardır, fakat burada gösterdiklerimiz mevcut algoritmaların çeşidi ve bu algoritmaların nasıl kullanılacağı hakkında bir fikir vermiş olmalıdır.

Sekans Konteynerleri

Önceden vurguladığımız gibi, STL'deki konteynerler başlıca iki kategoride toplanır: sekans konteynerleri ve birleşik konteynerler. Bu bölümde, üç sekans konteynerini (vektörler, listeler ve çift uçlu kuyruklar) ele alacağız; bu konteynerlerin nasıl çalıştığına ve bunların üye fonksiyonlarına odaklanacağız. İteratörleri henüz öğrenmediğimiz için bu konteynerler üzerinde gerçekleştiremeyeceğimiz bazı işlemler olacak. İteratörleri daha sonra inceleyeceğiz.

Buradaki programların her biri, konu edilen konteynere ait birkaç üye fonksiyon tanıtacaktır. Yine de, farklı çeşitteki konteynerlerin aynı isimde ve özellikte üye fonksiyonları kullandıklarını hatırınızdan çıkarmayın. Bu nedenle, diyelim ki, `push_back()` fonksiyonu hakkında vektörler konusunda öğrendikleriniz listelerle ve çift uçlu kuyruklarla da ilgili olacaktır.

Vektörler

Vektörleri akıllı diziler olarak düşünebilirsiniz. Vektörler, verileri eklerken veya çıkarırken vektör boyutunu genişletip, daraltarak bellekte yer ayırma işlemini sizin adınıza gerçekleştirirler. Vektör elemanlarına [] operatörü ile erişerek, vektörleri dizilere çok benzer şekilde kullanabilirsiniz. Bu tür rasgele erişimler vektörlerle çok hızlıdır. Vektörün sonuna (*arkasına*) yeni elemanlar eklemek de ayrıca hızlıdır. Bu tür bir durum karşısında, yeni veri öğesini tutmak için vektör boyutu otomatik olarak artırılır.

push_back(), size() ve operator [] Üye Fonksiyonları

İlk örneğimiz, **VECTOR**, en sık kullanılan vektör işlemlerini gösteriyor:

```
// vector.cpp
// push_back(),operator [],size()uye fonksiyonlari
#include <iostream>
#include <vector>
using namespace std;

int main()
   {
   vector<int> v;                    //bir tamsayilar vektoru tanimlar

   v.push_back(10);                  //degerleri dizinin sonuna yerlestir
   v.push_back(11);
   v.push_back(12);
   v.push_back(13);

   v[0] = 20;                        //yeni degerlerle degistir
   v[3] = 23;

   for(int j=0; j<v.size(); j++)     //vektor icerigini ekranda goster
      cout << v[j] << ' ';           //20 11 12 23
   cout << endl;
   return 0;
   }
```

v vektörünü tanımlamak için vektörün varsayılan kurucu fonksiyonu (argümansız) kullanılır. Tüm STL konteynerlerinde olduğu gibi, konteynerin tutacağı değişkenin tipini (bu örnekte, **int** tipi) belirtmek için şablon biçimi kullanılır. Konteynerin büyüklüğü açıkça belirtilmez; bu nedenle konteyner 0'dan başlar.

push_back() üye fonksiyonu, argümanının değerini vektörün arkasına ekler. (Vektörün arkası, en yüksek indeks numarasına sahip olan elemanın bulunduğu yerdir.) Vektörün önü (0 indeks numarasına sahip eleman), liste veya kuyruktakinden farklı olarak, yeni elemanlar eklemek için kullanılamaz. Bu örnekte 10, 11, 12 ve 13 değerleri eklenir; böylece, **v[0]**, 10 değerini; **v[1]**, 11 değerini; **v[2]**, 12 değerini ve **v[3]** ise 13 değerini içerir.

Vektör, veri içermeye başlar başlamaz, bu veriler tıpkı bir dizi içindeymiş gibi, aşırı yüklenmiş [] operatörü yardımıyla –okuma ve yazma için- erişilebilir hale gelir. Bu operatör, ilk değeri 10'dan 20'ye; son elemanı da 13'ten 23'e değiştirmek için kullanılır. **VECTOR** programının çıktısı şöyledir:

```
20 11 12 23
```


size() üye fonksiyonu, halihazırda konteynerde bulunan elemanların sayısını döndürür; **VECTOR** programında bu sayı, 4'tür. Bu değeri bir **for** döngüsü içinde kullanarak konteynerdeki elemanların değerlerinin dökümü alınır.

Bir başka üye fonksiyon olan **max_size()** (burada gösterilmemiştir), bir konteynerin genişleyebileceği maksimum boyutu döndürür. Bu boyut, konteynerde saklı olan verilerin tipine (elemanlar büyüdükçe daha az sayıda eleman saklayabilirsiniz), konteynerin tipine ve işletim sistemine bağlıdır. Örneğin, bizim sistemimizde **int** tipinde bir vektör için **max_size()** fonksiyonu 1,073,741,823 değerini döndürür.

swap(), empty(), back() ve pop_back() Üye Fonksiyonları

Sıradaki örnek olan **VECTCON**, vektörlerle ilgili birkaç ilave kurucu ve üye fonksiyonu gösterir.

```
// vectcon.cpp
// kurucu fonksiyonlar, swap(), empty(), back(), pop_back()
#include <iostream>
#include <vector>
using namespace std;

int main()
   {                          //double dizisi
   double arr[] = {1.1,2.2,3.3,4.4 };

   vector<double> v1(arr, arr+4);  //vektore, ilk deger olarak diziyi ata
   vector<double> v2(4);           //vektoru 4 elemanli buyukluge dusur

   v1.swap(v2);                    //v1 ve v2'nin iceriklerini takas yap

   while( !v2.empty() )            //vektor bosalana kadar,
      {
      cout << v2.back() << ' ';    //son elemani goster
      v2.pop_back();               //son elemani cikart
      }                            //cikti: 4.4 3.3 2.2 1.1
   cout << endl;
   return 0;
   }
```

Bu programda iki yeni vektör kurucu fonksiyonu kullanılmıştır. İlki, kendisine argüman olarak aktarılan normal bir C++ dizisinin değerlerini **v1** vektörüne başlangıç değeri olarak atar. Bu kurucu fonksiyonun argümanları, dizinin başına ve sonundan bir sonraki elemana işaret eden işaretçilerdir. İkinci kurucu fonksiyon **v2**'nin başlangıç boyutunu 4 olarak ayarlar, fakat ilk değerleri atamaz. Her iki vektör de **double** tipini tutar.

swap() üye fonksiyonu, bir vektördeki verilerin tümünü bir başkasındaki verilerle değiştirir. Bunu gerçekleştirirken elemanların sırasını korur. Bu programda, **v2** içinde yalnızca gereksiz veriler bulunur, bu nedenle **v2**, **v1**'in verileriyle takas yapılır. **v2** içinde artık **v1**'in değerleri olduğunu göstermek için **v2** görüntülenir. Çıktı şöyle görünür:

```
4.4, 3.3, 2.2, 1.1
```

back() üye fonksiyonu, vektör içindeki son elemanın değerini döndürür. Bu değer, **cout** ile ekranda gösterilir. **pop_back()** üye fonksiyonu, vektördeki son elemanı vektörden çıkarır.

Dolayısıyla, döngünün her tekrarlanışında farklı bir son eleman ile karşılaşılır. (`pop_back()`'in aynı anda son elemanı döndürüp vektörden çıkarmıyor oluşu biraz şaşırtıcıdır. Yığınların kullanıldığı önceki örneklerde `pop()` son elemanı döndürüyordu, fakat `pop_back()` döndürmez. Bu nedenle, ayrıca `back()` fonksiyonunun kullanılması gerekir.)

Birkaç üye fonksiyon ayrıca algoritma olarak da bulunur, mesela `swap()`. Böyle bir durum söz konusu olduğunda, genellikle üye fonksiyon versiyonu kullanıma sunulur, çünkü söz konusu konteyner için bu versiyon, algoritma versiyonundan çok daha randımanlıdır. Kimi zaman algoritma versiyonunu da kullanabilirsiniz. Örneğin, iki farklı konteynerdeki elemanları takas yapmak için algoritmayı kullanabilirsiniz.

insert() ve erase() Üye Fonksiyonları

`insert()` ve `erase()` üye fonksiyonları, bir konteyner içindeki rasgele bir konuma bir eleman ekler veya bu konumdan bir eleman çıkarırlar. Bu fonksiyonlar vektörlerle kullanmak için pek elverişli değildir; çünkü, ekleme sırasında yeni elemana yer açmak için, çıkarma sırasında da çıkarılan öğenin bulunduğu alanı kapatmak için tüm elemanların kaydırılması gerekir. Yine de, ekleme ve çıkarma, eğer hız söz konusu değilse, her şeye rağmen kullanışlı olabilir. Sıradaki örnek olan `VECTINS`, bu üye fonksiyonların nasıl kullanılacağını gösteriyor:

```
// vectins.cpp
// insert()ve erase()
#include <iostream>
#include <vector>
using namespace std;

int main()
   {
   int arr[] = { 100, 110, 120, 130 };     //tamsayilar dizisi

   vector<int> v(arr, arr+4);              //vektore ilk deger olarak bir diziyi atar

   cout << "\nBefore insertion: ";
   for(int j=0; j<v.size(); j++)           //elemanlarin tumunu goster
      cout << v[j] << ' ';

   v.insert( v.begin()+2, 115 );           //2. elemana 115 degerini ata

   cout << "\nAfter insertion: ";
   for(j=0; j<v.size(); j++)               //elemanlarin tumunu goster
      cout << v[j] << ' ';

   v.erase( v.begin()+2 );                 //2. elemani cikart

   cout << "\nAfter erasure: ";
   for(j=0; j<v.size(); j++)               //elemanlarin tumunu goster
      cout << v[j] << ' ';
   cout << endl;
   return 0;
   }
```

`insert()` üye fonksiyonu (en azından buradaki versiyonu) iki argüman alır: konteyner içinde elemanın ekleneceği yer ve elemanın değeri. Vektördeki 2. elemanı (üçüncü eleman) belirtmek için `begin()` üye fonksiyonuna 2 eklenir. Ekleme noktasından konteynerin sonuna kadar tüm elemanlar yer açmak için yukarıya doğru kaydırılır ve konteynerin büyüklüğü 1 artırılıyor.

`erase()` üye fonksiyonu belirtilen konumdaki elemanı çıkarır. Çıkartma noktasının aşağısında kalan elemanlar aşağıya doğru kaydırılır ve konteynerin boyutu 1 azaltılır. `VECTINS` programının çıktısı şöyledir:

```
Before insertion: 100 110 120 130
After insertion:  100 110 115 120 130
After erasure:    100 110 120 130
```

Listeler

Bir STL liste konteyneri, her elemanın sadece bir sonrakine işaret etmekle kalmayıp, aynı zamanda öncekine de işaret eden bir işaretçi içerdiği çift bağlantılı bir listedir. Konteyner, hem öndeki (ilk) hem de arkadaki (son) elemanların adreslerini saklayarak her iki uca da hızlı erişimde bulunur.

push_front(), front() ve pop_front() Üye Fonksiyonları

İlk örneğimiz olan `LIST`, verilerin hem önden hem arkadan nasıl yerleştirilebileceğini, nasıl okunabileceğini ve nasıl geri alınabileceğini gösteriyor.

```
// list.cpp
// push_front(),front(),pop_front() fonksiyonlari
#include <iostream>
#include <list>
using namespace std;

int main()
   {
   list<int> ilist;

   ilist.push_back(30);              //ogeleri arkadan yerlestir
   ilist.push_back(40);
   ilist.push_front(20);             //ogeleri onden yerlestir
   ilist.push_front(10);

   int size = ilist.size();          //ogelerin sayisi

   for(int j=0; j<size; j++)
      {
      cout << ilist.front() << ' '; //onden oge oku
      ilist.pop_front();             //onden oge al
      }
   cout << endl;
   return 0;
   }
```

Veriler listenin arkasına veya önüne öyle yerleştirilir ki, veriler ekranda gösterilip, sonra önden bir veri çıkarıldığında liste normal sırada kalır:

```
10 20 30 40
```

`push_front()`, `pop_front()` ve `front()` üye fonksiyonları, vektörlerde iş başında gördüğümüz `push_back()`, `pop_back()` ve `back()` fonksiyonlarına benzerler.

Dikkat ederseniz, liste elemanları için rasgele erişim kullanılamaz, çünkü bu tür bir erişim fazlasıyla yavaştır. Bu nedenle, [] operatörü listeler için tanımlanmamıştır. Tanımlanmış olsaydı, bu operatör liste üzerinden geçmesi, geçtikçe her elemanı sayması ve bu işlemi doğru elemana ulaşana kadar sürdürmesi gerekecekti. Bu, zaman alıcı bir işlemdir. Eğer rasgele erişime ihtiyacınız varsa vektör veya çift uçlu kuyruk kullanmalısınız.

Ekleme veya çıkarma işlemlerini listenin ortasından sıkça yapacaksanız, listeleri kullanmak uygundur. Bu, vektörler ve çift uçlu kuyruklar için randımanlı değildir, çünkü ekleme veya çıkarma noktasının üzerindeki tüm elemanların kaydırılması gerekir. Ancak, bu durum listelerle hızlıca gerçekleştirilir, çünkü yeni eleman eklemek veya çıkarmak için sadece birkaç işaretçinin değiştirilmesi gereklidir. (Her şeye rağmen, doğru ekleme noktasını bulmak hala zaman alıcı olabilir.)

`insert()` ve `erase()` üye fonksiyonları, listelere ekleme ve çıkarma yapmak için kullanılır, fakat bunlar, iteratörlerin kullanımını gerektirdiği için bu fonksiyonlarla ilgili tartışmayı erteleyeceğiz.

reverse(), merge() ve unique() Üye Fonksiyonları

Bazı üye fonksiyonlar sadece listeler için bulunur. Aynı işi gören algoritmalar mevcut olsa da, diğer konteynerler için bu tür üye fonksiyonlar tanımlanmaz. Sıradaki örneğimiz `LISTPLUS`, bu fonksiyonlardan bazılarını gösterir. Program, iki nesne listesini iki dizinin içeriği ile doldurarak işlemlere başlar.

```
// listplus.cpp
// reverse(),merge() ve unique()
#include <iostream>
#include <list>
using namespace std;

int main()
   {
   int j;
   list<int> list1, list2;

   int arr1[] = { 40, 30, 20, 10 };
   int arr2[] = { 15, 20, 25, 30, 35 };

   for(j=0; j<4; j++)
      list1.push_back( arr1[j] );      //list1: 40, 30, 20, 10
   for(j=0; j<5; j++)
      list2.push_back( arr2[j] );      //list2: 15, 20, 25, 30, 35

   list1.reverse();                    //list1'i ters cevir: 10 20 30 40
   list1.merge(list2);                 //list2'yi list1 ile birlestir
   list1.unique();                     //cift gorunen 20 ve 30'u cikart

   int size = list1.size();
   while( !list1.empty() )
      {
      cout << list1.front() << ' ';    //onden oge oku
      list1.pop_front();               //onden oge al
      }
   cout << endl;
   return 0;
   }
```


Birinci liste tersten sıralıdır; bu yüzden, `reverse()` üye fonksiyonu kullanılarak normal sıralı liste haline döndürülür. (Bir liste konteynerini tersine çevirmek kolaydır, çünkü liste konteynerinin her iki ucuna da erişilebilir.) Bu gereklidir, çünkü ikinci üye fonksiyon olan `merge()`, iki liste üzerinde işlem görür ve her ikisinin de sıralı olmasını gerektirir. Tersine çevirme işleminde sonra, iki liste şöyle görünür:

```
10, 20, 30, 40
15, 20, 25, 30, 35
```

Şimdi, `merge()` fonksiyonu, sıralı olan her şeyin sırasını koruyarak ve `list1`'i yeni öğeleri tutacak şekilde genişleterek `list2`'yi `list1` ile birleştirebilir. Sonuçta `list1`'in içeriği şöyle olur:

```
10, 15, 20, 20, 25, 30, 30, 35, 40
```

Son olarak, `list1`'e `unique()` üye fonksiyonu uygulanır. Bu fonksiyon aynı değere sahip komşu elemanları bulur ve ilki hariç tümünü listeden çıkartır. `list1`'in içeriği sonra ekranda gösterilir. `LISTPLUS` programının çıktısı şöyledir:

```
10, 15, 20, 25, 30, 35, 40
```

Listenin içeriğini göstermek için `front()` ve `pop_front()` üye fonksiyonları bir döngü içinde kullanılır. Elemanların her biri, önden arkaya doğru görüntülenir ve sonra listeden çıkartılır. Sonuçta, listeyi ekranda gösterme işlemi listeyi yok eder. İstediğiniz her zaman bu olmayabilir, fakat bu, listenin peş peşe gelen elemanlarına erişmek için şimdilik bildiğimiz tek yöntemdir. Bu problemi, bir sonraki bölümde ele alınan iteratörler çözecektir.

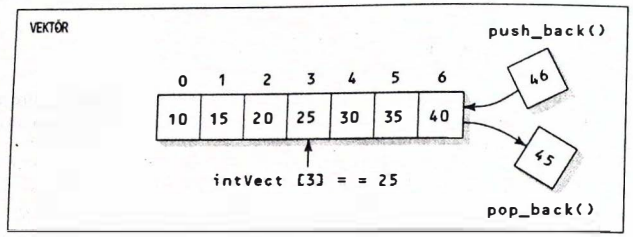
Çift Uçlu Kuyruklar (Deque)

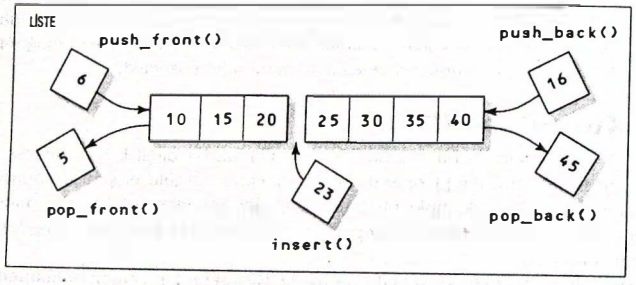
Çift uçlu bir kuyruk bazı açılardan bir vektöre, diğer açılardan bir bağlı listeye benzer. Tıpkı vektör gibi, çift uçlu bir kuyruk da [] operatörünü kullanarak rasgele erişimi destekler. Bununla birlikte, çift uçlu bir kuyruk, tıpkı bir bağlı liste gibi, arkadan olduğu gibi önden de erişilebilir. Çift uçlu kuyruk, `push_front()`, `pop_front()` ve `front()` fonksiyonlarını destekleyen bir tür çift uçlu vektördür.

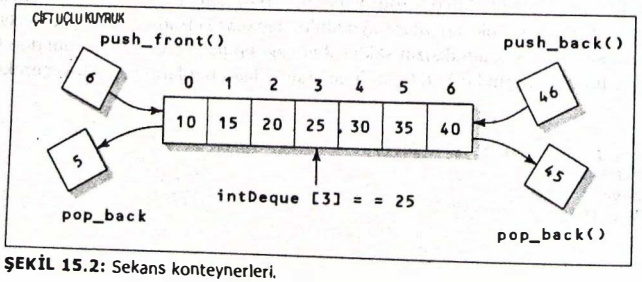
Vektör ve kuyruk için bellekte farklı şekilde yer ayrılır. Bir vektör her zaman bölünmemiş bir bellek alanını işgal eder. Eğer bir vektör çok büyürse, vektörü sığabileceği yeni bir konuma taşınmak gerekebilir. Çift uçlu bir kuyruk, öte yandan, birkaç parça halindeki bir bellek alanında saklanabilir; çift uçlu kuyruk parçalara ayrılabilir. `capacity()` adında bir üye fonksiyon, bir vektörün taşınmasına gerek kalmaksızın saklayabileceği en fazla eleman sayısını döndürür. Fakat, `capacity()` çift uçlu kuyruklar için tanımlı değildir, çünkü bunların taşınması gerekmez.

```
// deque.cpp
// push_back(), push_front() ve front()
#include <iostream>
#include <deque>
using namespace std;

int main()
   {
   deque<int> deq;
```

```
deq.push_back(30);          //ogeleri arkadan yerlestir
deq.push_back(40);
deq.push_back(50);
deq.push_front(20);         //ogeleri onden yerlestir
deq.push_front(10);

deq[2] = 33;                //ortadaki ogeyi degistir

for(int j=0; j<deq.size(); j++)
   cout << deq[j] << ' ';   //tum ogeleri goster
cout << endl;
return 0;
}
```







ŞEKİL 15.2: Sekans konteynerleri.

`push_back()`, `push_front()` ve `operator []` fonksiyonlarının örneklerini önceden görmüştük. Bunlar, diğer konteynerlerle ne şekilde çalışıyorlarsa çift uçlu kuyruklarla da aynı şekilde çalışırlar. Bu programın çıktısı ekranda şöyle görünür:

```
10 20 33 40 50
```

Şekil 15.2'de üç sekans konteyneri için bazı önemli üye fonksiyonlar gösterilmiştir.

İteratörler

İteratörler bir parça esrarengiz görünebilir, aynı zamanda STL'in işleyişinin belkemiğidir. Bu bölümde öncelikle, iteratörlerin üstlendikleri çifte rolden bahsedeceğiz: Akıllı işaretçiler olarak ve algoritmalar ve konteynerler arasında bir bağlantı olarak. Sonra, iteratörlerin kullanımlarıyla ilgili bazı örnekler göstereceğiz.

Akıllı İşaretçiler Olarak İteratörler

Genellikle konteyner içindeki elemanların tümü (veya belki de bir kısmı) üzerinde işlemler gerçekleştirmek gerekli olur. Konteyner içindeki elemanların her birinin değerini ekranda göstermek veya bu değerleri bir toplama eklemek, bu tür işlemler arasında yer alır. Sıradan bir C++ dizisinde bu tür işlemler bir işaretçi (ya da, temelinde aynı mekanizma yatan `[]` operatörü) kullanılarak yürütülür. Örneğin, aşağıdaki kod, kayan noktalı sayılardan oluşan bir dizi üzerinde iterasyon yaparken her elemanın değerini ekranda gösterir:

```
float* ptr = start_address;
for(int j=0; j<SIZE; j++)
   cout << *ptr++;
```

İşaretçinin gösterdiği öğenin değerini elde etmek için * operatörü kullanılarak `ptr` işaretçisi dereferanslanır ve `++` operatörü kullanılarak artırılır. Böylece, işaretçi bir sonraki öğeyi gösterir.

Sıradan İşaretçiler Zayıf Kalır

Ancak, daha sofistike konteynerlerde, basit C++ işaretçilerini kullanmak dezavantajlıdır. Bir defa, konteynerde saklı öğeler bellekte bir bütün olarak yerleştirilmemişse, işaretçiyi kontrol etmek fazlasıyla karmaşık hal alır; bir sonraki öğeyi göstermesi için işaretçiyi basitçe artıramayız. Örneğin, bağlı listede bir sonraki öğeye kayarken bu öğenin bir önceki öğenin yanında olduğunu farz edemeyiz; işaretçiler zincirini takip etmeliyiz.

Ayrıca, herhangi bir konteyner elemanının adresini bir işaretçi değişkeninde saklamak isteyebiliriz, böylece elemana daha sonra erişebiliriz. Konteynerin ortalarından ekleme ve çıkarma yaparsak bu saklı işaretçiye ne olur? Eğer konteynerin içeriği yeniden düzenlenmezse bu işaretçi geçerliliğini sürdüremeyecektir. Ekleme ve çıkarma işlemlerini gerçekleştirilirken saklı tutulan işaretçi değerlerinin tümünü yeniden gözden geçirmeyi dert etmek zorunda kalmasak iyi olurdu.

Bu tür problemlerin bir çözümü, bir "akıllı işaretçiler" sınıfı tanımlamaktır. Bu tür bir sınıfın nesnesi temel olarak kendisine ait üye fonksiyonlarını sıradan bir işaretçi ile sarmalar. `++` ve * operatörleri aşırı yüklenmiştir. Böylece bu operatörler, kendi konteynerleri içindeki elemanlar üzerinde ne şekilde işlem göreceklerini bilirler, hatta elemanlar bellekte bir bütün olarak yer

almıyorsa veya konumları değişmişse bile bu bilgiye sahiptirler. Bu işlemlerin nasıl gerçekleştirilebileceğine bir örnek, aşağıda görülmektedir (iskelet formunda):

```
class SmartPointer
   {
   private:
      float* p;                    //siradan bir isaretci
   public:
      float operator*()
         { }
      float operator++()
         { }
   };

void main()
   {
   ...
   SmartPointer sptr = start_address;
   for(int j=0; j<SIZE; j++)
      cout << *sptr++;
   }
```

Kimin Sorumluluğunda?

Akıllı işaretçi sınıfı bir konteyner içine mi gömülmeli, yoksa ayrı bir sınıf olarak mı kalmalıdır? STL'de tercih edilen yaklaşım, akıllı işaretçileri, (bunlar *iteratörler* olarak adlandırılır), tamamen ayrı bir sınıf (aslında bir şablonlaştırılmış sınıf ailesi) yapmaktır. Sınıfın kullanıcısı, iteratörleri bu tür sınıfların nesnesi olacak şekilde tanımlayarak oluşturur.

Arayüz Olarak İteratörler

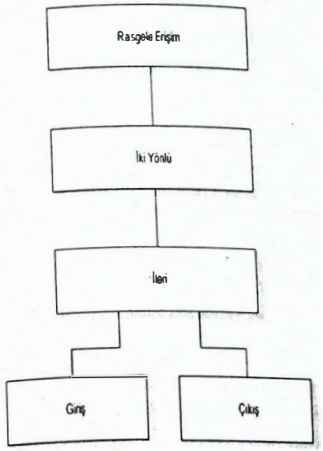
İteratörler, konteynerlerdeki öğeleri gösteren akıllı işaretçiler gibi davranmalarının yanı sıra STL'de bir başka önemli bir amaca daha hizmet ederler. İteratörler, hangi algoritmaların hangi konteynerlerle kullanılabileceğini belirlerler. Neden böyle bir şeye gerek vardır?

Teorik açıdan her algoritmayı her konteynere uygulayabiliyor olmalısınız. Üstelik, bir çok algoritma aslında STL konteynerlerinin tümüyle çalışacaktır. Yine de, bazı algoritmaların konteynerlerle kullanıldıklarında hiç elverişli olmadıkları (yani, yavaş oldukları) ortaya çıkar. Örneğin `sort()` algoritması, sıralamaya çalıştığı konteynere rasgele erişmek gereğindedir; aksi halde, elemanların her birini kaydırmadan önce aradığı elemanı bulmak için konteyner üzerinde tek tek ilerlemesi gerekecektir. Bu çok zaman alıcı bir işlemdir. Aynı şekilde, randımanlı çalışması için `reverse()` algoritmasının da konteyner üzerinde hem ileri hem de geri yönde ilerlemesi (iterasyon yapması) gerekir.

İteratörler ilgili algoritmaları konteynerlerle eşlemek için şaşırtıcı ölçüde ince bir yöntem sağlıyorlar. Önceden bahsettiğimiz gibi, bir iteratörü, bilgisayarı yazıcıya bağlamak için kullanılan bir kablo gibi düşünebilirsiniz. Kablonun bir ucu konteynere, diğer ucu da algoritmaya bağlanır. Ancak, her kablo bütün konteynerlere bağlanamaz ve her kablo bütün algoritmalara da bağlanamaz. Eğer belirli bir konteyner tipi için fazlasıyla güçlü bir algoritma kullanmaya çalışıyorsanız, bunları bağlayacak bir kablo (iteratör) bulmanız mümkün olmayabilir. Deneyecek olursanız, sizi problem hakkında uyaracak bir derleyici hata mesajı ile karşılaşacaksınız demektir.

Bu tür bir planın çalıştırılması için kaç çeşit iteratöre (kabloya) ihtiyacınız olabilir? Sadece beş çeşit iteratörün gerekli olduğu ortaya çıkar. Şekil 15.3 bu beş kategoriyi gösterir. Şekilde

kategoriler, aşağıdan yukarıya doğru artan karmaşıklık sırasına göre düzenlenmiştir (giriş ve çıkışlar eşit oranda karmaşıklığa sahiptir; aslında bu seviyede karmaşıklıktan söz etmeye bile gerek yoktur).



ŞEKİL 15.3: İteratör kategorileri.

Eğer bir algoritmanın bir konteyner boyunca öğeleri tek tek okuyarak (konteynere yazmadan) sadece adım adım ilerlemesi gerekiyorsa, bu algoritma kendisini konteynere bağlamak için *giriş* iteratörünü kullanabilir. Aslında, giriş iteratörleri konteynerlerle kullanılmak yerine genellikle dosyalardan okurken veya `cin` ile kullanılır.

Eğer bir algoritma bir konteyner boyunca ileri yönde adım adım ilerliyorsa, ama konteynerden okumak yerine konteynere yazıyorsa bu algoritma *çıkış* iteratörünü kullanabilir. Çıkış iteratörleri genellikle dosyalara veya `cout`'a yazarken kullanılır.

Eğer bir algoritma ileriye doğru adımlarken konteynerden okuyabilir ya da konteynere yazabilirse böyle bir algoritma *ileri* iteratör kullanmalıdır.

Eğer bir algoritma bir konteyner boyunca hem ileri hem geri adımlayabilmeliyse, böyle bir algoritma *iki yönlü* iteratör kullanmalıdır.

Son olarak, eğer bir algoritmanın bir konteyner içindeki herhangi bir öğeye konteyner boyunca adımlamaksızın anında erişmesi gerekiyorsa, böyle bir algoritma *rasgele erişim* (random access) iteratörü kullanmalıdır. Rasgele erişim iteratörleri, herhangi bir elemana erişebilirlik açısından dizilere benzer. Aritmetik işlemlerde kullanılabilen tek iteratörler rasgele erişim iteratörleridir:

```
iter2 = iter1 + 7;
```

Tablo 15.7 her iteratörün hangi işlemleri desteklediğini gösterir.

TABLO 15.7: Farklı İteratör Kategorilerinin Becerileri

İteratör Tipi	İleri Adımlama ++	Okuma değer=*i	Yazma *i=değer	Geri Adımlama --	Rasgele Erişim [n]
Rasgele erişim iteratörü	x	x	x	x	x
İki yönlü iteratör	x	x	x	x	
İleri iteratör	x	x	x		
Çıkış iteratörü	x		x		
Giriş iteratörü	x	x			

Gördüğünüz gibi, konteyner boyunca ileri yönde adımlamak için tüm iteratörler ++ operatörünü desteklerler. Giriş iteratörü, * operatörünü eşittir işaretinin sağ tarafında (sol tarafında değil) kullanabilir:

```
value = *iter;
```

Çıkış iteratörü * operatörünü sağda kullanabilir:

```
*iter = value;
```

İleri iteratör hem okuma hem yazma işlemini ele alır; iki yönlü iteratör ise artırılabildiği gibi, aynı zamanda azaltılabilir. Rasgele erişim iteratörü ise herhangi bir elemana hızlı erişmek için (+ veya - gibi basit aritmetik operatörlerinin yanı sıra) [] operatörünü kullanabilir.

Bir algoritma ihtiyacı olduğundan *daha fazla* beceriye sahip bir iteratörü her zaman kullanabilir. Eğer bir algoritma, örneğin ileri iteratöre gerek duyuyorsa, bu algoritmayı iki yönlü iteratöre veya rasgele erişim iteratörüne bağlamak sorun olmaz.

Algoritmaları Konteynerlerle Eşlemek

İteratörü için kablo benzetmesini kullanmıştık, çünkü bir iteratör, bir algoritma ile bir konteyneri birbirine bağlıyordu. Şimdi bu hayali kablonun iki ucuna odaklanalım: Konteyner ucu ve algoritma ucu.

Kabloyu Konteynere Takmak

Eğer kendinizi temel STL konteynerleri ile sınırlandırırsanız, sadece iki çeşit iteratör kullanıyor olacaksınız. Tablo 15.8'de gösterildiği gibi, vektör ve çift uçlu kuyruk her çeşit iteratörü kabul eder; oysa liste, küme, çoklu küme, eşleme ve çoklu eşleme, rasgele erişim iteratörü haricindeki iteratörleri kabul eder.

TABLO 15.8: Konteynerler Tarafından Kabul Edilen İteratör Tipleri

	Vektör	Liste	Çift Uçlu Kuyruk	Küme	Çoklu Küme	Eşleme	Çoklu Eşleme
Rasgele Erişim	x		x				
İki Yönlü	x	x	x	x	x	x	x
İleri	x	x	x	x	x	x	x
Giriş	x	x	x	x	x	x	x
Çıkış	x	x	x	x	x	x	x

Belirli bir konteyner için doğru iteratörün kullanılmasını STL nasıl yürütebilir? Bir iteratörü tanımladığınız zaman bu iteratörün ne tür bir konteynerle kullanılacağını da açıkça belirtmeniz gerekir. Örneğin, `int` tipinde elemanları tutan bir liste tanımlamışsanız,

```
list<int> iList;                    //tamsayılar listesi
```

bu listeyi gösteren bir iteratör tanımlamak için şöyle yazabilirsiniz:

```
list<int>::iterator iter;  //tamsayılar listesini gosteren iterator
```

Bu şekilde yazdığınız zaman STL bu iteratörü otomatik olarak çift yönlü iteratör olarak oluşturur, çünkü bir liste için gerekli olan, çift yönlü iteratördür. Bir vektörü veya çift uçlu kuyruğu gösteren iteratör ise otomatik olarak rasgele erişim iteratörü olarak oluşturulur.

Bu otomatik tercih işlemi, belirli bir konteyner için bir iteratör sınıfının, söz konusu konteynerle ilişkili daha genel bir iteratör sınıfından türetilmesi (kalıtım) yoluyla gerçeklenir. Yani, vektörleri ve çift uçlu kuyrukları gösteren iteratörler `random_access_iterator` sınıfından; listeleri gösteren iteratörler ise `bidirectional_iterator` sınıfından türetilir.

Artık, konteynerlerin, bizim hayali iteratör kablolarımızın uçları ile nasıl eşlendiğini öğrendik. Aslında bir kablo bir konteynere takılmaz; kablo (mecazi olarak konuşursak), tıpkı bir tost makinesinin kablosu gibi konteynere monte edilmiştir. Vektörler ve çift uçlu kuyruklar daima rasgele erişim kablolarına; listeler (ve bu bölümde karşılaşacağımız diğer birleşik konteynerlerin tümü) ise daima iki yönlü kablolara monte edilir.

Kabloyu Algoritmaya Takmak

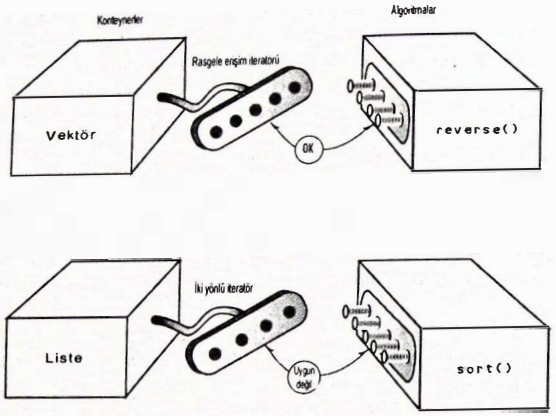
İteratör kablosunun bir ucunun konteynere nasıl takıldığını öğrendiğimize göre artık kablonun diğer ucunda neler olduğuna bakmaya hazırız demektir. Acaba iteratörler algoritmalara nasıl bağlanır? Her algoritma, bir konteynerdeki öğelere ne yapacağına bağlı olarak, belli bir tür iteratöre ihtiyaç duyar. Şayet algoritma, konteynerde herhangi bir yerde bulunan elemanlara erişmek zorunda ise, rasgele erişim iteratörüne gereksinim duyar. Eğer durum böyle değilse ve algoritmanın iteratör içinde adım adım ilerlemesi yeterli olacaksa, daha az güçlü olan ileri iteratörü kullanabilir. Tablo 15.9'da örnek bazı algoritmalar ve bunların ihtiyaç duyduğu iteratörler gösterilmiştir. (Bu tablonun tam halini Ek F'de bulabilirsiniz.)

TABLE 15.9: Temsili Bazı Algoritmaların Gerek Duyduğu İteratör Tipleri

Algoritma	Giriş	Çıkış	İleri	İki Yönlü	Rasgele Erişim
for_each	x				
find	x				
count	x				
copy	x	x			
replace			x		
unique			x		
reverse				x	
sort					x
nth_element					x
merge	x	x			
accumulate	x				

Elbette, her algoritma belli bir kapasiteye sahip bir iteratör gerektirir ama daha kuvvetli iteratörler de iş görür. Örneğin, `replace()` algoritması bir ileri iteratöre ihtiyaç duymasına rağmen iki yönlü ya da rasgele erişim iteratörleri ile de çalışır.

Algoritmaların tıpkı bilgisayarınızdaki kablo konektörleri gibi ucunda pinler olan kablolara sahip olduğunu hayal edin. Şekil 15.4'te bunu görebilirsiniz. Rasgele erişim iteratörlerine ihtiyaç duyan algoritmaların 5 pini, iki yönlü iteratöre ihtiyaç duyanları 4 pini, ileri iteratöre ihtiyaç duyan algoritmaların da 3 pini olsun.



ŞEKİL 15.4: Konteynerlerle algoritmaları birbirlerine bağlayan iteratörler

Bir iteratörün (kablonun) algoritma ucunda ise belli bir sayıda delik vardır. Beş delikli bir iteratörü 5 pinli bir algoritmaya bağlayabilirsiniz. Aynı iteratörü 4 ya da daha az sayıda pini olan bir algoritmaya da bağlamanız mümkündür. Ancak 4 delikli (iki yönlü) bir iteratörü 5 pinli (rasgele erişim) algoritmaya bağlayamazsınız. Dolayısıyla, vektörler ve çift uçlu kuyruklar rasgele erişim iteratörleri ile her türlü algoritmaya bağlanabilirler. Dört delikli iki yönlü iteratörleri olan listeler ve birleşik konteynerler ise sadece daha zayıf algoritmalara bağlanabilirler.

Konuyu Tablolar Özetler

Tablo 15.8 ve 15.9'a bakarak bir algoritmanın belli bir konteynerle çalışıp çalışamayacağına karar verebilirsiniz. Örneğin Tablo 15.9'da `sort()` algoritmasının bir rasgele erişim iteratörüne ihtiyaç duyduğunu görmekteyiz. Tablo 15.8 de rasgele erişim iteratörleriyle sadece vektör ve çift uçlu kuyruk türünde konteynerlerin çalışabileceğini göstermektedir. Kısacası, `sort()` algoritmasını listelere, kümelere, eşlemelere ve diğer uyumsuz türlere uygulamaya çalışmanın anlamı olmayacaktır.

Rasgele erişim iteratörüne gerek duymayan her algoritma, her türlü STL konteyneri ile çalışabilir. Çünkü bu konteynerlerin tümü iki yönlü iteratörlerdir, bu da rasgele erişimin bir seviye altına denk gelir. (Eğer STL'de tek bağlantılı bir liste olsaydı, bu liste, `reverse()` algoritması ile kullanılamasın diye sadece bir ileri iteratör kullanırdı.) Görebileceğiniz gibi, rasgele erişim iteratörlerine ihtiyaç duyan algoritmaların sayısı nispeten kısıtlıdır. Bir başka ifade ile, algoritmaların çoğunluğu, birçok konteyner ile çalışır.

Üye Fonksiyonlarla Algoritmaları Çakıştırmak

Bazen, bir üye fonksiyon ile aynı adı taşıyan bir algoritmayı kullanmak arasında tercih yapmak zorunda kalırsınız. Örneğin, `find()` algoritması tek bir giriş iteratörüne ihtiyaç duyduğu için herhangi bir konteynerle kullanılabilir. Ancak (sekans konteynerlerin aksine), kümeler ve eşlemelerin kendi `find()` üye fonksiyonları vardır. Peki bu durumda `find()`'in hangi versiyonunu kullanmalısınız? Genel olarak, eğer üye fonksiyon versiyonu varsa bunun sebebi, algoritmanın genel versiyonunun o konteyner için mümkün olan en verimli şekilde olmamasıdır. Dolayısıyla bu tür durumlarda üye fonksiyon versiyonunu kullanmak muhtemelen daha iyi olacaktır.

İteratörlerin Kullanımı

İteratörleri kullanmak, onları anlatmaktan büyük ölçüde daha kolaydır. İteratör kullanımının nispeten yaygın şekillerine ait örnekleri zaten önceden görmüş bulunuyoruz. Bu örneklerde teklarlayıcı değerlerini döndürme işi konteynerin `begin()` ve `end()` üye fonksiyonları tarafından görülüyordu. Daha önce gizlediğimiz bir gerçek var: Bu fonksiyonlar, iteratör değerlerini, sanki onlar işaretçiymiş gibi davranarak döndürürler. Gelin şimdi de gerçek iteratörlerin bu ve diğer fonksiyonlarla nasıl kullanıldığını görelim.

Veri Erişimi

Rasgele erişim sağlayan konteynerlerde (vektör ve çift uçlu kuyruk), [] operatörünü kullanarak konteyner üzerinde iterasyon yapmak gayet kolaydır. Listeler gibi rasgele erişimi desteklemeyen konteynerlerde ise farklı bir yaklaşım gerekir. `LIST` ve `LISTPLUS` gibi daha önceki örneklerimizde "okurken yok etme" yöntemi kullandık; bir listenin elemanlarını göstermek için elemanları tek tek listeden çıkardık. Daha pratik bir yaklaşım ise, konteyner için bir iteratör tanımlamak şeklinde olabilir. Aşağıdaki `LISTOUT` örneği, bunun nasıl yapılabileceğini göstermektedir:

```
// listout.cpp
// cikti icin iterator ve for dongusu
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
   {
   int arr[] = { 2, 4, 6, 8 };
   list<int> theList;

   for(int k=0; k<4; k++)                    //listeyi dizi elemanlari ile doldur
      theList.push_back( arr[k] );

   list<int>::iterator iter;                 //int'ler listesi icin iterator

   for(iter = theList.begin(); iter != theList.end(); iter++)
      cout << *iter << ' ';                  //listeyi goster
   cout << endl;
   return 0;
   }
```

Programın tek yaptığı iş, `theList` adlı konteynerin içeriğini göstermektir. Programın çıktısı şöyledir:

```
2 4 6 8
```

Burada konteyner tipine uyum sağlamak için `list<int>` tipi bir iteratör tanımlanmıştır. Tıpkı bir işaretçi değişkeninde olduğu gibi, bir iteratörü kullanmadan önce ona bir değer atanması gerekir. İşte bunun için `for` döngüsünde iteratöre `iList.begin()` ilk değeri verilmiştir. Bu değer konteynerin başlangıcıdır. İteratörü `++` operatörü ile artırabilir ve böylece bir konteynerin elemanlarını üzerinde adım adım ilerlemesini sağlayabiliriz. Ayrıca, `*` operatörünü kullanarak her elemanın işaret ettiği değeri öğrenebiliriz. Yapabileceğimiz bir başka şey de `!=` operatörünü kullanarak eşitliğe bakmak ve bu sayede `iList.end()`'in belirlediği konteyner sonuna ulaştığımızda döngüden çıkmaktır.

Yukarıda anlatılana denk bir yaklaşım da `for` döngüsü yerine `while` kullanarak şu şekilde olabilir:

```
iter = iList.begin();
while( iter != iList.end() )
cout << *iter++ << ' ';
```

Burada `*iter++` sözdizimi tıpkı bir işaretçide kullandığımız gibidir.

Veri Ekleme

Aşağıdaki `LISTFILL` örneğinde gösterildiği gibi, benzer bir kod kullanarak bir konteynerin mevcut öğelerine veri koyabiliriz.

```
// listfill.cpp
// listeyi veriyle doldurmak icin iterator kullanir
#include <iostream>
#include <list>
using namespace std;

int main()
   {
   list<int>iList(5);                    //5 adet int tutan bos liste
   list<int>::iterator it;               //iterator
   int data=0;
                                         //listeyi veriyle doldur
   for(it=iList.begin();it !=iList.end();it++)
      *it=data +=2;
                                        //listeyi goster
   for(it = iList.begin(); it != iList.end(); it++)
      cout << *it << ' ';
   cout << endl;
   return 0;
   }
```

İlk döngü, konteyneri 2, 4, 6, 8, 10 `int` değerleri ile doldurur. Bunu yaparken de aşırı yüklenmiş `*` operatörünün eşitliğin sadece sağ tarafında değil sol tarafında da çalıştığını gösterir. İkinci döngü ise bu değerleri ekrana yazar.

Algoritmalar ve İteratörler

Daha önce de sözünü ettiğimiz gibi, algoritmalar iteratörleri argüman olarak (bazen de dönüş değeri olarak) kullanırlar.

`ITERFIND` örneği, bir listeye uygulanan `find()` algoritmasını gösterir. (`find()` algoritması sadece bir giriş iteratörüne ihtiyaç duyduğu için bu algoritmanın listelerle kullanılabileceğini biliyoruz.)

```
// iterfind.cpp
// find() bir liste iteratoru dondurur
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;

int main()
   {
   list<int>theList(5);                  //5 int tutan bos liste
   list<int>::iterator iter;             //iterator
   int data = 0;

                                         //listeyi veriyle doldur
   for(iter = theList.begin(); iter != theList.end(); iter++)
      *iter = data += 2;                 //2,4,6,8,10
                                         //8 sayisini ara
   iter = find(theList.begin(), theList.end(), 8);
   if( iter != theList.end() )
      cout << "\nFound 8.\n";
   else
      cout << "\nDid not find 8.\n";
   return 0;
   }
```

Bir algoritma olarak `find()`'ın üç argümanı vardır. Bunlardan ikisi; arama yapılacak aralığı belirleyen iteratör değerleri, üçüncüsü de aranan değerdir. Konteyner bir önceki örnekte olduğu gibi 2, 4, 6, 8, 10 değerleri ile doldurulur. Ardından da 8 sayısını aramak için `find()` algoritması kullanılır. Eğer `find()` algoritması `iList.end()` döndürürse, aranan sayıya ulaşamadan konteynerin sonuna varıldığı anlaşılır. Başka bir dönüş değeri alınırsa, değeri 8 sayısı olan bir öğe bulunmuş demektir. Program şu çıktıyı verir:

```
Found 8.
```

8 değerinin konteyner içinde tam olarak nerede bulunduğunu anlamak için iteratörün değerini kullanabilir miyiz? Bulduğumuz öğenin konteynerin başından itibaren konumunun (`iter-iList.begin()`) ile hesaplanabileceğini düşünebilirsiniz. Ancak bu, listeler için kullanılan iteratörler üzerinde geçerli bir işlem değildir. Bir liste iteratörü sadece iki yönlü bir iteratör olduğundan onunla aritmetik yapamazsınız. Elbette, vektörler ve kuyruklarla kullanılanlar gibi rasgele erişim iteratörleri ile aritmetik yapılabilir. Bu nedenle, `iList` gibi bir listede değil de `v` adlı bir vektörde arama yapıyor olsaydınız, `ITERFIND`'ın son kısmını şu şekilde yazabilirdiniz:

```
iter = find(v.begin(), v.end(), 8);
if( iter != v.end() )
   cout << "\nFound 8 at location " << (iter-v.begin() );
else
   cout << "\nDid not find 8.";
```

Bu durumda programın çıktısı şöyle olurdu:

```
Found 8 at location 3
```

Şimdi de iteratörleri argüman olarak kullanan bir başka örnek görelim. Bu örnekte copy() algoritması bir vektörle kullanılır. Kullanıcı, bir vektörden diğerine kopyalanacak konum aralığını belirler, program da kopyalamayı gerçekleştirir. Söz konusu aralığı belirleme işini de iteratörler yapar.

```
// itercopy.cpp
// copy() algoritmasi icin iteratorler kullanir
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
   {
   int beginRange, endRange;
   int arr[] = { 11, 13, 15, 17, 19, 21, 23, 25, 27, 29 };
   vector<int> v1(arr, arr+10);     //ilk kullanima hazirlanmis vektor
   vector<int> v2(10);              //ilk kullanima hazirlanmamis vektor

   cout << "Enter range to be copied (example: 2 5): ";
   cin >> beginRange >> endRange;

   vector<int>::iterator iter1 = v1.begin() + beginRange;
   vector<int>::iterator iter2 = v1.begin() + endRange;
   vector<int>::iterator iter3;
                                    //araligi v1'den to v2'ye kopyala
   iter3 = copy( iter1, iter2, v2.begin() );
                                    //(it3 -> kopyalanan son oge)
   iter1 = v2.begin();              //aralik boyunca iterasyon yap
   while(iter1 != iter3)            //v2'de,degerleri goster
      cout << *iter1++ << ' ';
   cout << endl;
   return 0;
   }
```

Programla örnek bir etkileşim şu şekildedir:

```
Enter range to be copied (example: 2 5): 3 6
17 19 21
```

v2'nin bütün içeriği gösterilmez. Sadece kopyalanan öğeler aralığını gösterilir. Neyse ki, copy() hedef konteyner olan v2'nin son öğesine (aslında son öğeden bir sonrakine) işaret eden bir iteratör döndürür. Program bunu while döngüsünde kullanarak sadece kopyalanan öğeleri gösterir.

Özelleştirilmiş İteratörler

Bu bölümde iteratörlerin iki özelleştirilmiş şeklini inceleyeceğiz: iteratör adaptörleri, iteratörlerin davranışlarını enteresan yollardan değiştirebilirler; akış iteratörleri ise, giriş ve çıkış akışlarının iteratörler gibi davranmalarına imkan verirler.

İteratör Adaptörleri

STL, normal iteratörlerin üç değişik şeklini sunar. Bunlar *ters iteratör*, *ekleme iteratörü* ve *ham veri saklama iteratörüdür*. Ters iteratör, bir konteyner boyunca geriye doğru iterasyonla ilerlemenize imkan verir. Ekleme iteratörü, çeşitli algoritmaların, mesela copy() ve merge(), davranışlarını değiştirir, böylece, bu algoritmalar verileri mevcut verilerin üzerine yazmak yerine bir konteynere eklerler. Ham veri saklama iteratörü sayesinde çıkış iteratörleri, verileri değer atanmamış bellekte saklama imkanına kavuşur. Fakat bu, özelleştirilmiş durumlarda kullanılır; burada ham veri iteratörlerine yer vermeyeceğiz.

Ters İteratörler

Diyelim ki, bir konteynerin sonundan başına geriye doğru iterasyonla ilerlemek istiyorsunuz. Şöyle bir şey yazabileceğinizi düşünebilirsiniz:

```
list<int>::iterator iter;
iter = iList.end();                 //normal iterator
while( iter != iList.begin() )      //sondan basla
cout << *iter-- << ' ';             //basa git
                                    //iteratoru bir azalt
```

Fakat, ne yazık ki bu işe yaramaz. (Bir defa, tanım aralığı hatalı olacaktır: n-1'den 0 olması gerekirken n'den 1'e olur.)

Geriye doğru iterasyonla ilerlemek için *ters iteratör* kullanabilirsiniz. ITEREV programı, bir listenin içeriğini tersten sıralı olarak göstermek için bir ters iteratörün kullanıldığı bir örnek sunar.

```
// iterev.cpp
// ters iterator
#include <iostream>
#include <list>
using namespace std;

int main()
   {
   int arr[] = { 2, 4, 6, 8, 10 };     //tamsayilar dizisi
   list<int> theList;

   for(int j=0; j<5; j++)              //diziyi listeye
      theList.push_back( arr[j] );     //aktar

   list<int>::reverse_iterator revit; //ters iterator

   revit = theList.rbegin();           //liste boyunca
   while( revit != theList.rend() )    //ciktiyi gostererek
      cout << *revit++ << ' ';         //geriye dogru iterasyon yap
   cout << endl;
   return 0;
   }
```

Bu programın çıktısı şöyledir:

```
10 8 6 4 2
```

Ters iteratör kullandığınızda `rbegin()` ve `rend()` üye fonksiyonlarını da kullanmalısınız. (Bunları normal ileri iteratör ile kullanmaya çalışmayın.) Şaşırtıcı ama, konteynerin sonundan başlanır, fakat `rbegin()` üye fonksiyonu çağrılır. Ayrıca, iteratör artırılmalıdır. Bir ters iteratörü azaltmaya çalışmayın; `revit--`düşündüğünüz gibi çalışmaz. `reverse_iterator` yardımıyla, artırım operatörünü kullanarak her zaman `rbegin()`'den `rend()`'e doğru gidin.

Ekleme İteratörleri

Bazı algoritmalar, mesela `copy()`, hedef konteynerin mevcut içeriğinin (eğer varsa) üzerine yazar. Çift uçlu bir kuyruktan bir başka çift uçlu kuyruğa kopyalama yapan `COPYDEQ` programı bununla ilgili bir örnek sunar:

```
// copydeq.cpp
// kuyruklarla normal bir kopyalama islemi
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;

int main()
   {
   int arr1[] = { 1, 3, 5, 7, 9 };
   int arr2[] = { 2, 4, 6, 8, 10 };
   deque<int> d1;
   deque<int> d2;

   for(int j=0; j<5; j++)           //dizileri cift uclu kuyruklara aktar
      {
      d1.push_back( arr1[j] );
      d2.push_back( arr2[j] );
      }
                                    //d1'i d2'ye kopyala
   copy( d1.begin(), d1.end(), d2.begin() );

   for(int k=0; k<d2.size(); k++)   //d2'yi goster
      cout << d2[k] << ' ';
   cout << endl;
   return 0;
   }
```

Bu programın çıktısı şöyledir:

```
1 3 5 7 9
```

`d2`'nin içeriği `d1`'in içeriğinin üzerine yazılır. Yani, `d2` ekranda gösterildiği zaman eski içeriğinden (çift sayılardan ibaret) hiçbir iz taşımadığı görülür. Genellikle istenen, bu tür bir davranıştır. Yine de, kimi zaman bu `copy()` ile eklenen yeni elemanların, eskilerinin üzerine yazmak yerine konteyner içinde eskilerle birlikte olmasını tercih edebilirsiniz. Böyle bir davranışı *ekleme iteratörünü* kullanarak elde edebilirsiniz. Bu iteratörün üç çeşidi vardır:

- `back_inserter`, yeni öğeleri sondan ekler
- `front_inserter`, yeni öğeleri baştan ekler
- `inserter`, yeni öğeleri belirtilen bir konumdan ekler

`DINSITER` programı arkadan eklemenin nasıl gerçekleştirildiğini gösterir.


```
// dinsiter.cpp
// bir kuyrukla birlikte ekleme iteratoru
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;

int main()
   {
   int arr1[] = { 1, 3, 5, 7, 9 };     //d1'i ilk kullanima hazirla
   int arr2[] = { 2, 4, 6};            //d2'yi ilk kullanima hazirla
   deque<int> d1;
   deque<int> d2;

   for(int i=0; i<5; i++)
      d1.push_back( arr1[i] );         //dizileri cift uclu kuyruklara aktar
   for(int j=0; j<3; j++)
      d2.push_back( arr2[j] );
                                       //d1'i d2'nin arkasina kopyala
   copy( d1.begin(), d1.end(), back_inserter(d2) );

   cout << "\nd2: ";                   //d2'yi ekranda goster
   for(int k=0; k<d2.size(); k++)
      cout << d2[k] << ' ';
   cout << endl;
   return 0;
   }
```

Arkadan ekleyici, yeni öğeleri kaynak konteynerden (`d1`) hedef konteynerin (`d2`) sonuna, mevcut öğelerin peşine eklemek için konteynerin `push_back()`üye fonksiyonunu kullanır. `d1` konteyneri değişmeden kalır. `d2`'nin yeni içeriğini gösteren program çıktısı şöyle görünür:

```
d2: 2 4 6 1 3 5 7 9
```

Bunun yerine bir önden ekleyici belirtmiş olsaydık, yeni öğeler konteynerin önüne eklenmiş olacaktı:

```
copy( d1.begin(), d1.end(), front_inserter(d2) );
```

Önden ekleyicinin altında yatan mekanizma, öğeleri konteynerin önüne ekleyen, böylece öğelerin sırasını etkili biçimde ters çeviren, konteynerin `push_front()` üye fonksiyonudur. Çıktı bu durumda şu şekilde olacaktır:

```
9 7 5 3 1 2 4 6
```

Ekleme iteratörünün *ekleyici* (*inserter*) versiyonunu kullanarak yeni öğeleri herhangi bir elemandan başlayarak ekleyebilirsiniz. Örneğin, yeni öğeleri `d2`'nin başına eklemek için, şöyle yazardık:

```
copy( d1.begin(), d1.end(), inserter(d2, d2.begin()) );
```

Ekleyicinin ilk argümanı içine kopyalama yapılacak konteynerdir; ikinci argüman ise kopyalamanın başlaması gereken konuma işaret eden bir iteratördür. Ekleyici, konteynerin

`insert()` üye fonksiyonunu kullandığı için elemanların sırası ters çevrilmez. Bu ifadeden elde edilen çıktı şöyle olurdu:

```
1 3 5 7 9 2 4 6
```

Ekleyicinin ikinci argümanını değiştirerek yeni verilerin `d2`'nin içinde herhangi bir yere eklenmesini sağlayabiliriz.

Dikkat ederseniz, `front_inserter` bir vektörle birlikte kullanılamaz, çünkü vektörlerin `push_front()` üye fonksiyonu yoktur; vektörler sadece sondan erişilebilirler.

Akış İteratörleri

Akış iteratörleri, dosyaları ve I/O aygıtlarını (mesela, `cin` ve `cout`) sanki birer iteratör gibi ele almanıza imkan verirler. Bu, dosyaları ve I/O aygıtlarını algoritmalara argüman olarak kullanmayı kolaylaştırır. (Bu, algoritmaları ve konteynerleri bağlamak için iteratör kullanmanın sağladığı çok yönlülüğün bir başka göstergesidir.)

Giriş ve çıkış iteratör kategorilerinin başlıca amacı, bu akışa iteratör sınıflarını desteklemektir. Giriş ve çıkış iteratörleri, ilgili algoritmaların doğrudan giriş ve çıkış akışları üzerinde kullanılmalarını mümkün kılar.

Akış iteratörleri aslında farklı tipte girdi ve çıktılar için şablonlaştırılmış sınıf nesneleridir. İki çeşit akış iteratörü vardır: `ostream_iterator` ve `istream_iterator`. Şimdi sırayla bunlara bir göz atalım.

ostream_iterator Sınıfı

`ostream_iterator` nesnesi, bir çıkış iteratörünü açıkça belirten herhangi bir algoritmaya argümanı olarak kullanılabilir. `OUTITER` örneğinde bunu `copy()` fonksiyonuna argümanı olarak kullanacağız:

```
// outiter.cpp
// ostream_iterator
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;

int main()
   {
   int arr[] = { 10, 20, 30, 40, 50 };
   list<int> theList;

   for(int j=0; j<5; j++)                          //diziyi listeye aktar
      theList.push_back( arr[j] );

   ostream_iterator<int> ositer(cout, ", ");       //ostream iteratoru

   cout << "\nContents of list: ";
   copy(theList.begin(), theList.end(), ositer); //listeyi goster
   cout << endl;
   return 0;
   }
```


`int` tipinde değerleri okumak için bir `ostream` iteratörü tanımlanır. Bu kurucu fonksiyonun iki argümanı, `int` değerlerin yazılacağı akış ve her değerin peşinden gösterilecek olan bir karakter katarıdır. Akış değeri genellikle bir dosya ismi veya `cout`'tur; bu örnekte `cout`'tur.

`cout`'a yazarken sınırlandırıcı görevini taşıyan karakter katarı, istediğiniz karakterlerden oluşabilir; bu örnekte virgül ve boşluk karakteri kullanılmıştır.

`copy()` algoritması listenin içeriğini `cout`'a kopyalar. `ostream` iteratörü `copy()`'nin üçüncü argümanı olarak kullanılır; `ostream`, hedefi gösterir.

`OUTITER`'ın çıktısı şöyledir:

```
Contents of list: 10,20,30,40,50,
```

Bir sonraki örneğimiz olan `FOUTITER`, bir dosyaya yazmak için `ostream` iteratörünün nasıl kullanıldığını gösterir:

```
// foutiter.cpp
// dosyalarla ostream_iterator
#include <fstream>
#include <algorithm>
#include <list>
using namespace std;

int main()
   {
   int arr[] = { 11, 21, 31, 41, 51 };
   list<int> theList;

   for(int j=0; j<5; j++)                    //listeyi diziye
      theList.push_back( arr[j] );           //aktar
   ofstream outfile("ITER.DAT");             //dosya nesnesi olustur

   ostream_iterator<int> ositer(outfile," ");     //iterator
                                                  //listeyi dosyaya yaz
   copy(theList.begin(), theList.end(), ositer);
   return 0;
   }
```

Bir `ostream` dosya nesnesi tanımlamalısınız ve bunu bir dosya ile ilişkilendirmelisiniz; burada bu dosya `ITER.DAT` olarak adlandırılmıştır. Bu nesne `ostream_iterator`'ın birinci argümanıdır. Bir dosyaya yazarken karakter katarı argümanı içinde `"--"` gibi karakterler yerine boşluklar kullanın. Bu, verileri dosyadan geri okurken kolaylık sağlar. Bu örnekte boşluk karakteri (" ") kullanılmıştır.

`FOUTITER`'ın ekranda gösterilebilir bir çıktısı yoktur. Fakat `ITER` programı tarafından oluşturulan `ITER.DAT` dosyasını incelemek için bir metin editörü (Windows'daki Notepad aksesuarı gibi) kullanabilirsiniz. Dosyada şu veriler yer alıyor olmalıdır:

```
11 21 31 41 51
```

istream_iterator Sınıfı

Bir `istream_iterator` nesnesi bir giriş iteratörünün belirtildiği herhangi bir algoritmaya argüman olarak kullanılabilir. Örneğimiz olan `INITER`, `copy()`'nin ilk iki argümanı olarak kullanılan bu tür nesneleri gösterir. Bu program, kullanıcı tarafından `cin` üzerinden (klavyeden) girilen kayan noktalı sayıları okur ve bunları bir listede saklar.

```
// initer.cpp
// istream_iterator
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
   {
   list<float> fList(5);           //ilk kullanima hazirlanmamis liste

   cout << "\nEnter 5 floating-point numbers: ";
                                   //istream iteratorleri
   istream_iterator<float> cin_iter(cin);          //cin
   istream_iterator<float> end_of_stream;          //eos
                                   //cin'den fList'e kopyala
   copy( cin_iter, end_of_stream, fList.begin() );

   cout << endl;                   //fList'i goster
   ostream_iterator<float> ositer(cout,"--");
   copy(fList.begin(), fList.end(), ositer);
   cout << endl;
   return 0;
   }
```

INITER programıyla bir etkileşim şöyle olabilir:

```
Enter 5 floating-point numbers: 1.1 2.2 3.3 4.4 5.5
1.1--2.2--3.3--4.4--5.5--
```

copy()'ye dikkat edin; **cin**'den gelen veri hedef olmayıp kaynak olduğu için kopyalanacak veri aralığının hem başı, hem de sonu fonksiyonda belirtilir. Veri aralığının başı **cin**'e bağlanmış bir **istream_iterator**'dır; Bu, tek argümanlı kurucu fonksiyon kullanılarak **cin_iter** olarak tanımlanır. Peki, ya veri aralığının sonu? **istream_iterator**'ın argümansız (varsayılan) kurucu fonksiyonu burada özel bir rol oynar. Bu kurucu, her zaman, akışın sonunu simgeleyen bir **istream_iterator** nesnesi oluşturur.

Peki, kullanıcı verileri girerken akışın böyle bir "son" değerini nasıl üretir? Ctrl+Z tuş çiftine basarak .Ctrl+Z, akışlarda normal olarak kullanılan dosya sonu (end-of-file) karakterini gönderir. Kimi zaman Ctrl+Z'ye birkaç defa basmak gerekebilir. Enter'a basmak sayıları birbirinden ayırırsa da dosyanın sona ermesini sağlamaz.

Listenin içeriğini ekranda göstermenin elbette birçok yolu olsa da, biz bu iş için **ostream_iterator**'ı kullanıyoruz.

Herhangi bir ekran çıktısını, mesela "5 kayan noktalı sayı giriniz:" şeklinde bir yönlendirmeyi, **istream** iteratörünü kullanmadan önce gerçekleştirebileceğiniz gibi, hatta tanımlamadan önce bile gerçekleştirebilirsiniz. Bu iteratör tanımlanır tanımlanmaz ekranı kilitler ve girdiyi beklemeye başlar.

Sıradaki örneğimiz olan **FINITER**, **copy()** algoritmasına girdi olarak **cin** yerine bir dosya kullanır.

```
// finiter.cpp
// dosyalarla istream_iterator
#include <iostream>
#include <list>
```



```
#include <fstream>
#include <algorithm>
using namespace std;

int main()
   {
   list<int> iList;
   ifstream infile("ITER.DAT");          //bos liste
                                         //girdi dosya nesnesini tanimla
                                         //(ITER.DAT onceden mevcut olmali)
                                         //istream iteratorleri
   istream_iterator<int> file_iter(infile);      //dosya
   istream_iterator<int> end_of_stream;          //eos
                                         //infile'dan iList'e kopyala
   copy( file_iter, end_of_stream, back_inserter(iList) );

   cout << endl;                         //iList'i goster
   ostream_iterator<int> ositer(cout, "--");
   copy(iList.begin(), iList.end(), ositer);
   cout << endl;
   return 0;
   }
```

FINITER'in çıktısı şöyledir:

```
11--21--31--31--41--51--
```

ITER.DAT dosyasını simgeleyen bir **ifstream** nesnesi tanımlanır. **ITER.DAT** dosyası önceden mevcut ve verileri içeriyor olmalıdır. (**FOUTITER** programı, eğer çalıştırırsanız, bu dosyayı üretmiş olur.)

INITER örneğindeki **istream** iteratöründeki gibi **cout** kullanmak yerine, **infile** adında bir **ifstream** nesnesi kullanılır. Akış sonu nesnesi (**eos**) aynı şekilde kullanılır.

Bu programda bir değişiklik daha yapılmıştır: Program verileri **iList**'e eklemek için **back_inserter** iteratörünü kullanır. Bu sayede, **iList**'i belirli bir boyuttaki bir konteyner olarak tanımlamak yerine boş bir konteyner olarak tanımlamak mümkün olur. Girdi olarak kaç tane öğenin girileceği bilinemeyeceği için, girdi okurken bunu kullanmak genellikle makul olur.

Birleşik Konteynerler

Sekans konteynerlerinin (vektör, liste ve çift uçlu kuyruk) verileri sabit doğrusal bir sırayla sakladıklarını gördük. Bu tür bir konteyner içinde bir öğeyi aramak (indeks numarası bilinmiyorsa veya veri konteynerin sonunda değilse), konteyner içindeki öğelerin üzerinden tek tek adımlamayı gerektiren yavaş bir işlemden ibaret olacaktır.

Bir birleşik konteynerde öğeler sırayla düzenlemezler. Bunun yerine, aranılan öğenin çok daha hızlı bulunmasını sağlayacak daha karmaşık bir yöntemle düzenlenirler. Bu düzenleme için farklı yöntemleri kullanmak mümkün olsa da (mesela, hash tabloları gibi) genellikle ağaç (tree) yapısı kullanılır. Arama hızı, birleşik konteynerlerin en büyük avantajıdır.

Arama, genellikle bir sayı veya karakter kataı gibi tek bir değer olarak ifade edilen bir *anahtar* kullanılarak gerçekleştirilir. Bu değer, konteyner içindeki nesnelerin bir niteliği olabilir veya nesnenin bütünü de olabilir.

STL'de birleşik konteynerlerin iki ana kategorisini kümeler (set) ve eşlemeler (map) oluşturur.

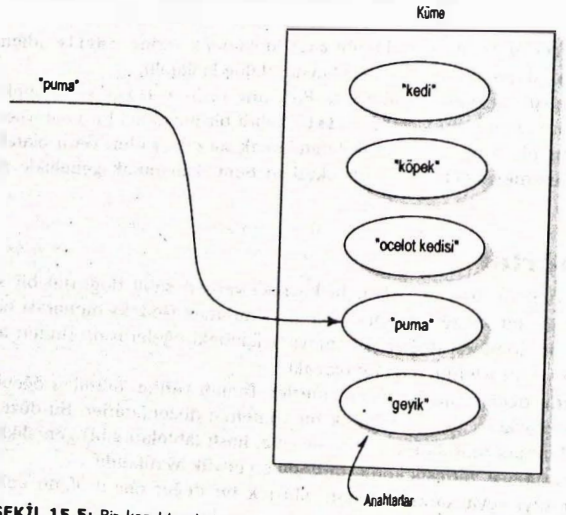
Bir küme, anahtar(lar) içeren nesneleri saklar. Bir eşleme ise, nesne çiftlerini saklar. Bu nesne çiftlerinin ilk parçası, bir anahtar içeren nesne; ikinci parçası ise bir değer içeren nesne-dir.

Kümede ve eşlemede her anahtarın tek örneği saklanabilir. Bu, her sözcük için birden fazla karşılığa izin verilmeyen bir sözlük gibidir. Ancak, STL bu kısıtlamayı gevşeten alternatif küme ve eşleme versiyonlarını da içerir. Bir *çoklu küme* (*multiset*) ve *çoklu eşleme* (*multimap*), bir kümeye ve eşlemeye benzerler ama, aynı anahtarın birkaç örneğini de içerebilirler.

Birleşik konteynerler, diğer konteynerlerle birçok üye fonksiyon paylaşırlar. Ancak, bazı algoritmalar, mesela **lower_bound()** ve **equal_range()**, yalnızca birleşik konteynerlerde mevcuttur. Ayrıca, diğer konteynerlerde gerçekten mevcut olan bazı üye fonksiyonların (**push_back()** vs) birleşik konteynerlere özgü versiyonları yoktur. "push" ve "pop" gibi fonksiyonları birleşik konteynerlerle kullanmanın pek bir esprisi yoktur; çünkü, elemanlar konteynerin başına veya sonuna eklenmek yerine her zaman kendi sıralı konumlarına eklenmelidir.

Kümeler ve Çoklu Kümeler

Kümeler genellikle kullanıcı tarafından tanımlanan sınıfların nesnelerini (mesela, bir personel veri tabanındaki çalışanlar gibi) tutmak için kullanılır. Bununla birlikte, kümeler daha basit elemanları, mesela karakter katarlarını tutmak için de kullanılabilir. Şekil 15.5'te buna bir örnek verilmiştir. Nesneler sırayla yerleştirilir ve bütün bir nesne anahtar olarak kullanılır.



ŞEKİL 15.5: Bir karakter katarı nesneleri kümesi.

İlk örneğimiz olan **SET**, **string** sınıfına ait nesneleri saklayan bir kümeyi gösterir.


```
// set.cpp
// kume string nesnelerini saklar
#pragma warning (disable:4786)
#include <iostream>                    //set icin (sadece Microsoft icin gerekli)
#include <set>
#include <string>
using namespace std;

int main()
   {
   string names[] = {"Juanita", "Robert",
                     "Mary", "Amanda", "Marie"};
                                       //string nesneleri dizisi
   set<string, less<string> > nameSet(names, names+5);
                                       //kumeyi diziyle ilk kullanıma hazırla
   set<string, less<string> >:: iterator iter;
                                       //kumeye ait iterator

   nameSet.insert("Yvette");           //daha fazla isim ekle
   nameSet.insert("Larry");
   nameSet.insert("Robert");           //etkisiz; zaten kumede var
   nameSet.insert("Barry");
   nameSet.erase("Mary");              //bir isim sil
                                       //kumenin buyuklugunu goster
   cout << "\nSize=" << nameSet.size() << endl;
   iter = nameSet.begin();             //kumenin uyelerini goster
   while( iter != nameSet.end() )
      cout << *iter++ << '\n';

   string searchName;                  //kullanicidan bir isim al
   cout << "\nEnter name to search for: ";
   cin >> searchName;
                                       //kume icinde eslenen ismi bul
   iter = nameSet.find(searchName);
   if( iter == nameSet.end() )
      cout << "The name " << searchName << " is NOT in the set.";
   else
      cout << "The name " << *iter << " IS in the set.";
   cout << endl;
   return 0;
   }
```

Şu direktif,

```
#pragma warning (disable:4786)
```

SET ve **MAP** dosyalarını kullanırken Microsoft derleyicilerde gerekli olabilir. Bu direktif, bir hata gibi görünen 4786 numaralı uyarıyı ("hata ayıklama bilgisi çerçevesinde tanımlayıcı 255 karakter kalacak şekilde kesildi" anlamındaki uyarı) etkisiz hale getirir. Pragma sadece, problem neden olan **SET** ve **MAP** başlık dosyalarından önce değil, tüm dosyalar için kullanılan **#include**'lardan önce yer almalıdır. *Pragma*, derleyici işlemlerini ayarlayan derleyiciye özgü bir direktiftir.

Bir kümeyi tanımlamak için saklanacak nesnelerin tipleri (bu örnekte, **string** sınıfı) ve kümenin üyelerini sıralamak için kullanılacak fonksiyon nesnesi açıkça belirtilir. **string** nesnelerine uygulamak için burada **less<>()** operatörü kullanılır.

Gördüğünüz gibi, küme diğer STL konteynerleriyle benzer bir arayüze sahiptir. Bir kümeye bir dizinin değerleri atanabilir ve yeni üyeler, bir kümeye **insert()** üye fonksiyonu ile eklenebilir. Kümeyi görüntülemek için küme üzerinde tek tek ilerlenebilir.

Küme içindeki belirli bir değeri bulmak için **find()** üye fonksiyonu kullanılır. (Sekans konteynerleri, **find()**'ın algoritma versiyonunu kullanırlar.) İşte **SET** ile gerçekleştirilen birkaç örnek etkileşim. Kullanıcı, aranılacak isim olarak "George" girer:

```
Size = 7
Amanda
Barry
Juanita
Larry
Marie
Robert
Yvette
Enter name to search for: George
The name George is NOT in the set.
```

Birleşik konteynerlerin arama hızında sağladığı avantaj, elbette bu örnektekinden daha fazla sayıda değer kullanılınca ortaya çıkacaktır.

Şimdi, sadece birleşik konteynerlerde mevcut olan önemli bir çift üye fonksiyona göz atalım. Örneğimiz olan **SETRANGE**, **lower_bound()** ve **upper_bound()** fonksiyonlarının kullanımını gösterir:

```
// setrange.cpp
// bir kume icindeki deger araliklarini test eder
#pragma warning (disable:4786)   //set icin (sadece Microsoft icin gerekli)
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
   {                                    //string nesneleri kumesi
   set<string, less<string> > organic;
                                        //kumeye ait iterator
   set<string, less<string> >::iterator iter;

   organic.insert("Curine");            //organik bilesikleri ekle
   organic.insert("Xanthine");
   organic.insert("Curarine");
   organic.insert("Melamine");
   organic.insert("Cyanimide");
   organic.insert("Phenol");
   organic.insert("Aphrodine");
   organic.insert("Imidazole");
   organic.insert("Cinchonine");
   organic.insert("Palmitamide");
   organic.insert("Cyanimide");

   iter = organic.begin();              //kumeyi goster
   while( iter != organic.end() )
      cout << *iter++<< '\n';

   string lower, upper;                 //deger araligi icindeki degerleri goster
```



```
   cout << "\nEnter range (example C Czz): ";
   cin >> lower >> upper;
   iter = organic.lower_bound(lower);
   while( iter != organic.upper_bound(upper) )
      cout << *iter++ << '\n';
   return 0;
   }
```

Program öncelikle bütün bir organik bileşenler kümesini ekranda gösterir. Kullanıcı daha sonra bir çift anahtar değer girmesi için yönlendirilir ve program bu aralık içinde yer alan anahtarları ekranda gösterir. İşte bir etkileşim örneği:

```
Aphrodine
Cinchonine
Curarine
Curine
Cyanimide
Imidazole
Melamine
Palmitamide
Phenol
Xanthine

Enter range (example C Czz): Aaa Curb
Aphrodine
Cinchonine
Curarine
```

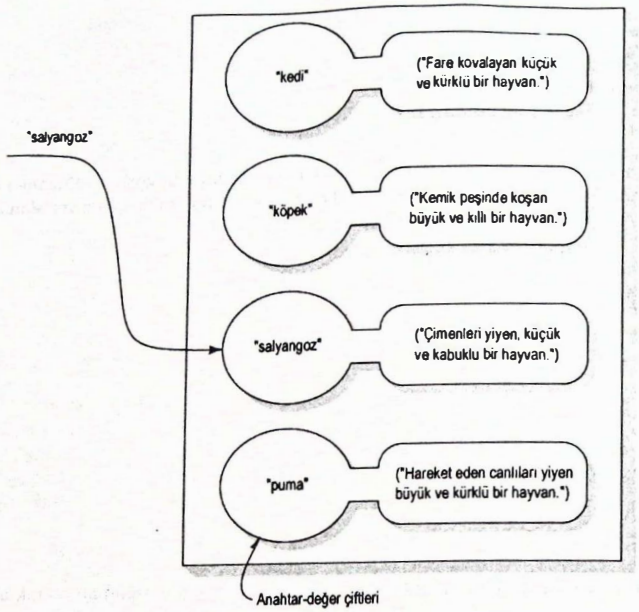
lower_bound() üye fonksiyonu, anahtar ile aynı tipteki bir değeri argüman olarak alır. Bu argümandan daha küçük olmayan ilk değere bir iteratör döndürür (buradaki "küçük" ifadesi küme tanımında kullanılan fonksiyon nesnesi tarafından belirlenir). **upper_bound()** fonksiyonu, argümanından daha büyük olan ilk değere bir iteratör döndürür. Bu fonksiyonlar birlikte, belirli bir değer aralığına erişmenize imkan verirler.

Eşlemeler ve Çoklu Eşlemeler

Bir eşleme, nesne çiftlerini saklar. Bir nesne çifti bir *anahtar nesnesinden* ve bir *değer nesnesinden* oluşur. Anahtar nesnesi aranılacak anahtarı içerir. Değer nesnesi ilave veriler içerir. Kümelerde olduğu gibi, bir anahtar nesnesi karakter katarı, sayı veya daha karmaşık bir sınıfa ait bir nesne olabilir. Değerler ise genellikle karakter katarı veya sayı olabilir ama, nesne hatta konteyner bile olabilirler. .

Örneğin anahtar, bir kelime olabilir ve değer, bu kelimenin bir belge içinde kaç kez göründüğünü simgeleyen bir sayı olabilir. Bu tür bir eşleme, bir *frekans tablosu* oluşturur. Ya da anahtar, bir kelime olabilir ve değer ise sayfa numaralarının bir listesi olabilir. Bu düzenleme, kitabının sonundaki gibi bir indeksi simgeleyebilir. Şekil 15.6'da, sıradan bir sözlükteki gibi, anahtarların kelimelere, değerlerin ise tanımlara karşılık geldiği bir durumu gösterir.

Eşlemelerin en yaygın olarak kullanıldığı yöntemlerden biri, birleşik diziler olarak kullanımlarıdır. Sıradan bir C++ dizisinde, belirli bir elemana erişmek için kullanılan dizi indeksi bir tamsayıdır. Yani, **anArray[3]** deyimindeki 3 dizi indeksidir. Bir birleşik dizi de aynı şekilde çalışır. Aradaki fark, birleşik dizilerde dizi indeksinin tipini seçebilmenizdir. Eğer indeksi karakter katarı olarak tanımlamışsanız, örneğin **anArray["jane"]** diyebilirsiniz.



ŞEKİL 15.6: Kelime-cümlecik çiftlerinin bir eşlemesi.

Bir Birleşik Dizi

Şimdi, bir birleşik dizi olarak kullanılan basit bir eşleme örneğine göz atalım. Anahtarlar, eyalet isimleri; değerler ise eyaletlerin nüfusları olacaktır. **ASSO_ARR** programının listesi şöyledir:

```
// asso_arr.cpp
// iliskili dizi olarak kullanilan bir esleme
#pragma warning (disable:4786)      //map icin (sadece Microsoft icin)
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
   {
   string name;
   int pop;

   string states[] = { "Wyoming", "Colorado", "Nevada",
                       "Montana", "Arizona", "Idaho" };
   int pops[] = { 470, 2890, 800, 787, 2718, 944 };

   map<string, int, less<string> >mapStates;              //esleme
   map<string, int, less<string> >::iterator iter;        //iterator
```



```
   for(int j=0; j<6; j++)
      {
      name = states[j];
      pop=pops[j];                          //dizilerden veri al
      mapStates[name] = pop;
      }                                     //bunu eslemeye koy
   cout << "Enter state: ";
   cin >> name;                             //kullanicidan eyaleti ogren
   pop = mapStates[name];
   cout << "Population: " << pop << ",000\n";   //nufusu bul

   cout << endl;
   for(iter = mapStates.begin(); iter != mapStates.end(); iter++)   //eslemenin tamamini goster
      cout << (*iter).first << "  " << (*iter).second << ",000\n";
   return 0;
   }
```

Program çalışırken, kullanıcıdan bir eyalet ismi girmesi istenir. Program sonra eşlemeye bakar, eyalet ismini indeks olarak kullanarak eyaletin nüfusunu döndürür. Son olarak, eşlemedeki tüm eyalet-nüfus çiftlerinin dökümünü alır. İşte birkaç örnek çıktı:

```
Enter state: Wyoming
Population: 470,000

Arizona 2718,000
Colorado 2890,000
Idaho 944,000
Montana 787,000
Nevada 800,000
Wyoming 470,000
```

Arama hızı, kümeler ve eşlemeler üstün nitelikler kazandıkça artar. Bu örnekte, kullanıcı bir eyalet ismi girdiğinde program karşılık gelen nüfusu çabucak bulur. (Eğer milyonlarca veri öğesi olsaydı bu daha anlamlı olacaktı.) Eyaletler ve nüfuslar listesi ile gösterildiği gibi, konteyner boyunca iterasyonla ilerlemek sekans konteynerlerdeki kadar hızlı değildir ama, yine de oldukça verimlidir. Orijinal veriler sıralı olmamasına rağmen eyaletlerin alfabetik olarak sıralanmış olduğuna dikkat edin.

Bir eşleme tanımında üç şablon argümanı kullanılır:

```
map<string, int, less<string> > maStates;
```

Bunlardan ilki, anahtarın tipidir. Bu örnekte, anahtarın tipi, eyalet ismini simgeleyen karakter katarıdır. İkinci argüman değerin tipidir; bu örnekte değer, 1,000'ler mertebesinde nüfusu simgeleyen bir tamsayıdır. Üçüncü argüman ise anahtarlar için kullanılacak olan sıralamayı belirtir. Biz bu argümanın, **less<string>** ifadesinin yaptığı gibi, eyaletlerin isimlerini alfabetik olarak sıralamasını tercih ettik. Ayrıca, bu eşlemede bir de iteratör tanımlanır.

Girdi verileri iki ayrı dizide saklanır. (Gerçek bir programda bu veriler muhtemelen bir dosyadan gelir.) Bu verileri bir eşlemeye yerleştirmek için veriler, **name** ve **pop** adlı değişkenlere okunur ve aşağıdaki ifade çalıştırılır:

```
mapStates[name] = pop;
```

Bu, tıpkı sıradan bir diziye ekleme yapılıyormuş gibi görünen özellikle çok zarif bir ifadedir. Ancak, dizi indeksi bir tamsayı yerine bir karakter katarıdır.

Kullanıcı bir eyalet ismi girdiğinde program karşılık gelen nüfusu şu ifade ile bulur:

```
pop = mapStates[name];
```

Dizi indeksi söz dizimini kullanmanın yanı sıra eşlemedeki bir öğenin her iki parçasına (anahtara ve değere) bir iteratör kullanarak erişilebilir. Anahtar, `(*iter).first` ile, değer ise `(*iter).second` ile elde edilir. Aksi halde, iteratör tıpkı diğer konteynerlerdeki gibi çalışır.

Kullanıcı Tarafından Tanımlanan Nesneleri Depolamak

Şimdiye kadar örnek programlarımız temel tiplerdeki nesneleri depolamışlardı. Ancak, STL ile elde edilen büyük kazanç, STL'i kullanarak kendi yazdığınız (veya bir başkası tarafından yazılmış olan) sınıf nesnelerini depolayabilmeniz ve idare edebilmenizdir. Bu bölümde, bunun nasıl gerçekleştirildiğini göstereceğiz.

person Nesnelerinin Kümesi

Bir kişinin soyadını, adını ve telefon numarasını içeren bir `person` sınıfı ile başlayacağız. Bu sınıf için bazı üye fonksiyonlar oluşturacağız ve bunları bir kümeye ekleyeceğiz; böylece, bir telefon rehberi ver tabanı oluşturacağız. Kullanıcı bir kişinin ismini girerek programla etkileşir. Ardından program listeyi arar ve eğer bir eşleme bulursa karşılık gelen kişinin verilerini ekranda gösterir. Çoklu kümeleri kullanacağız; böylece, iki veya daha fazla kişi aynı isme sahip olabilecek. **SETPERS** programının listesi şöyledir:

```
// setpers.cpp
// person nesnelerini tutmak icin coklu kume kullanir
#pragma warning (disable:4786)   //set icin (sadece Microsoft icin)
#include <iostream>
#include <set>
#include <string>
using namespace std;

class person
   {
   private:
      string lastName;
      string firstName;
      long phoneNumber;
   public:
      person() : lastName("blank"),                    //varsayilan kurucu fonk.
                 firstName("blank"), phoneNumber(0)
         { }

      person(string lana, string fina, long pho) :     //3 argumanli kurucu fonk.
            lastName(lana), firstName(fina), phoneNumber(pho)
         { }
      friend bool operator<(const person&, const person&);
      friend bool operator==(const person&, const person&);

      void display() const                             //kisinin verilerini goster
```



```
         {
         cout << endl << lastName << ",\t" << firstName
              << "\t\tPhone: " << phoneNumber;
         }
   };

bool operator<(const person& p1, const person& p2)       //person sinifi icin < operatoru
   {
   if(p1.lastName == p2.lastName)
      return (p1.firstName < p2.firstName) ? true : false;
   return (p1.lastName < p2.lastName) ? true : false;
   }

bool operator==(const person& p1, const person& p2)      //person sinifi icin == operatoru
   {
   return (p1.lastName == p2.lastName &&
           p1.firstName == p2.firstName) ? true : false;
   }
//////////////////////////////////////////////////////////////
int main()
   {
   person pers1("Deauville", "William", 8435150);        //person nesnelerini olustur
   person pers2("McDonald", "Stacey", 3327563);
   person pers3("Bartoski", "Peter", 6946473);
   person pers4("KuangThu", "Bruce", 4157300);
   person pers5("Wellington", "John", 9207404);
   person pers6("McDonald", "Amanda", 8435150);
   person pers7("Fredericks", "Roger", 7049982);
   person pers8("McDonald", "Stacey", 7764987);

   multiset< person, less<person> > persSet;             //kisilerin coklu kumesi
   multiset<person, less<person> >::iterator iter;       //kisilerin coklu kumesini gosteren iterator

   persSet.insert(pers1);                                //kisileri coklu kumeye koy
   persSet.insert(pers2);
   persSet.insert(pers3);
   persSet.insert(pers4);
   persSet.insert(pers5);
   persSet.insert(pers6);
   persSet.insert(pers7);
   persSet.insert(pers8);

   cout << "\nNumber of entries = " << persSet.size();

   iter = persSet.begin();                               //coklu kumenin icerigini goster
   while( iter != persSet.end() )
      (*iter++).display();

                                                         //soyadini ve adini al
   string searchLastName, searchFirstName;
   cout << "\n\nEnter last name of person to search for: ";
   cin >> searchLastName;
   cout << "Enter first name: ";
   cin >> searchFirstName;
                                                         //bu isimde bir kisi olustur
   person searchPerson(searchLastName, searchFirstName, 0);

                                                         //bu tur kisilerin sayisini al
   int cntPersons = persSet.count(searchPerson);
```

```
cout << "Number of persons with this name = " << cntPersons;
                                        //eslenenlerin tumunu goster
iter = persSet.lower_bound(searchPerson);
while( iter != persSet.upper_bound(searchPerson) )
   (*iter++).display();
cout << endl;
return 0;
}     //main()'in sonu
```

Gerekli Üye Fonksiyonlar

STL konteynerler çalışmak için **person** sınıfı birkaç ortak üye fonksiyona gereksinim duyar. Bunlar varsayılan (argümansız) kurucu fonksiyon (bu örnekte aslında gerekli değil ama, genellikle önemlidir), aşırı yüklenmiş < operatörü ve aşırı yüklenmiş == operatörüdür. Bu üye fonksiyonlar liste sınıfı ve çeşitli algoritmalar tarafından kullanılırlar. Diğer spesifik durumlar için başka üye fonksiyonlara ihtiyaç duyabilirsiniz. (Bir çok sınıfta olduğu gibi, burada da muhtemelen aşırı yüklenmiş atama operatörünü, kopyalama kurucu fonksiyonunu ve yok edici fonksiyonu ayrıca sağlamanız gerekebilir. Fakat, listenin karmaşıklaşmasını önlemek amacıyla biz bunlara burada yer vermedik.)

Aşırı yüklenmiş < ve == operatörleri **const** argümanlar kullanmalıdır. Bunları **friend** yapmak genellikle en iyi yöntemdir ama, üye fonksiyonları da ayrıca kullanabilirsiniz.

Sıralama

Aşırı yüklenmiş < operatörü, kümenin içindeki elemanların sıralanış yöntemini belirtir. **SETPERS** programında bu operatör, kişileri soyadına göre sıralamak için, eğer soyisimler aynıysa isimlere göre sıralamak için kullanılır.

SETPERS programıyla gerçekleştirilen etkileşim örnekleri aşağıda yer alıyor. Program öncelikle listenin tümünü ekranda gösterir. (Elbette çok sayıda elemanı olan gerçek bir veri tabanında bu işlem pratik olmayacaktır.) Elemanlar çoklu kümede saklandıkları için otomatik olarak sıralanırlar. Sonra, girdi satırında kullanıcı "McDonald"'ın peşinde "Stacey" ismini giriyor (önce soyadı giriliyor). Listede bu isimde iki kişi vardır; bu nedenle, her ikisi de ekranda gösterilir.

```
Number of entries = 8
Bartoski,       Peter       phone: 6946473
Deauville,      William     phone: 8435150
Fredericks,     Roger       phone: 7049982
KuangThu,       Bruce       phone: 4157300
McDonald,       Amanda      phone: 8435150
McDonald,       Stacey      phone: 3327563
McDonald,       Stacey      phone: 7764987
Wellington,     John        phone: 9207404

Enter last name of person to search for: McDonald
Enter first name: Stacey
Number of persons with this name = 2
McDonald,       Stacey      phone: 3327563
McDonald,       Stacey      phone: 7764987
```


Tıpkı Temel Tipler Gibi

Gördüğünüz gibi, bir sınıf tanımlanır tanımlanmaz, bu sınıfın nesneleri konteyner tarafından temel tipte değişkenlerle aynı şekilde ele alınır.

Öncelikle, öğelerin toplam sayısını göstermek için **size()** üye fonksiyonu kullanılır. Sonra, öğelerin tümünü ekranda göstererek liste üzerinde iterasyonla ilerlenir.

Bir çoklu küme kullanıldığı için belli bir aralığın içine düşen elemanların tümünü göstermek için **lower_bound()** ve **upper_bound()** üye fonksiyonları kullanıma hazırdır. Örnekteki çıktıda alt ve üst sınırlar aynıdır. Bu nedenle, aynı isimdeki kişilerin tümü ekranda gösterilir. Dikkat ederseniz, bulmak istediğimiz kişi (veya kişiler) ile aynı isimde "hayali" bir kişi oluşturulması gerekir. Sonra, **lower_bound()** ve **upper_bound()** fonksiyonları bu kişiyi listedeki diğer kişilerle eşlerler.

person Nesnelerinin Bir Listesi

SETPERS örneğindeki gibi, belirli bir isme sahip bir kişiyi bir kümede veya çoklu kümede aramak çok hızlı gerçekleştirilir. Eğer bir kişi nesnesinin çabucak eklenmesi veya silinmesi ile daha çok ilgileniyorsanız, bunun yerine bir liste kullanmaya karar verebiliriz. **LISTPERS** örneğinde bu durum ele alınmıştır:

```
// listpers.cpp
// kisi nesnelerini tutmak icin bir liste kullanir
#include <iostream>
#include <list>
#include <algorithm>
#include <string>
using namespace std;

class person
   {
   private:
      string lastName;
      string firstName;
      long phoneNumber;
   public:
      person() :                          //argumansiz kurucu fonks.
            lastName("blank"), firstName("blank"), phoneNumber(0L)
         { }
                                          //3 argumanli kurucu fonk.
      person(string lana, string fina, long pho) :
            lastName(lana), firstName(fina), phoneNumber(pho)
         { }
      friend bool operator<(const person&, const person&);
      friend bool operator==(const person&, const person&);
      friend bool operator!=(const person&, const person&);
      friend bool operator>(const person&, const person&);

      void display() const                //verilerin tumunu goster
         {
         cout << endl << lastName << ",\t" << firstName
              << "\t\tPhone: " << phoneNumber;
         }

      long get_phone() const              //telefon numarasini dondur
         { return phoneNumber; }
```

```
    };                          //person sinifi icin asiri yuklenen == operatoru
bool operator==(const person& p1, const person& p2)
   {
   return (p1.lastName == p2.lastName &&
           p1.firstName == p2.firstName) ? true : false;
   }
                                //person sinifi icin asiri yuklenen < operatoru
bool operator<(const person& p1, const person& p2)
   {
   if(p1.lastName == p2.lastName)
      return (p1.firstName < p2.firstName) ? true : false;
   return (p1.lastName < p2.lastName) ? true : false;
   }
                                //person sinifi icin asiri yuklenen != operatoru
bool operator!=(const person& p1, const person& p2)
   { return !(p1==p2); }
                                //person sinifi icin asiri yuklenen > operatoru
bool operator>(const person& p1, const person& p2)
   { return !(p1<p2) && !(p1==p2); }
////////////////////////////////////////////////////////////////
int main()
   {
   list<person> persList;       //kisilerin listesi
                                //kisilerin listesini gosteren bir iterator
   list<person>::iterator iter1;
                                //kisileri listeye koy
   persList.push_back( person("Deauville", "William", 8435150) );
   persList.push_back( person("McDonald", "Stacey", 3327563) );
   persList.push_back( person("Bartoski", "Peter", 6946473) );
   persList.push_back( person("KuangThu", "Bruce", 4157300) );
   persList.push_back( person("Wellington", "John", 9207404) );
   persList.push_back( person("McDonald", "Amanda", 8435150) );
   persList.push_back( person("Fredericks", "Roger", 7049982) );
   persList.push_back( person("McDonald", "Stacey", 7764987) );

   cout << "\nNumber of entries = " << persList.size();

   iter1 = persList.begin();                //listenin icerigini goster
   while( iter1 != persList.end() )
      (*iter1++).display();

//belirtilen isme sahip kisi veya kisileri bul (soyadi ve adi)
   string searchLastName, searchFirstName;
   cout << "\n\nEnter last name of person to search for: ";
   cin >> searchLastName;
   cout << "Enter first name: ";
   cin >> searchFirstName;
                                //bu isimde bir kisi olustur
   person searchPerson(searchLastName, searchFirstName, 0L);
                                //ilk eslenen ismi ara
   iter1 = find(persList.begin(), persList.end(), searchPerson);
   if( iter1 != persList.end() )          //diger eslenenleri de bul
      {
      cout << "Person(s) with that name is(are)";
      do
         {
         (*iter1).display();    //esleneni goster
         ++iter1;               //tekrar ara, eslenenin bir sonrasindan itibaren
```



```
         iter1 = find(iter1, persList.end(), searchPerson);
         } while( iter1 != persList.end() );
      }
   else
      cout << "There is no person with that name.";

//belirtilen telefon numarasina sahip kisi veya kisileri bul
   cout << "\n\nEnter phone number (format 1234567): ";
   long sNumber;                //aranılacak numarayi al
   cin >> sNumber;
                                //listede iterasyonla ilerle
   bool found_one = false;
   for(iter1=persList.begin(); iter1 != persList.end(); ++iter1)
      {
      if(sNumber == (*iter1).get_phone() )        //sayilari karsilastir
         {
         if( !found_one )
            {
            cout << "Person(s) with that phone number is(are)";
            found_one = true;
            }
         (*iter1).display();                      //esleneni goster
         }
      }     //for'un sonu
   if( !found_one )
      cout << "There is no person with that phone number";
   cout << endl;
   return 0;
   }     //main()'in sonu
```

Belirli Bir İsme Sahip Tüm Kişileri Bulmak

`lower_bound()/upper_bound()` üye fonksiyonlarını kullanamayız çünkü bir küme veya eşleme ile değil, bir listeyle ilgileniyoruz. Bunun yerine, belirli bir isme sahip tüm kişileri bulmak için `find()` üye fonksiyonunu kullanırız. Eğer bu fonksiyon bir isabet kaydederse, aynı isimde başka kişiler olup olmadığını anlamak için asıl eşlenen kişinin bir sonrasındaki kişiden başlayarak bu fonksiyonu tekrar uygulamalıyız. Bu, programlamayı karmaşıklaştırır; bir döngü kullanmalıyız ve `find()`'a iki kez çağrıda bulunmalıyız.

Belirli Bir Telefon Numarasına Sahip Tüm Kişileri Bulmak

Belirli bir telefon numarasına sahip bir kişiyi aramak belirli bir isme sahip olanı aramaktan daha zordur, çünkü sınıf üye fonksiyonlarının (mesela `find()`) en temel arama özelliğini bulmak için kullanılmaları planlanmıştır. Bu örnekte telefon numarasını bulmak için kaba kuvvet yöntemini kullanıyoruz; listede iterasyonla ilerlerken aradığımız sayı ile listenin her üyesini "elle" karşılaştırıyoruz:

```
if( sNumber == (*iter1).getphone() )
   ...
```

Program öncelikle tüm öğeleri ekranda gösterir, sonra kullanıcının bir isim girmesini ister ve eşlenen kişi veya kişileri bulur. Program sonra bir telefon numarası sorar ve yine eşlenen herhangi biri olup olmadığını bulur. `LISTPERS` programı ile birkaç etkileşim örneği şöyledir:

```
Number of entries = 8
Deauville,      William     phone: 8435150
McDonald,       Stacey      phone: 3327563
Bartoski,       Peter       phone: 6946473
KuangThu,       Bruce       phone: 4157300
Wellington,     John        phone: 9207404
McDonald,       Amanda      phone: 8435150
Fredericks,     Roger       phone: 7049982
McDonald,       Stacey      phone: 7764987

Enter last name of person to search for: Wellington
Enter first name: John
Person(s) with that name is(are)
Wellington,     John        phone: 9207404

Enter phone number (format 1234567): 8435150
Person(s) with that number is(are)
Deauville,      William     phone: 8435150
McDonald,       Amanda      phone: 8435150
```

Buradaki program, belirtilen isimde bir kişi ve belirtilen telefon numarasına sahip iki kişi bulur.

Sınıf nesnelerini saklamak için listeleri kullanırken söz konusu sınıf için dört adet karşılaştırma operatörü bildirilmesi gerekir: `==`, `!=`, `<` ve `>`. Gerçekte hangi algoritmayı kullandığınıza bağlı olarak bu operatörlerin tümünü tanımlamak (operatörler için fonksiyon gövdeleri sağlamak) zorunda kalmayabilirsiniz. Bu örnekte, bütünlük açısından her ne kadar dördü de tanımlanmış olsa da aslında yalnızca `==` operatörünü tanımlamaya ihtiyaç duyulur. Eğer liste üzerinde **`sort()`** algoritması kullanılmış olsaydı, `<` operatörünün de tanımlanması gerekecekti.

Fonksiyon Nesneleri

Fonksiyon nesneleri STL'de yaygın biçimde kullanılır. Belirli algoritmalara argüman olarak kullanılmaları bunların önemli kullanımlarından biridir. Fonksiyon nesneleri, bu algoritmaların işleyişini ayarlayabilmenize imkan verir. Bu bölümde daha önce fonksiyon nesnelerinden bahsetmiştik ve **`SORTEMP`** programında bir tane kullanmıştık. O programda, derleyici içinde önceden tanımlı bir fonksiyon örneği olarak, verileri tersten sıralamak için kullanılan **`greater<>()`** fonksiyonunu göstermiştik. Bu bölümde, önceden tanımlı diğer fonksiyon nesnelerini inceleyeceğiz; ayrıca, kendi fonksiyon nesnelerinizi nasıl yazabileceğinizi de öğreneceğiz. Böylece STL algoritmaları üzerinde daha fazla kontrole sahip olacaksınız.

Hatırlarsanız, bir fonksiyon nesnesi, bir nesne gibi görünecek biçimde, bir sınıf tarafından sarmalanmış bir fonksiyondur. Ancak, sınıfın hiç verisi yoktur; sadece bir tane üye fonksiyonu vardır. Bu üye fonksiyon da, aşırı yüklenmiş `()` operatörüdür. Sınıf genellikle şablonlaştırılır, böylece farklı tiplerle de çalışabilir.

Derleyici İçinde Tanımlı Fonksiyon Nesneleri

`FUNCTIONAL` başlık dosyasında yer alan, derleyici içinde önceden tanımlı STL fonksiyon nesneleri Tablo 15.10'da gösterilmiştir. Başlıca C++ operatörlerinin tümüne karşılık gelen fonksiyon nesneleri mevcuttur. Tabloda *T* harfi ya kullanıcı tarafından yazılmış ya da temel tipte herhangi bir sınıfı belirtir. `x` ve `y` değişkenleri, fonksiyon nesnesine argüman olarak aktarılan `T` sınıfının nesnelerini simgeler.

TABLO 15.10: Derleyici İçinde Tanımlı Fonksiyon Nesneleri

Fonksiyon Nesnesi	Dönüş Değeri
`T=plus(T, T)`	`x+y`
`T=minus(T, T)`	`x-y`
`T=times(T, T)`	`x*y`
`T=divide(T, T)`	`x/y`
`T=modulus(T, T)`	`x%y`
`T=negate(T)`	`-x`
`bool=equal_to(T, T)`	`x == y`
`bool=not_equal_to(T, T)`	`x != y`
`bool=greater(T, T)`	`x > y`
`bool=less(T, T)`	`x < y`
`bool=greater_equal(T, T)`	`x >= y`
`bool=less_equal(T, T)`	`x <= y`
`bool=logical_and(T, T)`	`x && y`
`bool=logical_or(T, T)`	`x || y`
`bool=logical_not(T)`	`!x`

Aritmetik işlemler, karşılaştırmalar ve mantıksal işlemler için fonksiyon nesneleri mevcuttur. Şimdi, bir aritmetik fonksiyon nesnesinin kullanışlı olabileceği bir örneğe göz atalım. Örneğimiz **`airtime`** adında bir sınıf kullanır. Bu sınıf, saniye haricinde saat ve dakikadan oluşan zaman değerlerini simgeler. Bu veri tipi, hava alanlarında uçakların iniş ve kalkış saatleri için uygundur. Bu örnek, bir konteyner içindeki **`airtime`** değerlerinin tümünü toplamak için **`plus<>()`** fonksiyon nesnesinin nasıl kullanılabileceğini gösterir. **`PLUSAIR`** programının listesi şöyledir:

```
// plusair.cpp
// accumulate() algoritmasini ve plus() fonksiyon nesnesini kullanir
#include <iostream>
#include <list>
#include <numeric>                          //accumulate() icin
using namespace std;
////////////////////////////////////////////////////////////////
class airtime
   {
   private:
      int hours;                            //0'dan 23'e
      int minutes;                          //0'dan 59'a
   public:
                                            //varsayilan kurucu fonk.
      airtime() : hours(0), minutes(0)
         { }
                                            //2-argumanli kurucu fonk.
      airtime(int h, int m) : hours(h), minutes(m)
         { }
      void display() const                  //ciktiyi ekrana gonder
         { cout << hours << ':' << minutes;}
      void get()                            //kullanicidan girdi al
```

```
        {
        char dummy;
        cout << "\nEnter airtime (format 12:59): ";
        cin >> hours >> dummy >> minutes;
        }
                                        //asiri yuklenmis + operatoru
    airtime operator + (const airtime right) const
        {                               //uyeleri topla
        int temph = hours + right.hours;
        int tempm = minutes + right.minutes;
        if(tempm >= 60)                 //eldeyi kontrol et
           { temph++; tempm -= 60; }
        return airtime(temph, tempm);   //toplami dondur
        }
                                        //asiri yuklenmis == operatoru
    bool operator == (const airtime& at2) const
        { return (hours == at2.hours) &&
                 (minutes == at2.minutes); }
                                        //asiri yuklenmis < operatoru
    bool operator < (const airtime& at2) const
        { return (hours < at2.hours) ||
                 (hours == at2.hours && minutes < at2.minutes); }
                                        //asiri yuklenmis != operatoru
    bool operator != (const airtime& at2) const
        { return !(*this==at2); }
                                        //asiri yuklenmis > operatoru
    bool operator > (const airtime& at2) const
        { return !(*this<at2) && !(*this==at2); }
};    //class airtime'ın sonu
////////////////////////////////////////////////////////////////
int main()
    {
    char answer;
    airtime temp, sum;
    list<airtime> airlist;              //airtime'lar listesi

    do {                                //airtime'ları kullanicidan al
       temp.get();
       airlist.push_back(temp);
       cout << "Enter another (y/n)? ";
       cin >> answer;
       } while (answer != 'n');
                                        //airtime'ların tumunu topla
    sum = accumulate( airlist.begin(), airlist.end(),
                      airtime(0, 0), plus<airtime>() );
    cout << "\nsum = ";
    sum.display();                      //toplami goster
    cout << endl;
    return 0;
    }
```

Bu program **accumulate()** algoritması ile dikkat çekiyor. Bu fonksiyonun iki versiyonu vardır. Üç argümanlı versiyonu, her zaman bir değer aralığındaki değerleri (+ operatörünü kullanarak) toplar. Burada gösterilen dört argümanlı versiyonunda Tablo 15.10'da gösterilen aritmetik fonksiyon nesnelerinden herhangi biri kullanılabilir.

accumulate()'in bu versiyonundaki dört argüman şunlardır: Değer aralığının ilk ve son elemanlarını gösteren iteratörler, toplamın başlangıçtaki değeri (genellikle 0) ve elemanlara

uygulanacak işlem. Bu örnekte elemanlar, **plus<>()** fonksiyon nesnesi kullanılarak toplanır; ancak elemanlar çıkarılabilir, çarpılabilir veya farklı fonksiyon nesneleri kullanılarak diğer işlemler de gerçekleştirilebilirdi. **PLUSAIR** programıyla birkaç etkileşim şöyle görünür:

```
Enter airtime (format 12:59): 3:45
Enter another (y/n)? y
Enter airtime (format 12:59): 5:10
Enter another (y/n)? y
Enter airtime (format 12:59): 2:25
Enter another (y/n)? y
Enter airtime (format 12:59): 0:55
Enter another (y/n)? n
sum = 12:15
```

Konteyner üzerinden iterasyonla ilerleyip elemanları sizin toplamanıza kıyasla **accumulate()** algoritması çok daha kolay ve anlaşılır olmasını yanı sıra, aynı zamanda çok daha verimlidir (elbette kodunuza yapılması gereken çok fazla iş yerleştirmediyseniz).

plus<>() fonksiyon nesnesi, **airtime** sınıfı için + operatörünün aşırı yüklenmesini gerektirir. **plus<>()** fonksiyon nesnesinin beklediği bir sabit fonksiyon olduğu için bu operatör de bir **const** fonksiyon olmalıdır.

Diğer aritmetik fonksiyon nesneleri de aynı şekilde çalışır. Mantıksal fonksiyon nesneleri ise, mesela **logical_and<>()**, bu işlemlerin anlamlı olacağı sınıflara ait nesneler (mesela, **bool** tipinde değişkenler) üzerinde kullanılabilir.

Kendi Fonksiyon Nesnelerinizi Yazmak

Eğer standart fonksiyon nesnelerinden biri istediğiniz gibi çalışmıyorsa, kendi fonksiyon nesnenizi yazabilirsiniz. Sıradaki örneğimiz kendi fonksiyon nesnemizi yazmak isteyebileceğimiz iki durumu gösterir. Biri **sort()** algoritmasını, diğeri ise **for_each()**'i içerir.

Sınıfın içindeki < operatöründe belirtilen ilişkileri temel alarak bir grup elemanı sıralamak kolaydır. Peki, nesnelerin kendilerini gösteren bir konteyner yerine nesnelere işaret eden işaretçileri içeren bir konteyneri sıralamak isterseniz ne olur? İşaretçileri saklamak, özellikle büyük nesneler açısından verimliliği artırmak için iyi bir yöntemdir, çünkü bu sayede, bir nesne bir konteynere yerleştirildiği zaman gerçekleştirilmesi gereken kopyalama işleminden kurtulmuş oluruz. Yine de, işaretçileri sıralamaya çalışırsanız, nesnelerin herhangi bir niteliğe göre değil de, işaretçi adreslerine göre düzenlendiğini göreceksiniz.

sort() algoritmasının bir işaretçi konteynerinde istediğimiz gibi çalışmasını sağlamak için **sort()** algoritmasını, verilerin sıralanışını istediğimiz gibi tanımlayan bir fonksiyon nesnesi ile birlikte tedarik etmeliyiz.

Örnek programımız **person** nesnelerini gösteren bir işaretçi vektörü ile başlar. Bu nesneler vektörün içine yerleştirilir, sonra her zamanki gibi sıralanır. Bu işlem **person** nesnelerinin değil, işaretçilerin sıralanmasına yol açar. Oysa, istediğimiz bu değildir. Ayrıca, söz konusu işlem, örneğimizdeki sıralamada hiçbir değişikliğe neden olmaz, çünkü öğeler zaten artan adres sırasına göre eklenmişlerdir. Daha sonra vektör, **comparePersons()** fonksiyon nesnesi kullanılarak düzgün biçimde sıralanır. Bu, işaretçilerin kendilerini sıralamak yerine *içeriklerini* sıralar. Sonuçta **person** nesneleri isimlerine göre alfabetik olarak sıralanmış olur. **SORTPTRS** programının listesi şöyledir:

```
// sortptrs.cpp
// isaretci kullanilarak saklanan person nesnelerini siralar
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

class person
   {
   private:
      string lastName;
      string firstName;
      long phoneNumber;
   public:
      person() :                              //varsayilan kurucu fonks.
           lastName("blank"), firstName("blank"), phoneNumber(0L)
         { }
                                              //3-argumanli kurucu fonks.
      person(string lana, string fina, long pho) :
           lastName(lana), firstName(fina), phoneNumber(pho)
         { }
      friend bool operator<(const person&, const person&);
      friend bool operator==(const person&, const person&);

      void display() const                    //kisinin verilerini goster
         {
         cout << endl << lastName << ",\t" << firstName
              << "\t\tPhone: " << phoneNumber;
         }

      long get_phone() const                  //telefon numarasini dondur
         { return phoneNumber; }
   };    //class person'un sonu
//--------------------------------------------------------------
//person sinifi icin asiri yuklenen < operatoru
bool operator<(const person& p1, const person& p2)
   {
   if(p1.lastName == p2.lastName)
      return (p1.firstName < p2.firstName) ? true : false;
   return (p1.lastName < p2.lastName) ? true : false;
   }
//--------------------------------------------------------------
//person sinifi icin asiri yuklenen == operatoru
bool operator==(const person& p1, const person& p2)
   {
   return (p1.lastName == p2.lastName &&
           p1.firstName == p2.firstName) ? true : false;
   }
//--------------------------------------------------------------
//isaretci kullanarak kisileri karsilastiran fonksiyon nesnesi
class comparePersons
   {
   public:
      bool operator() (const person* ptrP1,
                       const person* ptrP2) const
         { return *ptrP1 < *ptrP2; }
   };
//--------------------------------------------------------------
```



```
//isaretci kullanarak kisinin verilerini ekranda gosteren fonksiyon nesnesi
class displayPerson
   {
   public:
      void operator() (const person* ptrP) const
         { ptrP->display(); }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   vector<person*> vectPtrsPers;          //kisilere isaret eden isaretci vektoru

   person* ptrP1 = new person("KuangThu", "Bruce", 4157300);
   person* ptrP2 = new person("Deauville", "William", 8435150);
   person* ptrP3 = new person("Wellington", "John", 9207404);
   person* ptrP4 = new person("Bartoski", "Peter", 6946473);
   person* ptrP5 = new person("Fredericks", "Roger", 7049982);
   person* ptrP6 = new person("McDonald", "Stacey", 7764987);

   vectPtrsPers.push_back(ptrP1);         //kisileri kumeye koy
   vectPtrsPers.push_back(ptrP2);
   vectPtrsPers.push_back(ptrP3);
   vectPtrsPers.push_back(ptrP4);
   vectPtrsPers.push_back(ptrP5);
   vectPtrsPers.push_back(ptrP6);

   for_each(vectPtrsPers.begin(),         //vektoru goster
            vectPtrsPers.end(), displayPerson() );
                                          //isaretcileri sirala
   sort( vectPtrsPers.begin(), vectPtrsPers.end() );
   cout << "\n\nSorted pointers";
   for_each(vectPtrsPers.begin(),         //vektoru goster
            vectPtrsPers.end(), displayPerson() );

   sort(vectPtrsPers.begin(),             //kisileri sirala
        vectPtrsPers.end(), comparePersons() );
   cout << "\n\nSorted persons";
   for_each(vectPtrsPers.begin(),         //vektoru goster
            vectPtrsPers.end(), displayPerson() );
   while( !vectPtrsPers.empty() )
      {
      delete vectPtrsPers.back();         //kisiyi sil
      vectPtrsPers.pop_back();            //isaretciyi al
      }
   cout << endl;
   return 0;
   }     //main()'in sonu
```

SORTPTRS programının çıktısı şöyle olur:

```
KuangThu,      Bruce    phone: 4157300
Deauville,     William  phone: 8435150
Wellington,    John     phone: 9207404
Bartoski,      Peter    phone: 6946473
Fredericks,    Roger    phone: 7049982
McDonald,      Stacey   phone: 7764987

Sorted pointers
```

```
KuangThu,      Bruce     phone: 4157300
Deauville,     William   phone: 8435150
Wellington,    John      phone: 9207404
Bartoski,      Peter     phone: 6946473
Fredericks,    Roger     phone: 7049982
McDonald,      Stacey    phone: 7764987

Sorted persons
Bartoski,      Peter     phone: 6946473
Deauville,     William   phone: 8435150
Fredericks,    Roger     phone: 7049982
KuangThu,      Bruce     phone: 4157300
McDonald,      Stacey    phone: 7764987
Wellington,    John      phone: 9207404
```

Önce orijinal sıralama gösterilir, sonra işaretçilerin sıralanması ile elde edilen hatalı sıralama gösterilir ve son olarak da isme göre doğru sıralama gösterilir.

comparePersons() Fonksiyon Nesnesi

Eğer `sort()` algoritmasının iki argümanlı versiyonunu kullanırsak,

```
sort( vectPtrsPers.begin(), vectPtrsPers.end() );
```

o zaman sadece işaretçiler sıralanır. Sıralama, işaretçilerin bellek adreslerine göre yapılır. Genellikle istediğimiz bu değildir. `person` nesnelerini isme göre sıralamak için, `comparePersons()` fonksiyon nesnesi üçüncü argüman olacak şekilde `sort()` algoritmasının üç argümanlı versiyonu kullanılır:

```
sort(vectPtrsPers.begin(),
     vectPtrsPers.end(), comparePersons() );
```

`comparePersons()` fonksiyon nesnesi `SORTPTRS` programında aşağıdaki gibi tanımlanır:

```
//isaretci kullanarak kisileri karsilastiran fonksiyon nesnesi
class comparePersons
   {
   public:
      bool operator()(const person* ptrP1,
                      const person* ptrP2) const
         { return * ptrP1 < *ptrP2; }
   };
```

`operator()`, kişilere işaret eden iki işaretçiyi argüman olarak alır ve işaretçilerin kendilerini sıralamak yerine içeriklerini sıralar.

displayPersons() Fonksiyon Nesnesi

Bir konteynerin içeriğini görüntülemek için önceki yöntemlerden daha farklı bir yöntem kullanılır. Konteyner içinde iterasyonla ilerlemek yerine, `for_each()` fonksiyonu, fonksiyon nesnesi fonksiyonun üçüncü nesnesi olacak şekilde kullanılır.

```
for_each(vectPtrsPers.begin(),
         vectPtrsPers.end(), displayPeson() );
```


Bu, vektör içindeki her kişi için `displayPerson()` fonksiyon nesnesinin bir kez çağrılmasına neden olur. `displayPerson()`'un kodu şöyledir:

```
//isaretci kullanarak bir kisiye ait verileri gosteren fonksiyon nesnesi
class displayPerson
   {
   public:
      void operator() (const person* ptrP) const
         { ptrP->display(); }
   };
```

Bu düzenlemeyle, tek bir fonksiyon çağrısı, vektördeki `person` nesnelerinin tümünü görüntüler.

Konteyner Davranışını Değiştirmek İçin Kullanılan Fonksiyon Nesneleri

`SORTPTRS` programında, algoritmaların davranışlarını değiştirmek için kullanılan fonksiyon nesnelerini gösterdik. Fonksiyon nesneleri konteynerlerin de davranışlarını değiştirebilir. Örneğin, eğer nesnelere işaret eden bir işaretçi kümesinin kendi kendisini otomatik olarak ve işaretçiler yerine nesnelere dayalı biçimde sıralamasını istiyorsanız, konteyneri tanımladığınızda uygun bir fonksiyon nesnesi kullanabilirsiniz. Bu durumda, `sort()` algoritmasının kullanılmasına gerek kalmaz. Bu yöntemi alıştırmalarda inceleyeceğiz.

Özet

Bu bölümde STL'e yüzeysel bir giriş yaptık. Yine de, başlıca konulara değindik. STL'i yararlı bir şekilde kullanmaya başlayacak kadar yeterli bir bilgiye sahip olmuş olmalısınız. STL'i tam olarak anlamak için okuyucuların konuyla ilgili komple bir dokümandan yararlanmalarını tavsiye ediyoruz.

STL'in üç ana bileşenden meydana geldiğini öğrendiniz: Konteynerler (containers), algoritmalar ve iteratörler (iterators). Konteynerler iki kısma ayrılır: Sekans (sequential) ve birleşik (associative). Sekans konteynerleri vektör, liste ve çift uçlu kuyruktur (deque). Birleşik konteynerler ise küme, eşleme ve bunlarla yakından alakalı çoklu küme ve çoklu eşlemedir. Algoritmalar konteynerler üzerinde sıralama, kopyalama ve arama gibi çeşitli işlemler yürütürler. İteratörler, konteyner elemanlarına işaret eden işaretçiler gibi davranırlar ve algoritmalarla konteynerler arasında bağlantı sağlarlar.

Algoritmaların tümü konteynerlerin hepsi için uygun değildir. Algoritmalarla konteynerlerin uygun biçimde eşlendiğini garanti etmek için iteratörler kullanılır. Belirli tipteki konteynerler için iteratörler tanımlanır ve bu iteratörler algoritmalara argüman olarak aktarılır. Eğer konteynerin iteratörleri bir algoritmayla eşlenmiyorsa, bu durum derleyici hatası ile sonuçlanır.

Giriş ve çıkış iteratörleri doğrudan I/O akışlarına bağlanırlar; böylece, verilerin doğrudan I/O aygıtları ve konteynerler arasında akışına imkan verirler. Özelleştirilmiş konteynerler geriye doğru iterasyonları mümkün kılarlar; ayrıca, bazı algoritmaların davranışlarını değiştirerek verilerin mevcut verilerin üzerine yazılması yerine, mevcut verilere eklenmesini de sağlarlar.

Algoritmalar, birçok farklı konteyner üzerinde çalışabilen ayrı birer fonksiyondur. Ek olarak, her konteynerin kendine has üye fonksiyonları vardır. Bazı durumlarda aynı fonksiyon hem algoritma hem üye fonksiyon olarak mevcut olabilir.

STL konteynerleri ve algoritmaları, herhangi bir sınıfa ait nesnelerle çalışabilir. Bunun için sağlanması gereken şart, belirli üye fonksiyonların, mesela < operatörünün, bu sınıf için aşırı yüklenmiş olması gerektiğidir.

`find_if()` gibi bazı algoritmaların davranışı, fonksiyon nesneleri kullanılarak özelleştirilebilir. Bir fonksiyon nesnesi, bir sınıftan, sadece bir `()` operatörü kullanılarak örneklenir.

Sorular

Soruların cevapları Ek G'de bulabilirsiniz.

1. Bir STL konteyneri aşağıdakilerden hangisi için kullanılabilir?
   a. `employee` sınıfından nesneleri saklamak için.
   b. Elemanları, hızlıca erişilebilecek şekilde saklamak için.
   c. C++ programlarını derlemek için.
   d. Nesnelerin bellekte saklanma şeklini organize etmek için.
2. STL sekans konteynerleri şunlardır: v_____, l_____, ve ç_____.
3. Önemli iki "birleşik konteyner" şunlardır: k_____ ve e_____.
4. Bir STL algoritması,
   a. konteynerler üzerinde çalışan tek başına bir fonksiyondur.
   b. üye fonksiyonlarla konteynerler arasında bağlantıdır.
   c. uygun konteyner sınıflarının bir `friend` fonksiyonudur.
   d. uygun konteyner sınıflarının bir üye fonksiyonudur.
5. Doğru/Yanlış: STL'de iteratörün amaçlarından biri; algoritmalarla konteynerleri birbirine bağlamaktır.
6. `find()` algoritması için aşağıdakilerden hangisi doğrudur?
   a. İki konteynerde birbirine uyan eleman sekanslarını bulur.
   b. Belirtilen bir konteynere uyan bir konteyner bulur.
   c. İlk iki argümanı iteratörlerdir.
   d. İlk iki argümanı konteyner elemanlarıdır.
7. Doğru/Yanlış: Algoritmalar sadece STL konteynerleri üzerinde kullanılabilir.
8. Bir aralık (range) bir algoritmaya genellikle iki i_____ değeri ile sağlanır.
9. Bir algoritmanın davranışını özelleştirmek için sıkça kullanılan unsur nedir?
10. Bir vektör ne zaman uygun bir konteynerdir?
    a. Vektörün keyfi bir şekilde seçilecek konumlarına çok sayıda eleman ekleneceği zaman.
    b. Hepsi konteynerin önüne konulmak üzere, yeni elemanlar ekleneceği zaman.
    c. Elinizde bir indeks sayısı varken ve bu sayıya karşılık gelen elemana hızlıca erişmek istediğinizde.
    d. Elinizde bir elemanın anahtar değeri varken ve buna karşılık gelen elemana hızlıca erişmek istediğinizde.
11. Doğru/Yanlış: `back()` üye fonksiyonu, konteynerin sonundaki elemanı konteynerden çıkarır.
12. Varsayılan kurucu ile bir `v` vektörü tanımlar, tek argümanlı bir kurucu ile boyutu 11 olan bir `w` vektörü tanımlar, bu vektörlerin her birine `push_back()` ile 3'er eleman eklerseniz; `size()` üye fonksiyonu `v` için _____, `w` için de _____ döndürecektir.
13. `unique()` algoritması bir konteynerden tüm _____ elemanları çıkarır.

14. İki uçlu bir kuyrukta:
    a. Keyfi olarak seçilecek herhangi bir konumda hızlıca veri eklenebilir ve silinebilir.
    b. Keyfi olarak seçilecek herhangi bir konumda veri eklenebilir ve silinebilir ama bu işlem nispeten yavaş olur.
    c. İki uçta da veri hızlıca eklenebilir ve silinebilir.
    d. İki uçta da veri eklenebilir ve silinebilir ama bu işlem nispeten yavaş olur.
15. Bir iteratör, bir konteynerdeki belli bir elemana _____.
16. Doğru/Yanlış: Bir iteratör her zaman bir konteyner içinde ileri veya geri gidebilir.
17. Bir liste için en azından _____ bir iteratör kullanmalısınız.
18. `iter`'in bir konteynere iteratör olduğunu varsayarak, `iter`'in işaret ettiği nesnenin değerine sahip olacak, ardından `iter`'in bir sonraki elemana işaret etmesini sağlayacak bir ifade yazın.
19. `copy()` algoritması aşağıdakilerden hangisine işaret eden bir iteratör döndürür?
    a. Kendisinden kopyalanan son elemana. (Kopya işleminin kaynağı.)
    b. Kendisine kopyalanan son elemana. (Kopya işleminin hedefi.)
    c. Kendisinden kopyalanan son elemandan bir sonraki elemana.
    d. Kendisine kopyalanandan bir sonraki elemana.
20. Bir `reverse_iterator` kullanmak için ona aşağıdakilerden hangisini yapmak gerekir?
    a. Başlangıç değeri olarak `end()` vermek.
    b. Başlangıç değeri olarak `rend()` vermek.
    c. Konteynerde bir adım geri gidecek şekilde artırmak.
    d. Konteynerde bir adım geri gidecek şekilde azaltmak.
21. Doğru/Yanlış: `back_inserter` iteratörü her zaman yeni elemanların, mevcut elemanların peşine eklenmesini sağlar.
22. Akış iteratörleri; ekran ve klavye cihazları ile dosyalara sanki onlar birer _____ imiş gibi davranmanızı sağlar.
23. Bir `ostream_iterator`'ın ikinci argümanı neyi belirler?
24. Birleşik bir konteynerde aşağıdakilerden hangisi doğrudur?
    a. Değerler sıralı olarak tutulur.
    b. Anahtarlar sıralı olarak tutulur.
    c. Sıralama her zaman alfabetik ve nümeriktir.
    d. İçeriği sıralı tutmak için `sort()` algoritmasını kullanmak gereklidir.
25. Bir kümeyi tanımlarken _____ belirlemeniz gerekir.
26. Doğru/Yanlış: Bir kümede, `insert()` üye fonksiyonu bir anahtarı sıradaki doğru yerine ekler.
27. Bir eşleme, bir _____ nesne (veya değer) saklar.
28. Doğru/Yanlış: Bir eşleme, aynı anahtar değeri olan iki ya da daha fazla elemana sahip olabilir.
29. Bir konteynerde, nesneler yerine, nesneleri gösteren işaretçiler saklarsanız, aşağıdakilerden hangisi olur?
    a. Konteynerde veri saklamayı gerçeklemek için nesneleri kopyalamak gerekmeyecektir.
    b. Yalnızca birleşik konteynerler kullanılabilir.
    c. Nesne niteliklerini anahtar olarak kullanmak suretiyle nesneleri sıralamak mümkün olmayacaktır.

d. Konteynerler genelde daha az belleğe ihtiyaç duyacaktır.

30. Küme gibi birleşik bir konteynerin kendi kendisini otomatik olarak sıralamasını istiyorsanız, sıralama şeklini bir fonksiyon nesnesinde tanımlayabilirsiniz ve bu fonksiyon nesnesini konteynerin _____ belirtebilirsiniz.

Alıştırmalar

Alıştırmaların cevaplarını Ek G'de bulabilirsiniz.

1. `sort()` algoritmasını kullanıcı tarafından girilen bir kayan noktalı değerler dizisine uygulayan ve sonucu ekranda gösteren bir program yazın.*
2. `sort()` algoritmasını kullanıcı tarafından girilen kelime dizisine uygulayın ve sonucu ekranda gösterin. Kelimeleri eklemek için `push_back()`'i; kelimeleri ekranda göstermek için ise `[]` operatörünü ve `size()` fonksiyonunu kullanın.*
3. Bir tamsayı değerler listesiyle başlayın. Listenin içeriğini tersine çevirmek için, bir `while` döngüsü içinde biri liste üzerinden ileriye doğru hareket eden, diğeri ise geriye doğru giden iki normal (ters olmayan) iteratör kullanın. Birkaç ifade daha fazla yazmaktan kurtulmak için `swap()` algoritmasını kullanabilirsiniz. (Çözümünüzün hem çift hem de tek sayıdaki öğeler için çalıştığından emin olun.) Uzmanların bunu nasıl gerçekleştirdiğini anlamak için derleyicinizin `ALGORTIHM` başlık dosyası içindeki `reverse()` fonksiyonuna bir göz atın.
4. `person` sınıfı ile başlayın ve `person` nesnelerini gösteren işretçileri tutan bir çoklu küme tanımlayın. Çoklu kümeyi `comparePersons` fonksiyon nesnesiyle tanımlayın; böylece çoklu küme kişilerin isimlerine göre otomatik olarak sıralanacaktır. Yarım düzine kişi tanımlayın, bunları çoklu kümeye koyun ve çoklu kümenin içeriğini görüntüleyin. Çoklu kümenin aynı anahtarlı birden fazla nesneyi sakladığını doğrulamak için birkaç kişinin isimleri aynı olmalıdır.
5. Bir diziyi çift sayılarla, bir kümeyi tek sayılarla doldurun. Bu konteynerleri bir vektör oluşturacak şekilde birleştirmek için `merge()` algoritmasını kullanın. Her şeyin yolunda gittiğini göstermek için vektörün içeriğini ekranda gösterin.
6. Üçüncü alıştırmada, bir konteynerin içeriğini tersine çevirmek için sıradan (ters olmayan) iki iteratör kullanılmıştı. Şimdi, aynı işlemi gerçekleştirmek için, bu kez bir vektör üzerinde bir ileri yönde, bir de geriye doğru bir iteratör kullanın.
7. `PLUSAIR` programında `accumulate()` algoritmasının dört argümanlı versiyonunu göstermiştik. Algoritmanın üç argümanlı versiyonunu kullanarak bu örnek programı yeniden yazın.
8. `copy()` algoritmasını bir konteyner içindeki bir değer sekansını kopyalamak için kullanabilirsiniz. Yine de, hedef sekansın kaynak sekans ile çakışması durumunda dikkatli olmalısınız. `copy()` algoritmasını kullanarak, bir dizi içindeki herhangi bir sekansı yine dizi içinde farklı bir konuma kopyalamanıza imkan veren bir program yazın. Kullanıcının `first1`, `last1` ve `first2` değerlerini girmesini sağlayın. Hedefi ile çakışan bir sekansı, sağa değil, sola kaydırabileceğinizi doğrulamak için programınızı kullanın. (Örneğin, 10'dan 9'a kadar birkaç tane öğe kaydırabilirsiniz ama, 10'dan 11'e kaydıramazsınız.) Bunun nedeni, `copy()`'nin en soldaki elemandan başlamasıdır.
9. C++ operatörlerine karşılık gelen fonksiyon nesnelerini Tablo 15.10'da listeledik. Ayrıca, bu bölümün başlarındaki `PLUSAIR` programında, `accumulate()` algoritmasıyla birlikte kullanılan `plus<>()` fonksiyon nesnesini de gösterdik. O örnekte fonksiyon nesnelerine argüman sağlamamız şart değildi; fakat kimi zaman gerekli olabilir. Ancak,

tahmin edebileceğiniz gibi, argümanı fonksiyon nesnesinin parantezleri içine yerleştiremezsiniz. Bunun yerine, argümanı fonksiyona bağlamak için `bind1st` veya `bind2nd` adlandırılan bir fonksiyon adaptörü kullanırsınız. Örneğin, bir karakter katarı konteyneri içinde, varsayalım, belirli bir karakter katarını (bunu `searchName` olarak adlandıralım) arıyoruz. Şöyle yazabilirsiniz:

```
ptr = find_if(names.begin(), names.end(),
                          bind2nd(equal_to<string>(), searchName) );
```

Burada, `equal_to<>()` ve `searchName`, `bind2nd()`'ın argümanlarıdır. Bu ifade, konteyner içinde `searchName`'e eşit olan ilk karakter katarına bir iteratör döndürür. Bir karakter katarı konteyneri içinde bir karakter katarı bulmak için bu ifadeyi veya benzer bir ifadeyi içeren bir program yazın. Programınız `searchName`'in konteyner içindeki konumunu ekranda göstermelidir.

10. Yedinci alıştırmada bahsedilen problemin (yani, kaynaklardan herhangi biri hedeflerin herhangi biri ile çakışırsa sekansı sola kaydıramıyorsunuz) üstesinden gelmek için `copy_backward()` algoritmasını kullanabilirsiniz. Çakışmalardan bağımsız olarak, herhangi bir değer sekansını konteyner içinde herhangi bir yere kaydırmayı mümkün kılmak için hem `copy()` hem de `copy_backward()` algoritmalarını kullanan bir program yazın.
11. Akış iteratörlerini kullanarak, tamsayılardan oluşan bir kaynak dosyayı bir hedef dosyaya kopyalayan bir program yazın. Kullanıcı hem kaynak hem de hedef dosya isimlerini programa sağlamalıdır. Bir `while` döngüsü kullanabilirsiniz. Döngü içinde, giriş iteratöründen gelen tamsayı değerlerini tek tek okuyun ve anında çıkış iteratörüne yazın, sonra her iki iteratör bir kez artırın. Bu bölümdeki `FOUTITER` programı tarafından üretilen `ITER.DAT` dosyası uygun bir kaynak dosya görevi görür.
12. Bir frekans tablosu, bir metin dosyası içindeki kelimeleri ve her kelimenin dosyada kaç kez göründüğünü listeler. İsmini kullanıcının girdiği bir dosya için bir frekans tablosu oluşturan bir program yazın. `string-int` çiftlerinin bir eşlemesini kullanabilirsiniz. Noktalama işaretlerini kontrol etmek için C kütüphane fonksiyonlarından `ispunct()`'ı (`CTYPE.H` başlık dosyasında) kullanmak isteyebilirsiniz, böylece `substr()` karakter katarı üye fonksiyonunu kullanarak noktalama işaretlerini kelimenin sonundan çıkarıp alabilirsiniz. Ayrıca, kelimeleri küçük harfe çevirmek için `tolower()` fonksiyonu kullanışlı olabilir.

ONALTINCI BÖLÜM

16

NESNE YÖNELİMLİ YAZILIM GELİŞTİRME

Yazılım Geliştirme Sürecinin Gelişimi

Use Case Modelleme

Programlama Problemi

LANDLORD Programını Ayrıntılandırma Aşaması

Use Case'lerden Sınıflara

Kodu Yazmak

Programla Etkileşim

Nihai Düşünceler

Bu kitaptaki programlar oldukça kısadır; bu nedenle, bu programların geliştirilme yöntemlerinde fazla bir formalite gerekmez. Oysa, düzinelerce veya yüzlerce programcıyı içeren ve milyonlarca satır uzunluğunda kaynak kodlar üretilen tam ölçekli yazılım projelerinde durum farklıdır. Bu tür projelerde iyi tanımlanmış bir geliştirme yöntemini takip etmek gereklidir. Bu bölümde bu tür bir yönteme (en azından bu tür bir yöntemin çok yoğunlaştırılmış bir versiyonuna) göz atacağız.

Bu kitabın konuları içinde UML şemalarının birçok örneğini gördük. UML, bir yazılım geliştirme yöntemi değildir; UML görsel bir modelleme dilidir. Bununla birlikte UML, birazdan göreceğimiz gibi, geliştirme sürecinde büyük öneme sahip bir rol üstlenebilir.

Yazılım Geliştirme Sürecinin Gelişimi

Yazılım geliştirme için bir yöntem fikri, on yıllardır süren bilgisayar kullanımının neticesinde yavaş yavaş gelişti. Bu süreci kısaca özetleyeceğiz.

Gelişmemiş Programlama Süreci

İlk yıllarda yazılımla ilgili neredeyse hiçbir yöntem söz konusu değildi. Programcı, durumu potansiyel kullanıcılarla tartışır ve hemen kodu yazmaya başlardı. Çok küçük programlar için bu kadarı yeterliydi.

Şelale Yöntemi (Waterfall Process)

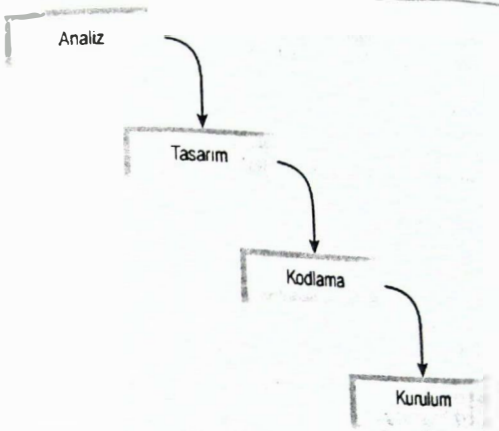
Daha sonraları, programlar büyümeye başladıkça geliştirme süreci, her biri birbirinin peşi sıra yürütülen birkaç aşamaya bölündü. Bu yöntem, imalat endüstrisinden örnek alındı. Aşamalar analiz, tasarım ve kuruluş gibi adlarla etiketlendirildi. Bu, genellikle *şelale* yöntemi (waterfall process) olarak bilinirdi, çünkü arka arkaya gelen işlemler, Şekil 16.1'de gösterildiği gibi, analiz aşamasından kuruluşa doğru tek yönde seyrederdi. Özellikle, her aşamada farklı çalışma ekipleri kullanılırdı. Her aşama tamamlandıktan sonra bu aşamada elde edilen sonuçlar farklı bir ekibe aktarılırdı.

Deneyimler şelale yönteminde temel problemler olduğunu ortaya koydu. Bu yöntemin temelini oluşturan varsayıma göre her aşama hatasız olarak (veya en azından, yalnızca çok ufak hatalarla) tamamlanacaktı. Oysa, gerçek dünyada bu durum çok ender ortaya çıkıyordu. Genellikle her aşamada ciddi yanlışlar veya ihmaller söz konusu oluyordu. Bu yanlışlar her aşamada, bir sonraki aşamaya çığ gibi büyüyerek aktarılıyor ve sonraki aşamalarda gerçekleştirilen işlerin tamamını veya bir kısmını ya tamamen kullanılmaz hale getiriyor ya da benzer şekilde hatalarla dolu olmasına neden oluyordu.

Ayrıca, geliştirme sürecinde sistemi kullanacak olan kişilerin ihtiyaçları da değişebilir, programa ilave bazı özellikler eklenmesini talep edebilirlerdi. Ancak, bir kez tasarım aşaması tamamlandıktan sonra tasarımı değiştirmek zor bir işti. Bu, programın daha kodlanırken kısmen de olsa eskimiş olması anlamına geliyordu.

Nesne Yönelimli Programlama

Bölüm 1'de "Genel Görünüm" başlığı altında bahsettiğimiz gibi, nesne yönelimli programlama, büyük programların geliştirilme sürecinde ortaya çıkan bazı problemleri çözmek amacıyla tasarlandı. Nesne yönelimli programlama elbette tasarım aşamasına yardımcı olur, çünkü programdaki nesneler, kullanıcının dünyasındaki nesnelere karşılık gelir.



ŞEKİL 16.1: Şelale yöntemi (waterfall process).

Bununla birlikte nesne yönelimli programlama, programın ne yapacağı hakkında bilgi vermez. Nesne yönelimli programlama, ancak projenin amaçları belirlendikten sonra gündeme gelir. Her şeyden önce, programın kullanıcılarına odaklanan ve kullanıcıların ihtiyaçlarına cevap veren bir başlangıç aşamasına gerek duyulur. Bu aşama başarıyla sonuçlanır sonuçlanmaz bunu bir nesne yönelimli program tasarımına dönüştürebiliriz. Fakat, kullanıcıların gerçekte neye ihtiyaç duyduklarını tespit etmemizi gerektiren ilk adımı nasıl gerçekleştireceğiz?

Modern Yöntemler

Geçmiş senelerde çok sayıda yazılım geliştirme yöntemi ortaya çıktı. Bu yöntemler, yöntemi oluşturan adımları ve müşteri, analist, tasarımcı ve programcıların birlikte ne şekilde çalışmaları gerektiğini açıkça belirtir. Bugüne kadar hiçbir sistem, UML'in modelleme dilleri alanında sahip olduğu evrensel kabule benzer bir seviyeye ulaşamadı. Aslında, uzmanların bir çoğu bir geliştirme yönteminin her duruma uygun olabileceğine inanmazlar. Belirli bir yöntem seçilmiş bile olsa, bu yöntemin, uygulanacağı projeye bağlı olarak az-çok zorla değiştirilmesi gerekebilir. Herşeye rağmen, modern gelişim sürecinin bir örneği olarak, Birleştirilmiş Süreç (Unified Process) olarak adlandıracağımız yöntemin göze çarpan kısımlarını inceleyeceğiz.

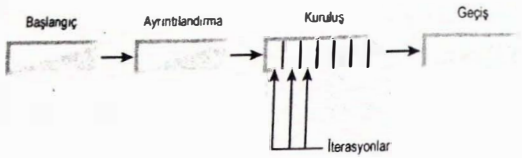
Birleştirilmiş Süreç, UML'i tasarlayan kişiler tarafından geliştirildi: Grady Booch, Ivar Jacobson ve James Rumbaugh. Bu yöntem kimi zaman Rasyonel Birleştirilmiş Süreç (yöntemin geliştirildiği şirketin isminden sonra yöntem bu şekilde anılmaya başlandı) veya Birleştirilmiş Yazılım Geliştirme Süreci olarak adlandırılır.

Birleştirilmiş Süreç dört aşamaya bölünür:

- Başlangıç
- Ayrıntılandırma
- Kuruluş
- Geçiş

Başlangıç aşamasında projenin genel olarak kapsamı ve fizibilitesi belirlenir. Bu aşama, yönetimin projeye devam onayını vermesi ile sona erer. Ayrıntılandırma aşamasında sistemin temel mimarisi tasarlanır. Kullanıcının ihtiyaçlarının belirlendiği aşama bu aşamadır. Kuruluş aşaması, yazılım tasarımını içeren ve kodun gerçek anlamda yazıldığı aşamadır. Geçiş aşamasında sistem test ve kurulum amacıyla kullanıcıya verilir.

Bu dört aşamanın her biri *iterasyonlar* (*iterations*) denilen birkaç parçaya ayrılabilir. Kuruluş aşaması özellikle birkaç iterasyondan oluşacaktır. Her iterasyon genel sistemin bir alt kümesidir ve kullanıcının programdan yürütmesini istediği belirli bir göreve karşılık gelir. (İleride göreceğimiz gibi, bir iterasyon genellikle bir use case'e karşılık gelir.) Birleştirilmiş Süreç Şekil 16.2'de gösterilmiştir.



ŞEKİL 16.2: Birleştirilmiş Süreç (Unified Process).

Her iterasyon kendisine ait analiz, tasarım, uygulama ve test zincirini içerir. Bu zincir birkaç kez iterasyon yapılabilir. Her iterasyonun amacı sistemin işleyen bir parçasını geliştirmektir.

Şelale (waterfall) yönteminden farklı olarak Birleştirilmiş Süreç, ilk aşamalara dönmeyi kolaylaştırır. Örneğin, geçiş aşamasında kullanıcının saptadığı değişiklikler, kuruluş aşamasının, hatta belki de ayrıntılandırma aşamasının yeniden gözden geçirilmesine neden olacaktır.

Birleştirilmiş Sürecin, yalnızca nesne yönelimli dillere değil, her tipte yazılım mimarisine uygulanabileceğini vurgulamalıyız. Aslında, bu yöntemin sahip olduğu muhtemel yetersizlik, bu yöntemin nesne yönelimli tasarımı etkin biçimde desteklememesidir.

Birleştirilmiş Sürecin ayrıntılandırma aşaması genellikle "*use case*" *modelleme* denilen bir teknikle başlar. Bu, sisteme uygun ayrıntılı bir tasarım geliştirmek için başlangıç noktasıdır. Bu nedenle, Birleştirilmiş Sürecin bir use case yönelimli (*use case driven*) süreç olduğu söylenebilir. Aşağıda, use case modellemeyi ele alacağız, ondan sonra da, bunu örnek bir yazılım projesine uygulayacağız.

Use Case Modelleme

Use case modelleme, bir yazılım sisteminin gelecekteki kullanıcılarına, sistemin tasarımıyla ilgili olabildiğince katkıda bulunma imkanı verir. Use case modelleme, kullanıcıların dilini konuşur; programcıların dilini kullanmaz. Kullanıcılar üzerine bu tür bir odaklanma, programın başlangıçtaki spesifikasyonlarının hem kullanıcılar tarafından hem de onu dizayn eden yazılım mühendisleri tarafından anlaşılabilir olmasını sağlar.

Use case yaklaşımında iki ana unsur vardır: *Aktörler* ve *use case*'ler. Şimdi bunların ne anlama geldiğini görelim.

Aktörler

Aktör, (genellikle) tasarlamakta olduğumuz sistemi kullanacak olan kişidir. Bir ATM makinesinin yazılımı ile etkileşmekte olan bir banka müşterisi bir aktördür. Teleskopu belirli bir konuma

hedefleyen bir programa bir yıldızın koordinatlarını giren bir gökbilimci de bir aktördür. Belirli bir kitabın mevcut olup olmadığını anlamak için bilgisayarı kontrol eden bir kitap satıcısı da bir aktördür. Genellikle bir aktör bazı işlemleri başlatır. Gerçi kimi zaman aktörün farklı şekilde davranması da söz konusu olabilir; mesela, bilgi almak veya bir işleme yardımcı olmak gibi.

Aslında, "rol" belki de "aktör"den daha iyi bir isimdir. Farklı rollerde bulunan bir kişi birkaç aktörle simgelenebilir. Örneğin, küçük bir işletmede, Harry Jones satışta bulunurken "satıcı" olarak adlandırılan bir aktör ile simgelenebilir; fakat, günün satış toplamını hesaplarken "kitapçı" olarak adlandırılan bir aktörle temsil edilebilir. Tam tersi de geçerlidir; tek bir aktör birkaç tane farklı kişiyi simgeleyebilir. Harry, Jose ve Elma, bu kişilerin tümü "satıcı" olarak adlandırılan bir aktör ile simgelenebilirler.

Bizim tasarlamakta olduğumuz sisteme bağlı diğer sistemler de, mesela farklı bir bilgisayar sistemi ya da Web'e bağlantı sağlayan bir köprü, ayrıca aktör olabilir. Örneğin, belirli bir kitapçıdaki bilgisayar sistemi, merkez olisteki (uzaktaki) sisteme bağlanmış olabilir. Bu uzaktaki sistem, kitapçı sistemindeki bir aktör olarak düşünülebilir.

Büyük bir projede aktörlerin tümünü belirlemek zor olabilir. Tasarımcının aşağıdaki görevleri yerine getiren kişileri veya sistemleri bulması gereklidir:

- Sisteme bilgi sağlamak.
- Sistemden bilgi talep etmek.
- Diğer aktörlere yardımcı olmak.

Use Case'ler

Bir use case genellikle bir aktör tarafından başlatılan spesifik bir görevdir. Use case, bir aktörün ulaşmaya çalıştığı tek bir amacı tanımlar. Banka müşterisinin nakit çekmesi, gökbilimcinin teleskobu hedefine ayarlaması ve kitap satıcısının kitabın mevcut olup olmadığını araştırması use case örneklerindendir.

Bir çok durumda bir use case bir aktör tarafından başlatılır; fakat, kimi zaman sistem tarafından da başlatılabilir. Elektrik şirketinin muhasebe programının faturayı ödemediğinize dair bir uyarı göndermesi veya arabanızın bilgisayarının araba motorunun fazlasıyla ısındığına karar verip ikaz lambasını yakması gibi durumlar sistem tarafından başlatılan use case'lere örnektir.

Genelde, sistemin gerçekleştirmesini istediğiniz her şey bir use case ile belirtilmelidir.

Senaryolar

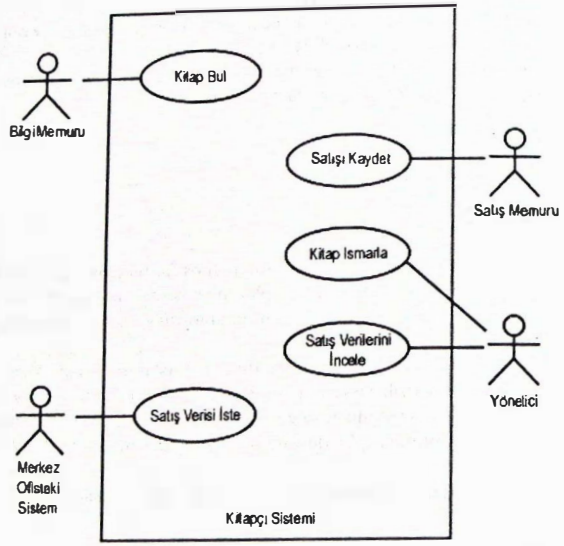
Bir use case genellikle birkaç *senaryodan* oluşur. Bir use case bir amacı belirtir; oysa, bir senaryo, bu amaca ulaşmaya çalışırken elde edilen belirli bir sonucu simgeler. Örneğin, bir use case'in belirli bir kitabın konumunu kitapevinin bilgisayar sisteminde sorgulayan bir kitap satıcısından ibaret olduğunu düşünelim. Bu durumda muhtemel birkaç sonuç veya senaryo söz konusudur:

- Kitap kitapevinde mevcuttur ve bilgisayar kitabın raftaki konumunu gösterir.
- Kitap stokta kalmamıştır, ama sistem, müşterinin kitabı yayınevinden ısmarlamasına fırsat verir.
- Kitap stokta kalmadığı gibi, önceki baskıları da tükenmiştir; bu durumda, sistem müşteriye şansının kalmadığını haber verir.

Kurallara uygun bir geliştirme sürecinde her senaryo, senaryodaki tüm olayları ayrıntılarıyla tarif eden kendisine ait bir dokümantasyon içerecektir.

Use Case Şemaları

UML, use case'lerin şematik olarak nasıl çizileceğini belirtir. Aktörler çöpten adam figürleri ile, use case'ler ise oval şekillerle simgelenir. Bir dikdörtgen çerçeve aktörleri dışarıda bırakarak use case'leri çevreler. Bu dikdörtgen *sistemin sınırlarını* çizer. İçeride kalan sistem, yazılımı geliştiricinin tasarlamaya çalıştığı sistemdir. Şekil 16.3 bir kitapevinin bilgisayar sistemi için bir use case şemasını gösteriyor.



ŞEKİL 16.3: Bir kitapevi için use case şeması.

Use case şemasında *bağlantılar* (*associations*) olarak adlandırılan çizgiler aktörleri use case'lerle birleştirirler. Çizgiler tipik olarak belirli bir yönü göstermez; fakat, use case'i kimin başlattığını belirtmek için ok uçları kullanılabilir.

Bu şemada, söz konusu kitapevinin bir kitapevi zincirinin halkası olduğunu ve muhasebe ve benzeri işlevlerin merkez ofiste kontrol altına alındığını varsayıyoruz. Mağazadaki tezgahtarlar satışı yapılan kitapların her birini kayda geçirirler ve bir kitabın konumu ve mevcut olup olmadığıyla ilgili bilgileri talep ederler. Yönetici hangi kitapların satıldığını kontrol edebilir ve yenilerini ısmarlayabilir. Bu sistemin aktörleri Satış Memuru, Bilgi Memuru, Yönetici ve Merkez Ofisteki Sistemdir. Use case'ler ise, Satışı Kaydet, Kitabı Bul, Kitap Ismarla, Satış Verilerini İncele ve Satış Verilerini İste'dir.

Use Case Tanımları

Use case şemasında use case'lerin tek tek ayrıntılı tanımları için yer yoktur; bu nedenle, tanımlar her use case ayrı ayrı tedarik edilmelidir. Bu use case şemalarında projenin boyutuna ve geliştiricilerin felsefesine bağlı olarak farklı formalite dereceleri kullanılır. Birçok durumda bir use case'deki senaryoların her birinin ayrıntılı birer tanımı da yer alacaktır.

Bir use case tanımının en basit versiyonu, bir paragraf uzunluğunda veya benzeri uzunlukta bir metinden ibarettir. Kimi zaman, aktörün faaliyetleri bir sütunda, sistemin tepkileri bir başka sütunda olacak şekilde iki sütun kullanılır. Biraz daha kurallara bağlı bir versiyon ise önkoşullar, koşul sonrası durumlar ve ayrıntılı bir adım zinciri gibi ayrıntıları içerebilir. Bir tür akış diyagramı olan ve *faaliyet şeması* olarak adlandırılan bir UML şeması kimi zaman bir use case'de peş peşe gelen adımları grafiksel olarak göstermek için kullanılır.

Use case şemaları ve use case tanımları esasen kullanıcılar ve geliştiriciler arasındaki iletişime yardımcı olmak amacıyla sistemin başlangıç tasarımında kullanılır. Bununla birlikte, bunlar geliştirme süreci boyunca da yardımcı olurlar. Sistemin gerçekleştirmesi gereken görevleri doğrulamak gereğinde olan herhangi bir kişi use case şemalarına ve tanımlarına başvurabilir. Use case şemaları ve tanımları ayrıca testler ve dokümantasyon için de bir temel teşkil eder.

Use Case'lerden Sınıflara

Aktörler ve use case'ler belirlendikten sonra geliştirme süreci ayrıntılandırma aşamasından kuruluş aşamasına ilerler. Odak noktası, kullanıcılardan geliştiricilere kayar. Öncelikle ilgilendiğimiz konu, programda kullanılacak olan sınıfları geliştirmektir.

Bir yaklaşım, use case tanımlarında kullanılan isimlere bakmaktır. Programdaki nesnelerin gerçek dünyadaki nesnelere karşılık gelmesini isteriz; bu isimler ise kullanıcı tarafından belirtilen, gerçek dünyaya ait unsurları simgelerler. Yani, bu isimler sınıf olarak tanımlanmaya aday ama, isimlerin tümü iyi bir sınıf oluşturmaz. Fazlasıyla genel olan, fazlasıyla belirgin olan veya nitelik olarak daha iyi temsil edilebilen (basit değişkenler) isimleri dikkate almamamız gerekir.

Birkaç tane sınıf adayını tespit eder etmez, artık bunların nasıl etkileştiklerini use case tanımlarındaki eylemleri inceleyerek anlamaya başlayabiliriz. Bir çok durumda bir eylem, bir nesneden diğerine gönderilen mesaj şekline veya sınıflar arası diğer bağlantılara dönüşür.

Sınıfları ve sınıfların ilişkilerini göstermek için bir UML sınıf şeması (önceki bölümlerde anlatılmıştı) kullanılabilir. Bir use case, nesneler arası bir mesaj zinciri ile gerçekleştirilir. Bu tür bir zincirin ayrıntılarına inmek için *etkileşim şeması* (*interaction diagram*) denilen bir başka UML şemasını kullanabiliriz. Aslında, bir use case içindeki her senaryo için ayrı bir etkileşim şeması kullanmayı tercih edebiliriz. İleride, bir tür etkileşim şeması olan sekans şemalarının (sequence diagrams) örneklerini göreceğiz.

Geliştirme sürecini bir örnek ile göz önünde canlandırmak daha kolaydır; bu nedenle, şimdi gerçek bir programın geliştirilmesini adım adım inceleyelim. Şartlar gereği bu program o kadar küçük ki, programın kurallara uygun bir geliştirme sürecini gerektirip gerektirmediği bile aslında tartışmalıdır. Yine de, bu süreci bu denli küçük bir projeye dahi uygulamak, bahsettiğimiz konular üzerindeki esrar perdesini kaldırmaya yardımcı olacaktır.

Programlama Problemi

Bu bölümde tasarlayacağımız program **LANDLORD** adını taşır. Ev sahibinizi seversiniz ya da sevmezsiniz, ancak ev sahibinin ilgilenmesi gereken çeşitli verileri (mesela, kira ve masraflar)

anlayabilirsiniz. Bu program, kolaylıkla anlaşılan *iş alanına (business domain)* (programı yazma nedenimiz) adım atmamızı sağlar.

Şimdi, diyelim ki siz bağımsız bir programcısınız ve Beverly Smith adında, müşteriniz olma potansiyeline sahip biri ile yakınlaşma içindesiniz. Beverly küçük çaplı bir ev sahibi: 12 dairesi olan bir apartmana sahip. Beverly, sizden apartmanıyla ilgili verileri kaydedecek ve apartmanın muhasebesiyle ilgili rapor hazırlayacak bir program yazmanızı istiyor. Eğer Beverly ve siz ödeme, ödeme takvimi ve programın genel amacı hakkında anlaşabilirseniz, geliştirme sürecinin başlangıç aşamasını tamamlamışsınız demektir.

Elle Doldurulan Formlar

Beverly, şimdilik, apartmanıyla ilgili tüm bilgileri eski moda muhasebe defterlerine elle kaydediyor. Beverly, size halen kullanmakta olduğu formları gösteriyor. Bu tür formlardan dört adet mevcuttur:

- Kiracı Listesi
- Aylık Kira Geliri Kayıtları
- Masraf Kayıtları
- Yıllık Gelir/Gider Tablosu

Kiracı Listesi, bir sütunda daire numaralarını, yan sütunda ise karşılık gelen kiracıların isimlerini gösterir.

Aylık Kira Geliri Kayıtları gelen kira ödemelerini kaydeder. Aylık Kira Geliri Kayıtları, her aya karşılık gelen 12 sütun ve her daireye karşılık bir satır içerir. Beverly bir kiracıdan ne zaman bir kira ödemesi alsa, Şekil 16.4'te gösterilen Aylık Kira Geliri Kayıtlarının ilgili satır ve sütununa bunu kaydeder.

Ayık Kira Geliri Kayıtları, hangi kira ödemelerinin gerçekleştirildiğini takip etmeyi kolaylaştırır.

Masraf Kayıtları yapılan ödemeleri kaydeder. Bu, şahsi çek kaydınıza benzer. Tarih, muhatap (alacaklının ismi - Beverly'nin çek yazdığı şirket ve şahıs) ve ödenecek miktar için ayrı sütunlar içerir. İlaveten, ödemenin yapılacağı bütçe kategorisi Beverly'nin belirtebilmesi için bir de ayrı sütun yer alır. Bütçe kategorileri arasında İpotek (Mortgage), Tamirat (Repairs), Yararlı Harcamalar (Utilities), Vergiler (Taxes), Sigorta (Insurance) vs yer alır. Masraf Kayıtları Şekil 16.5'te gösterilmiştir.

Yıllık Rapor, yıl içinde gelen ve giden paranın miktarını özetlemek amacıyla Aylık Kira Geliri Kayıtlarındaki ve Masraf Kayıtlarındaki verileri kullanır. Tüm kira gelirleri toplanır ve toplam gösterilir. Masraflar toplanır ve bütçe kategorisine göre gösterilir. Masrafları bu şekilde göstermek masrafların takibini kolaylaştırır, mesela yıl içinde tamiratlara ne kadar para harcandığını kolaylıkla görülebilir. Şekil 16.6, Yıllık Raporu göstermektedir. Son olarak, masraflar gelirlerden çıkarılarak Beverly'nin yıl içinde ne kadar para kazandığı (veya kaybettiği) görülebilir.

Beverly'nin mevcut sisteminde yıl sonu gelene kadar, Aralık ayı için de tüm kira ve harcamalar kaydedilene dek Yılık Rapor hazırlanmaz. Bizim bilgisayar tabanlı sistemimiz yılın herhangi bir zamanında kısmi bir Yıllık Rapor gösterebilecek beceride olmalıdır.

Beverly halen kağıt formlar üzerinde yaptıklarının neredeyse bir kopyasını çıkaracak bir program yazmanızı istediğini size bildirir. Beverly kiracılar, kiralar ve masraflarla ilgili verileri girebilmeyi ve çeşitli sonuçları ekranda gösterebilmeyi ister.

Aylık Kira Geliri Kayıtları

Apartment No	Jan	Feb	Mar	Apr	May	June	July	Aug
101	595	595	595	595	595			
102	595	595	595	595	595			
103	810	810	825	825	825			
104	720	720	720	720	720			
201	680	680	680	680	680			
202	510	510	510	530	530			
203	790	790	790	790	790			
* 204	495	495	495	495	495			
301	585	585	585	585	585			
302	530	530	530	530	550			
303	810	810	810	810	810			
304	745	745	745	745	745			

ŞEKİL 16.4: Masraf kayıtları.

Masraf Kayıtları

Date	Payee	Amount	Budget Category
1/3	Fine Mortgages	5187.30	Mortgage
1/8	City Water	943.10	Utilities
1/9	Stately Suits	4840.00	Insurance
1/15	P.G. & E.	727.53	Utilities
1/22	Sam's Hardware	84.81	Supplies
1/25	Ernie Ortiz	160.00	Repairs
2/3	Fine Mortgages	5187.30	Mortgage
2/7	City Water	845.93	Utilities
2/15	P.G. & E.	764.30	Utilities
2/18	Pete & Shorty	1200.00	Legal Fees
3/3	Fine Mortgages	5187.30	Mortgage
3/7	City Water	890.27	Utilities
3/10	County of Springfield	9437.00	Property Taxes
3/14	P.G. & E.	779.38	Utilities
3/20	Gotham Courier	28.40	Advertising
3/25	Ernie Ortiz	450.00	Repairs
3/27	Acme Printing	800.00	Maintenance
4/3	Fine Mortgages	5187.30	Mortgage

ŞEKİL 16.5: Aylık Kira Geliri Kayıtları.

Yıllık Gelir/Gider Tablosu

1		
2	INCOME	
3	Rent	102,384.00
4	TOTAL INCOME	102,384.00
5		
6	EXPENSES	
7	Mortgage	82,347.60
8	Property taxes	8,427.00
9	Insurance	6,840.00
10	Utilities	18,336.78
11	Supplies	1,188.33
12	Repairs	6,274.80
13	Maintenance	2,808.45
14	Legal fees	1,200.00
15	Landscaping	800.00
16	Advertising	78.84
17		
18	TOTAL EXPENSES	106,034.18
19		
20	NET PROFIT OR (LOSS)	(3,770.18)
21		

ŞEKİL 16.6: Yıllık Rapor.

Varsayımlar

Önceden elbette bazı kolaylaştırıcı varsayımlarda bulunduk. Aslında, bir apartmanı işletirken başka türlü veriler de söz konusu olur, mesela hasara karşı depozito, yıpranma payı, ipotek faizi, geç ödemelerden ve çamaşır makinelerinin kiralanmasından kaynaklanan gelirler gibi. Biz bu ayrıntıları dikkate almayacağız.

Beverly'nin isteyebileceği ayrıca başka türlü raporlar da olabilir, mesela Net Değer hesapları gibi. Hatta, programın bir gelir vergisi programı ile ve online bankacılık sistemi ile bir arayüzünün olması hoş olurdu. Ayrıca, resme daha geniş bir perspektiften bakınca, ticari mülk sahibi programlarının piyasada mevcut olduğunu görmekteyiz; bu nedenle, Beverly'nin ısmarlama yazılmış bir programa sahip olmak için sözleşme yapması akıllıca olmayabilir. Programın daha kolay kontrol edilebilir olması için tüm bu dikkat dağıtıcı unsurları bir kenara bırakacağız.

LANDLORD Programının Ayrıntılandırma Aşaması

Başlı başına bir yazılım geliştirme projesinin ayrıntılandırma aşamasında, programın potansiyel kullanıcılarından ve programı tasarlayan yazılımcılarından oluşan bir grup insan programın ne yapacağını tartışmak üzere toplanırlar. Bu küçük örnekte grup, sistemi kullanacak olan Beverly'den ve sistemi hem tasarlayacak hem de kodlayacak olan sizden oluşuyor.

Aktörler

Grup, aktörleri belirleyerek işe başlar. Programa girdileri kim girecek? Bilgileri kim talep edecek? Başka birileri de programla etkileşecek mi? Program başka programlarla veya sistemlerle etkileşecek mi?

LANDLORD örneğinde programı sadece tek bir kişi kullanacaktır: Ev (mülk) sahibi. Aynı kişi bilgileri girecek ve bu bilgilerin çeşitli yöntemlerle gösterilmesini görmek isteyecektir.

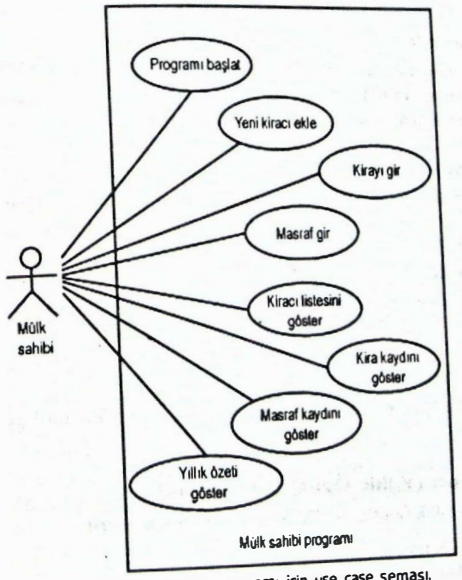
Bu kadar küçük bir projede bile diğer aktörler akla gelebilir. Ev sahibinin muhasebecisi eğer programın verilerine erişmek isterse (belki de Internet üzerinden), muhasebeci de bir aktör olacaktır; ayrıca, program bir gelir vergisi programına veri sağlamışsa o program da bir aktör olur. İşleri basitleştirmek adına bu ihtimalleri dikkate almayacağız.

Use Case'ler

Grup sonra, aktörlerin yürütmek isteyeceği görevlerin neler olabileceğini düşünür. Gerçek bir yazılım projesinde bu, birçok kullanıcıdan girdilerin geldiği, bir çok tartışmanın yürütüldüğü ve birçok fikrin yeniden gözden geçirildiği başlı başına emek gerektiren bir aşama olacaktır. Buradaki örnekte, ev sahibinin ihtiyaç duyduğu görevleri listelemek çok karmaşık değildir. Bu görevler use case şemasında kaydedilmiştir.

Bizim örneğimizde, ev sahibi aktörünün aşağıdaki görevleri yapması gerekiyor:

- Programı başlat
- Kiracı Listesine yeni bir kiracı ekle
- Kira Kayıtlarına kirayı gir
- Masraf Kayıtlarına masrafı gir
- Kiracı Listesini göster
- Kira Kayıtlarını göster
- Masraf Kayıtlarını göster
- Yıllık Gelir/Gider Tablosunu göster
- Bunun neticesinde elde edilen use case şeması Şekil 16.7'de gösteriliyor.



ŞEKİL 16.7: **LANDLORD** programı için use case şeması.

Use Case Tanımları

Artık her use case'i ayrıntılı olarak tarif etmemiz gerekir. Önceden bahsedildiği gibi, use case tanımları oldukça kurallı ve karmaşık olabilir. Ancak, bizim projemiz o kadar basit ki, tüm ihtiyacımız olan kısa, düz yazı şeklindeki tanımlardan ibarettir.

Programı Başlat

"Programı Başlat", bahsetmek için fazlasıyla bariz görünebilir. Ancak, program çalışmaya başladığında kullanıcının, gerçekleştirilmek üzere bir görev seçebileceği bir ekran kullanıcıya gösterilmelidir. Buna Kullanıcı Arayüz ekranı denilebilir.

Yeni Bir Kiracı Ekle

Program, Kiracı Giriş ekranını sunar. Bu ekran, kullanıcıdan yeni kiracı ismi ve daire numarası girmesini ister. Program sonra, bu bilgileri Kiracı Listesinde yeni bir satıra yerleştirir. Bu liste, daire numarasına göre otomatik olarak sıralanmıştır.

Kira Ödemesini Gir

Kira Giriş ekranı, kullanıcıyı kiracının ismini, aylık kirayı ve alınan kira miktarını girmeye yönlendirir. Program, kiracının ismini Kiracı Listesinde arar ve Kira Kayıtlarına erişmek için karşılık gelen daire numarasını kullanır. Eğer bu, kiracının ödediği ilk kira ise Kira Kayıtlarında yeni bir satır açılır ve kira miktarı ilgili aya eklenir. Aksi halde, kira mevcut satıra eklenir.

Masraf Ödemesini Eklemek

Masraf Giriş ekranı, kullanıcının, muhatabı (ev sahibinin ödeme yaptığı kişi ve kurum), ödenen miktarı, ödemenin yapıldığı gün ve ayı ve bütçe kategorisini girmesini ister. Program sonra, bu bilgiyi içeren bir satır oluşturur ve bunu Masraf Kayıtlarına ekler.

Kiracı Listesini Göstermek

Program, her satırında daire numarasını ve kiracının ismini içeren Kiracı Listesini ekranda gösterir.

Kira Kayıtlarını Göstermek

Program, her satırında daire numarasını ve aylık ödenen miktarı içeren Kira Kayıtlarını ekranda gösterir.

Masraf Kayıtlarını Göstermek

Program, her satırında ay, gün, muhatap, meblağ ve bütçe kategorisini içeren Masraf Kayıtlarını ekranda gösterir.

Yıllık Gelir/Gider Tablosunu (Yıllık Özet) Göstermek

Program, aşağıdaki unsurları içeren Yıllık Gelir/Gider Tablosunu ekranda gösterir:

1. Bugüne kadar yıl içinde ödenen tüm kiraların toplamı
2. Her bütçe kategorisi için toplam harcamaların listesi
3. Sonuçta ortaya çıkan bütçe (bugüne kadar yıl içinde kar veya zarar durumları)

Senaryolar

Önceden bahsettiğimiz gibi, bir use case birkaç senaryodan meydana gelebilir. Şimdiye kadar her use case için sadece ana senaryodan bahsettik. Bu, her şeyin kusursuzca çalıştığı ve hedefe ulaşılan senaryodur. Bununla birlikte, diğer sonuçlar da sıkça ortaya çıkar. Yeni Kiracı Ekle use case'inde ikinci bir senaryoya örnek olarak, kullanıcının zaten dolu olan bir daireye yeni bir kiracı eklemeye çalıştığını varsayalım.

Yeni Bir Kiracı Ekle, İkinci Senaryo

Program, Kiracı Giriş ekranını sunar. Bu ekran, kullanıcıdan yeni kiracı ismi ve daire numarası girmesini ister. Ancak, daire numarası önceden Kiracı Listesine girilmiştir, yani daire bir başkasına kiralanmıştır. Kiracı Ekle ekranı bu duruma karşılık bir hata mesajı verir.

İşte ikinci bir senaryo için başka bir örnek daha. Bu kez kullanıcı mevcut olmayan bir kiracı için kira ödemesini girmeye çalışıyor.

Kira Ödemesini Gir, İkinci Senaryo

Kira Giriş ekranı, kullanıcıyı kiracının ismini, aylık kirayı ve alınan kira miktarını girmeye yönlendirir. Program kiracını ismini Kiracı Listesinde arar, fakat bulamaz. Program kullanıcıya bir hata mesajı gösterir.

İşleri basitleştirmek amacıyla bu tür alternatif senaryolara devam etmeyeceğiz. Oysa, gerçek bir projede her senaryo, ana senaryo kadar ayrıntılı biçimde geliştirilmelidir. Yalnızca bu şekilde davranarak programlama elemanlarının tümü ortaya çıkarılabilir.

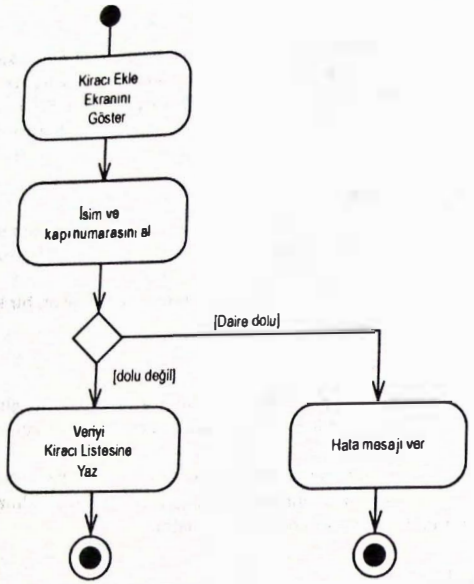
UML Faaliyet Şemaları

UML faaliyet şemaları use case'leri modellemek için kullanılabilir. Bu tür bir şema, bir faaliyetten diğerine kontrol akışını gösterir. Faaliyet şeması, programlamanın doğuşundan beri ortalıkta olan akış şemasına (flowchart) benzer. Ancak, faaliyet şeması, tıpkı diğer UML şemaları gibi, daha kurallı olarak belirtilir ve ilave becerilere sahiptir.

Faaliyetler, hapa benzeyen bir şekil ile gösterilirler. Faaliyetleri birleştiren çizgiler bir faaliyetten diğerine geçişleri simgelerler. Dallanmalar, bir gelen ve iki veya daha fazla terk eden geçişe sahip baklava şekliyle gösterilir. Durum şemalarında olduğu gibi, bu geçişlerden hangisinin seçileceğini belirtmek için geçişler üzerine koruyucular yerleştirebilirsiniz. Ayrıca, durum şemalarında olduğu gibi burada da, bir başlangıç durumu ve bir de son durumu mevcuttur. Başlangıç durumu içi dolu bir daire ile, son durumu ise bir çember içine yerleştirilmiş içi dolu bir daire ile simgelenir.

Şekil 16.8, Yeni Bir Kiracı Ekle use case'ini az önce bahsettiğimiz ikinci senaryoyla birlikte gösteriyor. Dallanma, kullanıcı tarafından girilen daire numarasının önceden dolu olup olmadığına bağlıdır. Eğer doluysa, bir hata mesajı verilir.

Faaliyet şemaları da, tıpkı akış şemaları gibi, program kodu içindeki karmaşık algoritmaları temsil etmek için de kullanılabilir. Faaliyet şemalarının burada ele almayacağımız bazı becerileri daha vardır; mesela, birkaç eş zamanlı faaliyeti temsil etmek gibi.



ŞEKİL 16.8: UML faaliyet şeması.

Use Case'lerden Sınıflara

Programı tasarlamaya başladığımız andan itibaren projemizin kuruluş aşaması da başlar. Önceden bahsedildiği gibi, use case tanımlarında kullanılan isimleri inceleyerek işe başlayacağız.

İsimleri Listelemek

Use case tanımlarından çıkarılan isimlerin listesi şöyledir:

1. Kullanıcı Arayüz ekranı
2. Kiracı
3. Kiracı Giriş ekranı
4. Kiracının ismi
5. Daire numarası
6. Kiracının sırası
7. Kiracı Listesi
8. Kira Ödemesi
9. Kira Giriş ekranı
10. Ay
11. Kira miktarı
12. Kira Kayıtları
13. Kira satırı

14. Masraf ödemeleri
15. Masraf Giriş ekranı
16. Muhatap (alacaklı)
17. Masraf miktarı
18. Gün
19. Bütçe kategorisi
20. Masraf satırı
21. Masraf Kayıtları
22. Yıllık Gelir/Gider Tablosu (Yıllık Özet)
23. Kira toplamı
24. Kategori bazında toplam masraflar
25. Bütçe

Listeyi Gözden Geçirmek

Çeşitli nedenlerden ötürü, birçok isim uygun birer sınıf adayı değildir. Şimdi hangi isimlerin kabul edilmemesi gerektiğini görelim.

Çeşitli kayıtlara ait satırları da listeye dahil ettik: Kiracı satırı, kira satırı ve masraf satırı. Kimi zaman bu satırlar iyi birer sınıf oluştururlar, çünkü karmaşıklardır veya karmaşık veriler içerirler. Ancak, Kiracı Kayıtlarındaki her satır tam olarak bir kiracı için veri tutar, Masraf Kayıtlarındaki her satır ise tam olarak bir harcama için verileri saklar. Kiracı ve masraf sınıfları zaten mevcut olduğu için aynı isimde iki sınıfa ihtiyaç olmayacağını tahmin ediyoruz ve kiracı satırı ve masraf satırı sınıflarını göz ardı ediyoruz. Kira satırı ise, öte yandan, daire numarasını ve 12 kiradan oluşan bir diziyi içerir. Yılın ilk kirası ödenene kadar kira satırı mevcut olmaz; ilk ödemenin peşinden, kiralar mevcut satıra eklenir. Bu, kiracılara ve masraflara kıyasla daha karmaşık bir durum olduğu için kira satırını bir sınıf olarak bırakacağız. Ancak bu durum kira ödeme sınıfını, kira miktarı hariç verisiz bıraktığı için bu sınıfı eleyeceğiz.

Program, Kira Kayıtlarından ve Masraf Kayıtlarından Yıllık Gelir/Gider Tablosundaki verileri elde edebileceği için kira toplamı, kategori bazında toplam masraflar ve bütçeden sınıf oluşturmamıza muhtemelen gerek kalmayacaktır. Bunlar sadece bazı hesaplamaların sonuçlarından ibarettir.

Böylece geriye şu sınıflar kalır:

1. Kullanıcı Arayüz ekranı
2. Kiracı
3. Kiracı Giriş ekranı
4. Kiracı Listesi
5. Kira Giriş ekranı
6. Kira Kayıtları
7. Kira satırı
8. Masraf ödemeleri
9. Masraf Giriş ekranı
10. Masraf Kayıtları
11. Yıllık Gelir/Gider Tablosu (Yıllık Özet)

Nitelikleri Keşfetmek

Sınıf olarak elediğimiz isimlerin birçoğu sınıf içindeki nitelikler (üye veriler) için aday olacaktır. Örneğin, Kiracı sınıfı, Kiracının İsmi ve Daire Numarası niteliklerini; Masraf sınıfı ise Alacaklı, Ay, Gün, Miktar ve Bütçe Kategorisi niteliklerini içerecektir. Niteliklerin büyük çoğunluğu bu şekilde tespit edilir.

Eylemlerden Mesajlara

Şimdi, use case'lerin bir sınıfın bir başka sınıfa gönderdiği mesajları nasıl aydınlattığını anlamak için use case'lere bir göz atalım. Bir mesaj aslında bir nesne içindeki bir üye fonksiyona yapılan bir çağrı olduğu için mesajları tespit etmek, mesajı alan sınıfın üye fonksiyonlarını keşfetmekle aynıdır. İsimlerde olduğu gibi, her eylem mesaj için bir aday değildir. Bazıları, mesaj için aday olmak yerine, kullanıcıdan bilgi almak, bilgileri ekranda göstermek ve diğer işleri gerçekleştirmekle ilgilidir.

Örnek olarak, şimdi Kiracı Listesini Göster use case'ine, eylemlerin altı çizili olduğu halde bakalım:

Program, her satırında daire numarasını ve kiracının ismini içeren Kiracı Listesini ekranda gösterir.

"Program" ile kast ettiğimiz aslında Kullanıcı Arayüz ekranıdır. Bu nedenle, "gösterir" demek, Kullanıcı Arayüz ekranının Kiracı Listesine bir mesaj göndererek – yani, Kiracı Listesinin bir üye fonksiyonunu çağırarak – kendisini göstermesini bildirmesi demektir. Üye fonksiyonun `display()` gibi bir isimle isimlendirileceğini tahmin edebilirsiniz.

"içeren" eylemi, bir mesaja karşılık gelmez; bu sadece Kiracı Listesindeki bir satırın içeriğini tarif eder.

Şimdi, daha karmaşık bir örneğe göz atalım: Yeni Bir Kiracı Ekle use case'i:

Program, Kiracı Giriş ekranını sunar. Bu ekran, kullanıcıdan yeni kiracı ismi ve daire numarası girmesini ister. Program sonra, bu bilgileri Kiracı Listesinde yeni bir satıra yerleştirir. Bu liste, daire numarasına göre otomatik olarak sıralanmıştır.

"sunar" eyleminin anlamı şudur: Kullanıcı Arayüz ekranı, Kiracı Listesine kendi içeriğini göstermesini ve kullanıcıdan verileri almasını bildiren bir mesaj gönderir. Bu mesaj, Kiracı Giriş ekranında `getTenant()` gibi bir isme sahip bir üye fonksiyona yapılan çağrı olabilir.

"girmesini" ve "ister" eylemlerinin her ikisi de Kiracı Giriş ekranının kullanıcı ile haberleşmesiyle bağlantılıdır. Bu eylemler, nesne yönelimli yaklaşım açısından mesajlara karşılık gelmez. Daha doğrusu, zaten `getTenant()`, kullanıcıyı girdiye yönlendiren komutları ekranda gösterir ve kullanıcının yanıtlarını (kiracının ismi ve daire numarası) kaydeder.

"yerleştirir" eylemi, Kiracı Giriş ekranının, muhtemelen yeni bir Kiracı nesnesini argüman olarak kullanarak Kiracı Listesine bir mesaj göndermesi anlamına gelir. Kiracı Listesi daha sonra bu yeni nesneyi listeye ekleyebilir. Bu fonksiyonun ismi `insertTenant()` olabilir.

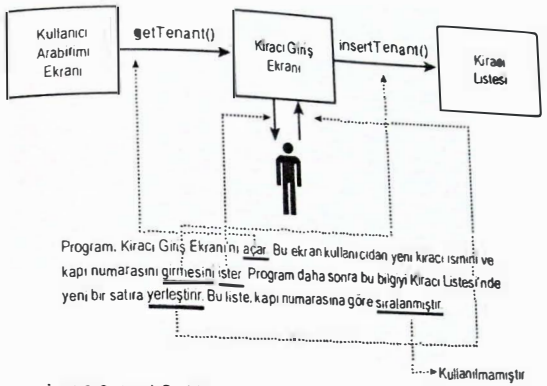
"sıralanmıştır" eylemi ise ne bir mesaj ne de bir tür iletişimdir. Bu sadece Kiracı Listesini tarif eden bir tanımdır.

Şekil 16.9 Yeni Bir Kiracı Ekle use case'ini ve bu use case'in bu mesajlarla bağlantısını gösteriyor.

Kodu yazmaya başladığımızda, use case'lerde bahsedilmeyen ama program için gerekli olan bazı faaliyetlerin olduğunu fark edeceğiz. Örneğin, use case'ler bir Kiracı nesnesinin oluşturulması hakkında hiçbir bilgi vermezler. Ancak, Kiracı Listesinin Kiracı nesnelerini tuttuğu ve Kiracı nesnesinin listeye yerleştirilmeden önce oluşturulmuş olması gerektiği, sanırız,

apaçık ortadadır. Yazılım mühendisi, Kiracı Giriş ekranındaki `getTenant()` üye fonksiyonunun, Kiracı Listesine eklenecek Kiracı nesnesini tanımlamak için uygun bir yer olduğuna karar verir.

Sınıflar arası ilişkiler hakkında ipuçları elde etmek için diğer use case'ler de benzer şekilde analiz edilebilir. Dikkat ederseniz, bu noktada sınıf isimleri hâlâ use case'lerde göründuğü gibi kullanıyoruz. Kodu yazmaya başladığınızda bunları tek kelimelik C++ sınıf isimleri olarak yeniden yazmamız gerekecektir.



ŞEKİL 16.9: Yeni Bir Kiracı Ekle use case'indeki eylemler.

Sınıf Şeması

Bir kez, hangi sınıflara ihtiyacımız olduğu hakkında ve bu sınıfların birbirleriyle nasıl bağlantılı olduğu hakkında bir fikir edindikten sonra, artık bir sınıf şeması tasarlayabiliriz. Sınıf şeması örneklerini önceki bölümlerde görmüştük. `LANDLORD` programının sınıf şeması Şekil 16.10'da gösteriliyor.

Sekans Şemaları

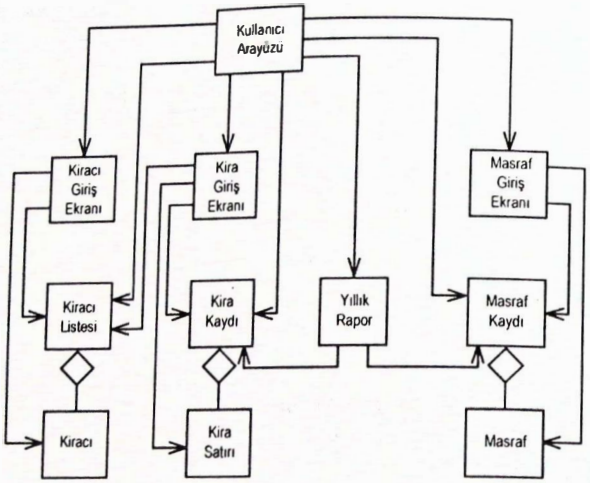
Kodu yazmaya başlamadan önce her use case'in içerdiği adımları daha ayrıntılı olarak anlamak isteyebilirsiniz. Bunu gerçekleştirmenin bir yolu, bir UML *sekans* (sequence) *şeması* üretmektir. Sekans şeması, iki çeşit UML etkileşim şemasından biridir. Diğer etkileşim şeması *işbirliği* (collaboration) *şemasıdır*. Her ikisi de, zaman içinde olayların nasıl ortaya çıktığını gözler önüne serer. Fakat sekans şeması, zamanı daha grafiksel bir yöntemle gösterir.

Bir sekans şemasında, en üstten başlayıp aşağıya doğru uzanan düşey eksen zamanı temsil eder. Tepelerde, use case'de yer alacak olan nesnelerin isimlerini içeren dikdörtgenler mevcuttur. Faaliyet genellikle soldaki nesnenin, sağındaki nesneye bir mesaj göndermesiyle başlar. Sağa doğru gittikçe, genellikle daha düşük öneme sahip (ya da daha çok bağımlı) nesneler yer alır.

Dikkat ederseniz şema, sınıfları değil, nesneleri gösterir. Biz, mesaj zincirlerine ve bir nesneden bir başka nesneye gönderilen mesajlara odaklanacağız. Bir sınıftan başka bir sınıfa

gönderilen mesajları dikkate almayacağız. UML şemalarında nesne isimlerini sınıf isimlerinden ayırt etmek için nesne isimlerinin altı çizilir.

Nesnelerin her birinden aşağıya doğru uzanan kesikli çizgi *yaşam çizgisi* (lifeline) olarak adlandırılır. Bu, nesnenin belirli bir süre için mevcut olduğunu gösterir. Eğer nesne ortadan kaldırılırsa, nesnenin yaşam çizgisi o noktada durur.



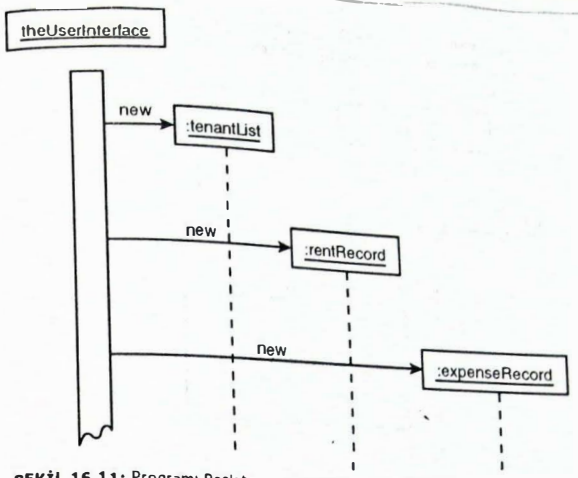
ŞEKİL 16.10: `LANDLORD` programının sınıf şeması.

"Programı Başlat" için Sekans şeması

Şimdi, `LANDLORD` programının birkaç sekans şemasına göz atalım. Kolay bir tanesiyle başlayalım. Şekil 16.11'de Programı Başlat use case'inin sekans şemasını gösteriliyor.

Program başlar başlamaz, use case'de bahsedilen Kullanıcı Arayüz ekranını gerçekleştirmek için `userInterface` adında bir sınıf tanımlar. Programın bu sınıfa ait `theUserInterface` adında tek bir nesne oluşturduğunu varsayalım. Use case'lerin tümüne bu nesne, başlangıç değerleri verecektir. Sekans şemalarında bu nesne, sol tarafta görünecektir. (Bu şemalarda use case isimlerini sıkıştırarak C++ sınıf isimleri şeklinde kullanıyoruz.)

`theUserInterface` nesnesi çalışmaya başlayınca ilk iş olarak programdaki üç temel veri yapısını tanımlar. Bunlar `tenantList`, `rentRecord` ve `expenseRecord` sınıflarının nesneleridir. Sonuçta, program bu nesneleri `new` ile oluşturur, yani nesneler isimsiz doğmuşlardır; sadece bunları gösteren işaretçilerin isimleri vardır. Peki, bu nesneleri nasıl çağıracağız? Neyse ki, nesne şemalarında gördüğümüz gibi UML, nesne isimlerini yazmak için birkaç değişik yöntem sunar. Nesnenin gerçek ismini bilmiyorsanız, iki nokta işaretini ve sınıf ismini kullanabilirsiniz: `:tenantList`. Şemalarda alt çizgi ve iki nokta işareti, ismin bir sınıfa değil, bir nesneye uygulandığını hatırlatır.



ŞEKİL 16.11: Programı Başlat use case'inin sekans şeması.

Nesne dikdörtgenlerinin düşey konumları, `tenantList` sınıfının nesnesinden başlayarak nesnelerin oluşturulduğu zamanı gösterir. Bu nesnelerin tümü programın ömrü boyunca mevcut olmaya devam edeceklerdir; bu nedenle, yaşam çizgileri şemanın alt tarafına kadar aşağı yönde uzanır. Zaman boyutunun ölçeklenmesine gerek yoktur; bu sadece çeşitli olaylar arasındaki ilişkileri göstermek amacıyla tasarlanmıştır.

Yatay çizgiler mesajları (üye fonksiyonlara çağrıları) simgeler. İçi dolu ok ucu, normal eş zamanlı (senkronize) bir fonksiyon çağrısına işaret eder. (İçi boş ok ucu ise asenkron bir olayı belirtir.)

`theUserInterface`'in altındaki dikdörtgen *faaliyet kutusu* (veya *kontrol odağı*) olarak adlandırılır. Bu, nesnesinin *aktif* olduğunu gösterir. `LANDLORD` gibi normal bir prosedürel programda "aktif" şu anlama gelir: Söz konusu nesneye ait bir üye fonksiyon ya kendisi çalışıyordur ya da bir başka fonksiyonu çağırmıştır ve bu fonksiyon henüz dönmemiştir. Bu şemadaki diğer üç nesne aktif değildir, çünkü `theUserInterface` henüz bu nesnelere ne yapmaları gerektiğini bildiren bir mesaj göndermez.

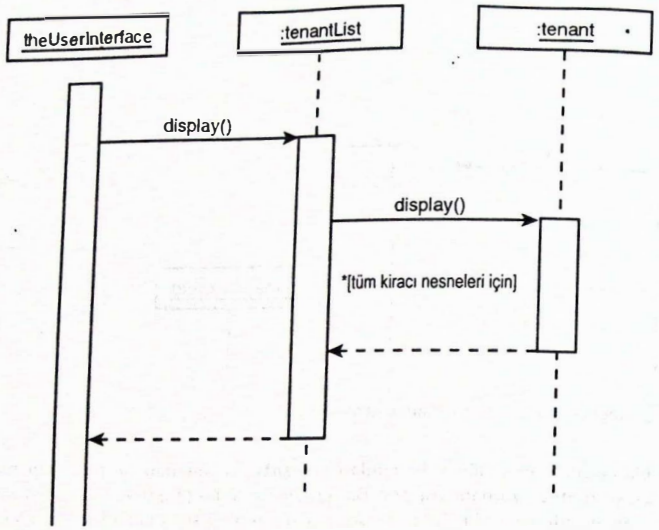
"Kiracı Listesini Göster" için Sekans şeması

Şimdi, bir başka sekans şemasını inceleyelim. Bu kez, Şekil 16.12'de gösterilen Kiracı Listesini Göster use case'ini ele alacağız.

Fonksiyon dönüşleri kesikli çizgilerle simgelenir. Nesnelerin sadece, üye fonksiyonlarından biri bir başka nesne tarafından çağrıldığı zaman aktif olduklarına (yaşam çizgileri üzerinde bir faaliyet kutusuna sahipler) dikkat edin. Mesaj çizgileri, çağrılmakta olan üye fonksiyonun ismini gösterebilir.

Burada, `theUserInterface`, `tenantList` nesnesine içeriğini göstermesini bildirir ve `tenantList` listesi de, sırayla, sahip olduğu tüm `tenant` nesnelerine içeriklerini göstermelerini bildirir. Asteriks, mesajın tekrar tekrar gönderileceğini; köşeli parantez içindeki sözcük öbek-

leri ise (mesela `[for all tenant objects]`), bu iterasyonun koşulunu belirtir. (Programda, burada gösterilen `display()` fonksiyonu yerine aslında `cout <<` kullanacağız.)



ŞEKİL 16.12: Kiracı Listesini Göster use case'inin sekans şeması.

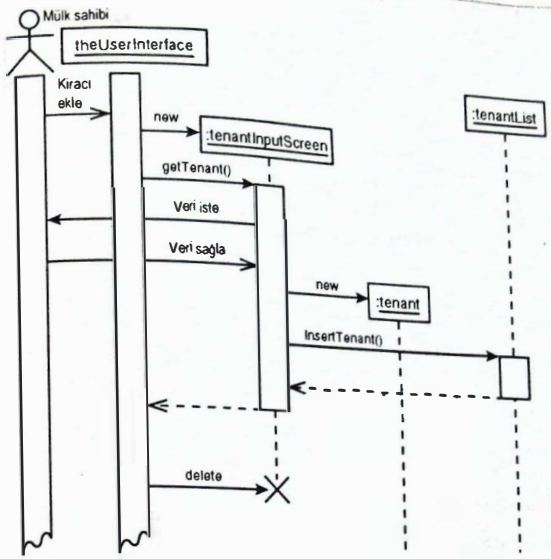
"Yeni Bir Kiracı Ekle" için Sekans şeması

Sekans şemalarımızın son örneği olarak şimdi de Yeni Bir Kiracı Ekle use case'ine bakalım. Bu örnek Şekil 16.13'te gösterilmiştir. Burada, ev (mülk) sahibini de bir nesne olarak, kendi faaliyet kutusuyla birlikte, şemaya dahil ettik. Bu sayede, program ve kullanıcı arasındaki etkileşimi göstermemiz mümkün olur.

Kullanıcı, yeni bir kiracı eklenmesi gerektiğini programa bildirir. `theUserInterface` nesnesi; `tenantInputScreen` sınıfına ait yeni bir nesne oluşturur. Bu nesne, kiracının verilerini kullanıcıdan alır, yeni bir `tenant` nesnesi tanımlar ve yeni tanımlanan kiracıyı eklemek için `tenantList` sınıfının nesnesini çağırır. Görevini tamamlayınca, `tenantInputScreen` nesnesini ortadan kaldırır. `tenantInputScreen`'in yaşam çizgisinin sonundaki büyük "X" işareti bu nesnenin bu noktada yok edildiğini gösterir.

Burada gösterdiğimiz sekans şemaları sadece use case'lerin ana senaryoları ile ilgilidir. Sekans şemaları üzerinde alternatif senaryolar göstermek için yöntemler mevcuttur, ya da her senaryo için yeni bir şema tasarlayabilirsiniz.

Yerimizin dar oluşu tüm use case'ler için ayrı sekans şemaları çizmemize engel oluyor; fakat, bu şemaları çizmek isterseniz, şu aşamada kendi kendinize tasarlamanıza yetecek kadar bilgi birikimine sahipsiniz.



ŞEKİL 16.13: Yeni Bir Kiracı Ekle use case'inin sekans şeması.

Kodu Yazmak

Use case şemaları, ayrıntılı use case'ler, sınıf şemaları ve sekans şemalarıyla donandıktan sonra nihayet derleyicinizi harekete geçirip gerçek kodu yazmaya başlayabilirsiniz. Bu, kuruluş aşamasının ikinci bölümüdür.

Ayrıntılandırma aşamasında belirlenen use case'ler kuruluş aşamasında iterasyonlara dönüştürülür. (Şekil 16.2'ye bakın.) Büyük bir projede bu iterasyonların her biri farklı bir programcı ekibi tarafından kontrol edilebilir. Her iterasyon ayrı ayrı geliştirilir ve değişiklikleri ve düzeltmeleri saptamaları için kullanıcılara geri gönderilir. Bizim küçük programımızda bu tür karmaşıklıklara gerek yoktur.

Başlık Dosyası

Kodlamaya başlamak için en uygun yer, sınıf uygulamalarıyla ilgili ayrıntılar yerine sınıf arayüzlerini tanımladığınız `.H` dosyasıdır. Önceden bahsettiğimiz gibi, `.H` dosyasındaki deklarasyonlar sınıfların public, yani bu sınıfı kullanan kullanıcıların gördükleri parçalarıdır. `.CPP` dosyasındaki fonksiyon gövdeleri, sınıf kullanıcıların erişmemesi gereken uygulamalardır.

`.H` dosyasının oluşturulması, tasarım ve yöntem gövdelerinin ayrıntılı yazılımı arasında bir ara basamaktır. `LANDLORD.H` dosyası aşağıdaki gibidir:

```
//landlord.h
//landlord.cpp icin baslik dosyasi - sinif deklarasyonlarini vs icerir
#pragma warning (disable:4786)   //set icin (sadece Microsoft icin)
```

```
#include <iostream>
#include <vector>
#include <set>
#include <string>
#include <algorithm>            //sort() icin
#include <numeric>              //accumulate() icin
using namespace std;
//////////////////////////global yontemler////////////////////////
void getaLine(string& inStr);   //metin satirini al
char getaChar();                //bir karakter al

//////////////////////////class tenant//////////////////////////
class tenant
   {
   private:
   string name;                 //kiracinin ismi
   int aptNumber;               //kiracinin daire numarasi
   //kiraciyla ilgili diger bilgiler (telefon no vs) burada yer alabilir

   public:
   tenant(string n, int aNo);
   ~tenant();
   int getAptNumber();
   //'kume'de kullanilmak icin gerekli
   friend bool operator < (const tenant&, const tenant&);
   friend bool operator == (const tenant&, const tenant&);
   //I/O icin
   friend ostream&operator << (ostream&, const tenant&);
};    //class tenant'in sonu
//////////////////////class compareTenants//////////////////////
class compareTenants            //fonksiyon nesnesi - kiracilari karsilastirir
   {
   public:
   bool operator () (tenant*, tenant*) const;
   };

//////////////////////class tenantList//////////////////////
class tenantList
   {
   private:
   //kiracilara isaret eden isaretciler kumesi
   set<tenant*, compareTenants> setPtrsTens;
   set<tenant*, compareTenants>::iterator iter;

   public:
   ~tenantList();                   //yok edici fonk. (kiracilari siler)
   void insertTenant(tenant*);      //kiraciyi listeye koy
   int getAptNo(string);            //daire numarasini dondur
   void display();                  //kiraci listesini goster
   };    //class tenantList'in sonu

//////////////////////class tenantInputScreen//////////////////////
class tenantInputScreen
   {
   private:
   tenantList* ptrTenantList;
   string tName;
   int aptNo;
```



```
   public:
   tenantInputScreen(tenantList* ptrTL) : ptrTenantList(ptrTL)
      { /*bos */ }
   void getTenant();
   };    //class tenantInputScreen'in sonu

//////////////////////class rentRow//////////////////////
//kira kaydının bir satiri:bir adres ve 12 adet kira tutari
class rentRow
   {
   private:
   int aptNo;
   float rent[12];

   public:
   rentRow(int);
   void setRent(int, float);        //1-argumanli kurucu fonk.
   float getSumOfRow();             //bir aylik kirayi kaydet
   // 'kume'de saklamak icin gerekli //satirdaki kiralarin toplamini dondur
   friend bool operator < (const rentRow&, const rentRow&);
   friend bool operator == (const rentRow&, const rentRow&);
   //cikti icin
   friend ostream& operator << (ostream&, const rentRow&);
   };    //class rentRow'un sonu

//////////////////////////////////////////////////////////////////
class compareRows               //fonksiyon nesnesi - kira satirlarini karsilastirir
   {
   public:
      bool operator () (rentRow*, rentRow*) const;
   };

//////////////////////class rentRecord//////////////////////
class rentRecord
   {
   private:-
   //rentRow nesnelerini gosteren isaretciler kumesi (kiraci basina bir)
   set<rentRow*, compareRows> setPtrsRR;
   set<rentRow*, compareRows>::iterator iter;

   public:
   ~rentRecord();
   void insertRent(int, int, float);
   void display();
   float getSumOfRents();           //kayittaki tum kiralari topla
   };    //class rentRecord'in sonu

//////////////////////class rentInputScreen//////////////////
class rentInputScreen
   {
   private:
   tenantList* ptrTenantList;
   rentRecord* ptrRentRecord;

   string renterName;
   float rentPaid;
   int month;
   int aptNo;
```

```
        public:
        rentInputScreen(tenantList* ptrTL, rentRecord* ptrRR) :
                        ptrTenantList(ptrTL), ptrRentRecord(ptrRR)
           { /*bos*/ }
        void getRent();                  //bir kiraci icin bir aylik kira
        };    //class rentInputScreen'in sonu

    ////////////////////////////class expense/////////////////////////
    class expense
       {
       public:
       int month, day;
       string category, payee;
       float amount;
       expense()
          { }
       expense(int m, int d, string c, string p, float a) :
               month(m), day(d), category(c), payee(p), amount(a)
          { /*bos */ }
       // 'kume'de kullanmak icin gerekli
       friend bool operator < (const expense&, const expense&);
       friend bool operator == (const expense&, const expense&);
       //cikti icin gerekli
       friend ostream& operator << (ostream&, const expense&);
       };    //class expense'in sonu
 ////////////////////////////////////////////////////////////////
 class compareDates                  //fonksiyon nesnesi-masraflari karsilastirir
    {
    public:
       bool operator () (expense*, expense*) const;
    };
 ////////////////////////////////////////////////////////////////
 class compareCategories           //fonksiyon nesnesi - masraflari karsilastirir
    {
    public:
       bool operator () (expense*, expense*) const;
    };
 ////////////////////////class expenseRecord//////////////////////
 class expenseRecord
    {
    private:
    //masraflara isaret eden isaretci vektoru
    vector<expense*> vectPtrsExpenses;
    vector<expense*>::iterator iter;

    public:
    ~expenseRecord();
    void insertExp(expense*);
    void display();
    float displaySummary();       //annualReport tarafindan kullanilir
    };    //class expenseRecord'in sonu

//////////////////////class expenseInputScreen//////////////////
class expenseInputScreen
   {
   private:
   expenseRecord* ptrExpenseRecord;

   public:
```



```
   expenseInputScreen(expenseRecord*);
   void getExpense();
   };    //class expenseInputScreen'in sonu

//////////////////////class annualReport//////////////////////
class annualReport
   {
   private:
   rentRecord* ptrRR;
   expenseRecord* ptrER;
   float expenses, rents;

   public:
   annualReport(rentRecord*, expenseRecord*);
   void display();
   };    //class annualReport'in sonu

//////////////////////class userInterface//////////////////////
class userInterface
   {
   private:
   tenantList*              ptrTenantList;
   tenantInputScreen*       ptrTenantInputScreen;
   rentRecord*              ptrRentRecord;
   rentInputScreen*         ptrRentInputScreen;
   expenseRecord*           ptrExpenseRecord;
   expenseInputScreen*      ptrExpenseInputScreen;
   annualReport*            ptrAnnualReport;
   char ch;

   public:
   userInterface();
   ~userInterface();
   void interact();
   };    //class userInterface'in sonu
////////////////////////////landlord.h dosyasinin sonu//////////////////////
```

Sınıf Deklarasyonları

Sınıf deklarasyonları işin kolay kısmıdır. Sınıf deklarasyonlarının birçoğu doğrudan use case tanımlarındaki isimlerin incelenmesiyle keşfedilen ve sınıf şeması üzerinde görülen sınıflardan ortaya çıkar. İsimler, birkaç kelimeden oluşan İngilizce versiyonlarından tek kelimelik bilgisayar versiyonlarına çevrilir; böylece, örneğin Tenant List (Kiracı Listesi) artık **tenantList** halini alır.

Burada başlık dosyasına birkaç yeni sınıf daha eklendi. Ayrıca, çeşitli tipte STL konteynerlerindeki nesneleri gösteren işaretçileri saklıyor olduğumuzu da fark edeceğiz. "Standart Şablon Kütüphanesi" adlı Bölüm 15'te bahsettiğimiz gibi bu, bu konteynerler için karşılaştırma nesneleri tanımlamamız gerekiyor, demektir. Bu karşılaştırma nesneleri aslında **compareTenants**, **compareRows**, **compareDates** ve **compareCategories** adındaki sınıflardır.

Nitelik Deklarasyonları

Bahsettiğimiz gibi, her sınıf için niteliklerin (üye verilerin) bir çoğu sınıf olarak kullanılmamış isimlerden belirlenebilir. Örneğin, **name** ve **aptNumber**, **tenant** sınıf deklarasyonunun nitelikleri haline gelir.

Diğer nitelikler sınıf şemasındaki bağlantılardan (associations) çıkarılabilir. Bağlantılar, diğer sınıfları gösteren işaretçi veya referansları da belirtebilirler. Bunun nedeni şudur: Eğer bir sınıfı ne ile ilişkilendirdiğinizi bulamıyorsanız, onunla ilişkilendiremezsiniz. Dolayısıyla, `rentInputScreen` sınıfı, `ptrTenantList` ve `ptrRentRecord` niteliklerini içerir.

Birlikler (Aggregates)

Birlik bağlantıları sınıf şeması üzerinde üç yerde gösterilir. Genellikle birlikler, nesneleri (parçalar) tutan bir sınıfın (bütün) niteliği durumunda olan konteynerleri belirtir.

Ne use case tanımları ne de sınıf şeması bu birlikler için ne tür bir konteyner kullanılması gerektiğini belirtmiyorlar. Bir programcı olarak, her birlik için uygun bir konteyner seçmeniz gerekecek. Bu konteyner, basit bir dizi, bir STL konteyneri veya başka bir şey olabilir. `LANDLORD` programı için aşağıdaki tercihlerde bulunduk:

- `tenantList` sınıfı, `tenant` nesnelerine işaret eden işaretçilerden oluşan bir STL kümesi içerir.
- `rentRecord` sınıfı, `rentRow` nesnelerine işaret eden bir işaretçi kümesi içerir.
- `expenseRecord` sınıfı, `expense` nesnelerine işaret eden bir işaretçi vektörü içerir.

Hızlı erişim sağlamak amacıyla `tenantList` ve `rentRecord` için küme kullandık. `expenseRecord` için ise bir vektör kullandık, çünkü `Expense` nesnelerini hem tarih hem de kategori bazında sıralamamız gerekiyor ve vektörler (kümelerden farklı olarak) verimli bir biçimde sıralanabilir.

Nesnelerin kendilerini ne zaman saklamak isterseniz, her seferinde bir kopyalama işlemini gerçekleştirmeniz gerekir. Bu kopyalama işlemini önlemek için birliklerin tümünde, asıl nesneler yerine bunlara işaret eden işaretçileri saklamayı tercih ettik. Doğrudan nesneleri saklamak, nesnelerin küçük ve sayıca az olduğu durumlarda uygun olabilir. Bunun gibi küçük bir programda nesneleri saklamaktan kaynaklanan performans kaybı elbette büyük değildir; ancak, verimlilik açısından, daima işaretçileri saklamayı dikkate almalısınız.

.CPP Dosyaları

`.CPP` dosyaları, deklarasyonları ve `.H` dosyasında verilen yöntem gövdelerini içerirler. Bu yöntemlerin kodunu yazmak bu aşamada son derece kolaydır. Fonksiyonun ismini, ne yapması gerektiğini ve muhtemelen fonksiyona aktarılacak argümanları zaten biliyorsunuz.

Sınıfların yöntem tanımlarını, kısa bir `LORDAPP.CPP` dosyasında yer alan `main()`'den ayırdık. `main()`'de bir `userInterface` nesnesi tanımlanır ve bu nesnenin `interact()` yöntemi çağrılır. `LORDAPP.CPP` dosyası şu şekildedir:

```
// lordApp.cpp
// ayri program icin istemci (client) dosya
#include "landlord.h "

int main()
   {
   userInterface theUserInterface;

   theUserInterface.interact();
   return 0;
   }
//////////////////////////lordApp.cpp dosyasinin sonu//////////////////////
```


Nihayet, sınıfların yöntem tanımlarının tümünü içeren `LANDLORD.CPP` dosyası, şu şekildedir:

```
// landlord.cpp
// bir apartmanin muhasebesini modeller
#include "landlord.h"
//////////////////////Global fonksiyonlar/////////////////////////////
void getaLine(string& inStr)          //metin satirini al
   {
   char temp[21];
   cin.get(temp, 20, '\n');
   cin.ignore(20, '\n');
   inStr = temp;
   }
//-------------------------------------------------------------
char getaChar()                       //bir karakter al
   {
   char ch = cin.get();
   cin.ignore(80, '\n');
   return ch;
   }
//-------------------------------------------------------------

//////////////////////tenant sinifi icin yontemler///////////////
tenant::tenant(string n, int aNo) : name(n), aptNumber(aNo)
   { /*bos */ }
//-------------------------------------------------------------
tenant::~tenant()
   { /*bos */ }
//-------------------------------------------------------------
int tenant::getAptNumber()
   {return aptNumber;}
//-------------------------------------------------------------
bool operator < (const tenant& t1, const tenant& t2)
   { return t1.name < t2.name; }
//-------------------------------------------------------------
bool operator == (const tenant& t1, const tenant& t2)
   { return t1.name == t2.name; }
//-------------------------------------------------------------
ostream& operator << (ostream& s, const tenant& t)
   { s << t.aptNumber << '\t' << t.name << endl; return s; }
//-------------------------------------------------------------

////////////////tenantInputScreen sinifi icin yontemler/////////
void tenantInputScreen::getTenant()   //kiraci bilgisini al
   {
   cout << "Enter tenant's name (George Smith): ";
   getaLine(tName);
   cout << "Enter tenant's apartment number (101): ";
   cin >> aptNo;
   cin.ignore(80, '\n');              //kiraci olustur
   tenant* ptrTenant = new tenant(tName, aptNo);
   ptrTenantList->insertTenant(ptrTenant); //kiraci listesine gonder
   }
//////////////////////////////////////////////////////////////////
bool compareTenants::operator () (tenant* ptrT1,
                                  tenant* ptrT2) const
   { return *ptrT1 < *ptrT2; }
//-------------------------------------------------------------
```

```
//////////////////tenantList sinifi icin yontemler//////////////
tenantList::~tenantList()                  //yok edici fonk.
   {
   while( !setPtrsTens.empty() )           //tum kiracilari sil,
      {                                    //isaretcileri kumeden cikart
      iter = setPtrsTens.begin();
      delete *iter;
      setPtrsTens.erase(iter);
      }
   } //~tenantList()'in sonu
//-------------------------------------------------------------
void tenantList::insertTenant(tenant* ptrT)
   {
   setPtrsTens.insert(ptrT);               //ekle
   }
//-------------------------------------------------------------
int tenantList::getAptNo(string tName)     //isim listede mi?
   {
   int aptNo;
   tenant dummy(tName, 0);
   iter = setPtrsTens.begin();
   while( iter != setPtrsTens.end() )
      {
      aptNo = (*iter)->getAptNumber();     //kiraciyi ara
      if(dummy == **iter++)                //listede mi?
         return aptNo;                     //evet
      }
   return -1;                              //hayir
   }
//-------------------------------------------------------------
void tenantList::display()                 //kiraci listesini goster
   {
   cout << "\nApt#\tTenant name\n------------------\n";
   if( setPtrsTens.empty() )
      cout << "***No tenants***\n";
   else
      {
      iter = setPtrsTens.begin();
      while( iter != setPtrsTens.end() )
         cout << **iter++;
      }
   }     //display()'in sonu
//-------------------------------------------------------------

//////////////////////rentRow sinifi icin yontemler/////////////
rentRow::rentRow(int an) : aptNo(an)   //1-argumanli kurucu fonk.
   { fill( &rent[0], &rent[12], 0); }
//-------------------------------------------------------------
void rentRow::setRent(int m, float am)
   { rent[m] = am; }
//-------------------------------------------------------------
float rentRow::getSumOfRow()     //bir satirdaki kiralar toplami
   { return accumulate( &rent[0], &rent[12], 0); }
//-------------------------------------------------------------
bool operator < (const rentRow& t1, const rentRow& t2)
   { return t1.aptNo < t2.aptNo; }
//-------------------------------------------------------------
bool operator == (const rentRow& t1, const rentRow& t2)
   { return t1.aptNo == t2.aptNo; }
```



```
//-------------------------------------------------------------
ostream& operator << (ostream& s, const rentRow& an)
   {
   s << an.aptNo << '\t';
   for(int j=0; j<12; j++)              //daire numarasini yazdir
      {                                 //12 kirayi yazdir
      if(an.rent[j] == 0)
         s << " 0 ";
      else
         s << an.rent[j] << " ";
      }
   s << endl;
   return s;
   }
////////////////////////////////////////////////////////////////
bool compareRows::operator () (rentRow* ptrR1,
                               rentRow* ptrR2) const
   { return *ptrR1 < *ptrR2; }

//////////////////rentRecord sinifi icin yontemler/////////////
rentRecord::~rentRecord()                  //yok edici fonk.
   {
   while( !setPtrsRR.empty() )             //kira satirlarini yok et,
      {                                    //isaretcileri kumeden cikart
      iter = setPtrsRR.begin();
      delete *iter;
      setPtrsRR.erase(iter);
      }
   }
//-------------------------------------------------------------
void rentRecord::insertRent(int aptNo, int month, float amount)
   {
   rentRow searchRow(aptNo);     //ayni daire nosuna sahip gecici bir satir
   iter = setPtrsRR.begin();     //setPtrsRR'yi ara
   while( iter != setPtrsRR.end() )
      {
      if(searchRow==**iter)      //rentRow (kira satiri) bulundu mu?
         {                       //evet,
         (*iter)->setRent(month, amount);  //kirayi satira yerlestir (isle)
         return;
         }
      else
         iter++;
      }                                    //bulunamadi
   rentRow* ptrRow = new rentRow(aptNo);   //yeni bir satir ac
   ptrRow->setRent(month, amount);         //kirayi satira isle
   setPtrsRR.insert(ptrRow);               //satiri vektore koy
   }     //insertRent()'in sonu
//-------------------------------------------------------------
void rentRecord::display()
   {
   cout << "\nAptNo\tJan   Feb   Mar   Apr   May   Jun
        <<          "Jul   Aug   Sep   Oct   Nov   Dec\n"
        <<"--------------------------------"
        <<"--------------------------------\n";
   if( setPtrsRR.empty() )
      cout << "***No rents***\n";
   else
      {
```

```
      iter = setPtrsRR.begin();
      while( iter != setPtrsRR.end() )
         cout << **iter++;
      }
   }
//-------------------------------------------------------------------
float rentRecord::getSumOfRents()   //kiralarin tumunun toplamini dondur
   {
   float sumRents = 0.0;
   iter = setPtrsRR.begin();
   while( iter != setPtrsRR.end() )
      {
      sumRents += (*iter)->getSumOfRow();
      iter++;
      }
   return sumRents;
   }
//-------------------------------------------------------------------

//////////////////rentInputScreen sinifi icin yontemler///////////////
void rentInputScreen::getRent()
   {
   cout << "Enter tenant's name: ";
   getaLine(renterName);
   aptNo = ptrTenantList->getAptNo(renterName);
   if(aptNo > 0)                  //eger isim bulunduysa,
      {                           //kira miktarini al
      cout << "Enter amount paid (345.67): ";
      cin >> rentPaid;
      cin.ignore(80, '\n');
      cout << "Enter month rent is for (1-12): ";
      cin >> month;
      cin.ignore(80, '\n');
      month--;                    //(aylar programda 0-11 olarak tutulur)
      ptrRentRecord->insertRent(aptNo, month, rentPaid);
      }
   else                           //don
      cout << "No tenant with that name.\n";
   }     //end getRent()
//-------------------------------------------------------------------

//////////////////expense sinifi icin yontemler/////////////////
bool operator < (const expense& e1, const expense& e2)
   {                              //tarihleri karsilastirir
   if(e1.month == e2.month)       //aylar ayniysa,
      return e1.day < e2.day;     //gunleri karsilastir
   else                           //aksi halde,
      return e1.month < e2.month; //aylari karsilastir
   }
//-------------------------------------------------------------------
bool operator == (const expense& e1, const expense& e2)
   { return e1.month == e2.month && e1.day == e2.day; }
//-------------------------------------------------------------------
ostream& operator << (ostream& s, const expense& exp)
   {
   s << exp.month << '/' << exp.day << '\t' << exp.payee << '\t' ;
   s << exp.amount << '\t' << exp.category << endl;
   return s;
   }
```



```
//-------------------------------------------------------------------
////////////////////////////////////////////////////////////////
bool compareDates::operator () (expense* ptrE1,
                                expense* ptrE2) const
   { return *ptrE1 < *ptrE2; }
//-------------------------------------------------------------------
////////////////////////////////////////////////////////////////
bool compareCategories::operator () (expense* ptrE1,
                                     expense* ptrE2) const
   { return ptrE1->category < ptrE2->category; }
//-------------------------------------------------------------------

//////////////////expenseRecord sinifi icin yontemler///////////////
expenseRecord::~expenseRecord()        //yok edici fonk.
   {
   while( !vectPtrsExpenses.empty() )  //masraf nesnelerini yok et,
      {                                //isaretcileri vektorden cikart
      iter = vectPtrsExpenses.begin();
      delete *iter;
      vectPtrsExpenses.erase(iter);
      }
   }
//-------------------------------------------------------------------
void expenseRecord::insertExp(expense* ptrExp)
   { vectPtrsExpenses.push_back(ptrExp); }
//-------------------------------------------------------------------
void expenseRecord::display()
   {
   cout << "\nDate\tPayee\t\tAmount\tCategory\n"
        <<"-----------------------------------\n";
   if( vectPtrsExpenses.size() == 0 )
      cout << "***No expenses***\n";
   else
      {
      sort( vectPtrsExpenses.begin(),      //tarihe gore sirala
            vectPtrsExpenses.end(), compareDates() );
      iter = vectPtrsExpenses.begin();
      while( iter != vectPtrsExpenses.end() )
         cout << **iter++;
      }
   }
//-------------------------------------------------------------------
float expenseRecord::displaySummary()      //annualReport tarafindan kullanilir
   {
   float totalExpenses = 0;                //toplam, tum kategoriler

   if( vectPtrsExpenses.size() == 0 )
      {
      cout << "\tAll categories\t0\n";
      return 0;
      }
   //kategoriye gore sirala
   sort( vectPtrsExpenses.begin(),
         vectPtrsExpenses.end(), compareCategories() );

   //her kategori icin, unsurlari topla
```

```
      iter = vectPtrsExpenses.begin();
      string tempCat = (*iter)->category;
      float sumCat = 0.0;
      while( iter != vectPtrsExpenses.end() )
         {
         if(tempCat == (*iter)->category)
            sumCat += (*iter)->amount;          //ayni kategori
         else
            {                                   //farkli kategori
            cout << '\t' << tempCat << '\t' << sumCat << endl;
            totalExpenses += sumCat;            //onceki kategoriyi ekle
            tempCat = (*iter)->category;
            sumCat = (*iter)->amount;           //son meblagi ekle
            }
         iter++;
         }     //while'in sonu
      totalExpenses += sumCat;                  //son kategoriyi ekle
      cout << '\t' << tempCat << '\t' << sumCat << endl;
      return totalExpenses;
      }     //displaySummary()'nin sonu
//-------------------------------------------------------

//////////////expenseInputScreen sinifi icin yontemler/////////
expenseInputScreen::expenseInputScreen(expenseRecord* per) :
                                    ptrExpenseRecord(per)
   { /*bos*/ }
//-------------------------------------------------------
void expenseInputScreen::getExpense()
   {
   int month, day;
   string category, payee;
   float amount;

   cout << "Enter month (1-12): ";
   cin >> month;
   cin.ignore(80, '\n');
   cout << "Enter day (1-31): ";
   cin >> day;
   cin.ignore(80, '\n');
   cout << "Enter expense category (Repairing,Utilities): ";
   getaLine(category);
   cout << "Enter payee "
        << "(Bob 's Hardware,Big Electric Co): ";
   getaLine(payee);
   cout << "Enter amount (39.95): ";
   cin >> amount;
   cin.ignore(80, '\n');
   expense* ptrExpense = new
               expense(month, day, category, payee, amount);
   ptrExpenseRecord->insertExp(ptrExpense);
   }
//-------------------------------------------------------

//////////////////annualReport sinifi icin yontemler/////////
annualReport::annualReport(rentRecord* pRR,
                           expenseRecord* pER):
                                    ptrRR(pRR), ptrER(pER)
   { /*bos*/ }
//-------------------------------------------------------
```



```
void annualReport::display()
   {
   cout << "Annual Summary\n----------\n";
   cout << "Income\n";
   cout << "\tRent\t\t";
   rents = ptrRR->getSumOfRents();
   cout << rents << endl;

   cout << "Expenses\n";
   expenses = ptrER->displaySummary();
   cout << "\nBalance\t\t\t" << rents - expenses << endl;
   }
//-------------------------------------------------------

//////////////////userInterface sinifi icin yontemler//////////////
userInterface::userInterface()
   {
   //bu raporlar programin omru boyunca mevcuttur
   ptrTenantList = new tenantList;
   ptrRentRecord = new rentRecord;
   ptrExpenseRecord = new expenseRecord;
   }
//-------------------------------------------------------
userInterface::~userInterface()
   {
   delete ptrTenantList;
   delete ptrRentRecord;
   delete ptrExpenseRecord;
   }
//-------------------------------------------------------
void userInterface::interact()
   {
   while(true)
      {
      cout << "Enter 'i' to input data, \n"
           << "      'd' to display a report, \n"
           << "      'q' to quit program: ";
      ch = getaChar();
      if(ch=='i')                       //verileri gir
         {
         cout << "Enter 't' to add tenant, \n"
              << "      'r' to record rent payment, \n"
              << "      'e' to record expense: ";
         ch = getaChar();
         switch(ch)
            {
            //giris ekranlari yalnizca kullanimlari suresince mevcuttur
            case 't': ptrTenantInputScreen =
               new tenantInputScreen(ptrTenantList);
               ptrTenantInputScreen->getTenant();
               delete ptrTenantInputScreen;
               break;
            case 'r': ptrRentInputScreen =
               new rentInputScreen(ptrTenantList, ptrRentRecord);
               ptrRentInputScreen->getRent();
               delete ptrRentInputScreen;
               break;
            case 'e': ptrExpenseInputScreen =
               new expenseInputScreen(ptrExpenseRecord);
```

```
                ptrExpenseInputScreen->getExpense();
                delete ptrExpenseInputScreen;
                break;
             default: cout << "Unknown input option\n";
                break;
             }     //switch'in sonu
          }     //if'in sonu
       else if(ch=='d')                        //verileri goster
          {
          cout << "Enter 't' to display tenants, \n"
               << "      'r' to display rents \n"
               << "      'e' to display expenses,\n"
               << "      'a' to display annual report: ";
          ch = getaChar();
          switch(ch)
             {
             case 't': ptrTenantList->display();
                break;
             case 'r': ptrRentRecord->display();
                break;
             case 'e': ptrExpenseRecord->display();
                break;
             case 'a':
                ptrAnnualReport = new annualReport(ptrRentRecord,
                                                   ptrExpenseRecord);
                ptrAnnualReport->display();
                delete ptrAnnualReport;
                break;
             default: cout << "Unknown display option\n";
                break;
             }     //switch'in sonu
          }     //elseif'in sonu
       else if(ch=='q')
          return;                          //cik
       else
          cout << "Unknown option. Enter only 'i', 'd' or 'q'\n";
       }     //while'in sonu
    }     //interact()'in sonu
////////////////////landlord.cpp dosyasinin sonu//////////////////////
```

Biraz Daha Sadeleştirme

LANDLORD için gösterdiğimiz kod oldukça uzun olmakla birlikte yine de birçok sadeleştirme içerir. Program, modern bir Grafik Kullanıcı Arayüzünün menüleri ve pencereleri yerine karakter modunda bir kullanıcı arayüzü kullanır. Kullanıcı verileri için çok az hata kontrolü yapılır. Sadece bir yıllık veriler kontrol altına tutulur.

Programla Etkileşim

LANDLORD programını tasarlamak ve yazmak gibi bir zorluğa giriştikten sonra programla gerçekleştirilecek etkileşim örneklerini görmek ilginizi çekebilir. Beverly, yeni bir kiracı ismi ve daire numarası girmek için programı şu şekilde kullanır. Beverly, "insert tenant (kiracı ekle)" işlemi için önce 'i' ardından 't' harflerini girer, sonra programın yönlendirmelerine göre ilgili verileri girer. (Bu yönlendirmelerde genellikle, girdi için uygun biçim, parantez içinde gösterilir.)


```
Enter 'i' to input data,
      'd' to display a report,
      'q' to quit program: i
Enter 't' to add a tenant,
      'r' to record a rent payment,
      'e' to record an expense: t
Enter tenant's name (George Smith): Harry Ellis
Enter tenant's apartment number: 101
```

Beverly, kiracıların tümünü girdikten sonra kiracı listesini ekranda gösterebilir (çok yer kaplamaması için burada on iki kiracının yalnızca beş tanesini gösterir):

```
Enter 'i' to input data,
      'd' to display a report,
      'q' to quit program: d
Enter 't' to display tenants,
      'r' to display rents,
      'e' to display expenses,
      'a' to display annual report: t
Apt#  Tenant name
--------------------------
101   Harry Ellis
102   Wanda Brown
103   Peter Quan
104   Bill Vasquez
201   Jane Garth
```

Kiracı tarafından ödenen bir kirayı girmek için Beverly önce 'i' harfini, sonra 'r' harfini tuşlar. (Şu andan itibaren program tarafından gösterilen tercih listelerine burada yer vermeyeceğiz.) Etkileşim şu şekilde görünür:

```
Enter tenant 's name: Wanda Brown
Enter amount paid (345.67): 595
Enter month rent is for (1-12): 5
```

Böylece, Wanda Brown, Mayıs ayı kirası için 595 dolar tutarında bir çek göndermiş olur. (Kiracının ismi aynen kiracı listesinde göründüğü gibi yazılmalıdır. Daha akıllı bir program daha esnek olabilir.)

Kira Kayıtlarının tümünü görmek için Beverly 'd', ardından 'r' tuşluyor. Mayıs kiraları alındıktan sonra eldeki sonuçlar aşağıda görüldüğü gibidir (çok yer kaplamaması için Beverly'nin 12 dairesinin sadece beşinin kiraları gösterilmiştir):

```
AptNo Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
-------------------------------------------------------
101   695 695 695 695 695  0   0   0   0   0   0   0
102   595 595 595 595 595  0   0   0   0   0   0   0
103   810 810 825 825 825  0   0   0   0   0   0   0
104   645 645 645 645 645  0   0   0   0   0   0   0
201   720 720 720 720 720  0   0   0   0   0   0   0
```

Dikkat ederseniz, Beverly, Peter Quan'ın kirasını Mart ayında artırmıştır.

Masrafları girmek için ise Beverly önce 'i' sonra 'e' tuşlar. İşte bir etkileşim örneği:

```
Enter month: 1
Enter day: 15
Enter expense category (Repairing,Utilities): Utilities
Enter payee (Bob 's Hardware,Big Electric Co): P.G.&E.
Enter amount: 427.23
```

Masraf Kayıtlarını göstermek için 'd' ve 'e' tuşlar. Burada raporun sadece baş tarafını gösterilmiştir:

```
Date  Payee              Amount   Category
-----------------------------------------
1/3   First MegaBank    5187.30  Mortgage
1/8   City Water         963.10  Utilities
1/9   Steady State      4840.00  Insurance
1/15  P.G.&E.            727.23  Utilities
1/22  Sam 's Hardware     54.81  Supplies
1/25  Ernie Glotz        150.00  Repairs
2/3   First MegaBank    5187.30  Mortgage
```

Yıllık Raporu göstermek için ise Beverly 'd' ve 'a' harflerini girer. Yılın ilk beş ayını kapsayan kısmi bir rapor şöyle görünür:

```
Annual Summary
--------------
Income
            Rents       42610.12
Expenses
Advertising                95.10
Insurance                4840.00
Mortgage                25936.57
Repairs                  1554.90
Supplies                  887.22
Utilities                7636.15
Balance                  1660.18
```

Masraf kategorileri alfabetik olarak sıralanmıştır. Gerçek bir durumda birçok bütçe kategorisi daha mevcut olacaktır. Yasal ödemeler, vergiler, seyahat harcamaları, peyzaj, temizlik ve bakım maliyetleri vs bu tür kategoriler arasında yer alabilir.

Nihai Düşünceler

Büyüklüğü ne olursa olsun gerçek bir projede geliştirme süreci, muhtemelen bu bölümde tasvir ettiğimiz kadar düzgün gitmeyecektir. Burada gösterdiğimiz aşamaların her birinin birkaç kez tekrarlanması gerekecektir. Programcılar, kuruluş aşamasının ortalarındayken kullanıcının istekleri hakkında kararsızlığa düşebilirler ve ayrıntılandırma aşamasına geri dönmek zorunda kalabilirler. Kullanıcılar, sürecin sonlarına gelinirken fikirlerini değiştirebilirler ve ilk aşamalara dönmek zorunda kalabilirler.

Özet

Basit bir yazılım geliştirme için deneme-yanılma yöntemi yeterli olabilir. Karmaşık projeler için ise daha iyi organize olmuş bir yaklaşım genellikle gereklidir. Bu bölümde, mümkün olan yöntembilimlerden birini sunduk. Birleştirilmiş Süreç (Unified Process); başlangıç, ayrıntılandırma,

kuruluş ve geçiş aşamalarından oluşur. Ayrıntılandırma, programın analizine; kuruluş ise kodun tasarlanması ve yazılmasına karşılık gelir.

Birleştirilmiş Süreç, programın kullanıcılarını (aktörleri) ve programdan yürütmesini istedikleri görevleri (use case'ler) kontrol altında tutmak amacıyla use case yaklaşımını kullanır. UML use case şeması, aktörleri ve use case'leri gösterir. Use case tanımlarındaki herhangi bir isim, bir sınıf veya sınıf niteliği olmaya adaydır. Eylemler, sınıfların üye fonksiyonlarına dönüşürler (üye fonksiyonlar ayrıca işlem veya yöntem olarak da adlandırılır).

Use case şemalarına ek olarak diğer UML şemaları, programın kullanıcıları ile geliştiricileri arasındaki karşılıklı anlayışı kolaylaştırmaya yardımcı olurlar. Sınıflar arası ilişkiler sınıf şemaları ile; bir faaliyetten diğerine kontrol akışı faaliyet şemaları ile gösterilebilir. Sekans şemaları ise bir use case'in kullanımı süresince nesneler arasındaki iletişimi tarif eder.

Sorular

Bu soruların cevaplarını Ek G'de bulabilirsiniz.

1. Doğru/Yanlış: Use case yaklaşımı her şeyden önce bir sınıfın kullandığı yöntemlerle ilgilidir.
2. Use case'ler (diğer görevlerin yanı sıra) hangi amaç için kullanılır?
   a. Program kodu içinde karşılaşılan problemleri özetlemek için.
   b. Bir sınıfın hangi kurucu fonksiyonlara sahip olabileceğini belirlemek için.
   c. Uygun sınıf niteliklerini seçmeye yardımcı olmak için.
   d. Bir programda hangi sınıfların gerekli olabileceğini saptamak için.
3. Bir use case esasında bir _____dir.
4. Doğru/Yanlış: Bir use case şeması oluşturulduktan sonra yeni use case'ler kodlama başladıktan sonra eklenebilir.
5. Bir use case tanımı kimi zaman iki _____ yazılır.
6. Aktör,
   a. geliştirilmekte olan sistemle etkileşmekte olan farklı bir sistemdir.
   b. Belirli bir programlama problemini çözmek için geliştiriciye yardımcı olan bir yazılım unsurudur.
   c. Geliştirilmekte olan sistemle etkileşmekte olan kişidir.
   d. Sistemi tasarlayan kişidir.
7. Sınıflar (diğerlerinin yanı sıra) bir g_____, _____ veya _____ olarak ilişkilendirilebilir.
8. Şelale yöntemi (waterfall process)
   a. ayrık aşamalardan oluşur.
   b. Aslında hiçbir zaman kullanılmaz.
   c. Su kesintilerinden dolayı korumasız kalmıştır.
   d. Sadece tek yönde ilerleyebilir.
9. Doğru/Yanlış: Eğer UML kullanılıyorsa, Birleştirilmiş Süreç (Unified Process) de kullanılmalıdır.
10. Bir programdaki sınıflar aşağıdakilerden hangisine karşılık gelebilir?
    a. Use case tanımlarındaki isimlere.
    b. Use case'lere.
    c. Bir UML şemasındaki bağlantılara (associations).

d. Ünlü programcıların isimlerine.

11. Doğru/Yanlış: Use case tanımlarındaki belirsiz, genel unsurlar (mesela, "sistem" gibi), bir programdaki sınıflar için iyi birer adaydır.
12. Doğru/Yanlış: Tek bir niteliği olan, yöntemleri olmayan unsurlar sınıflar için iyi birer adaydır.
13. Birleştirilmiş Süreçte (Unified Process), aşağıdakilerden hangisi zaman zaman ortaya çıkabilir?
    a. Kullanıcılar use case'lerin tümünü belirtmeden önce use case şeması çizilmiş olacaktır.
    b. Bazı use case tanımları yazılmadan önce bir sınıf şeması çizilmiş olacaktır.
    c. Sınıf şeması tamamlanmadan önce kodun bir kısmı yazılmış olacaktır.
    d. Yöntemlerin kodlanması halen sürerken, sınıf deklarasyonlarını içeren başlık dosyası değiştirilmiş olacaktır.
14. Aktörler, _____ veya _____ etkileşen _____ dir.
15. `LANDLORD` programında, STL konteyner sınıfları
    a. kullanılamaz, çünkü bunlar use case şemalarında temsil edilemezler.
    b. masrafları saklamak için iyi bir yer oluştururlar.
    c. kullanılamaz, çünkü C++ nesne yönelimli bir dildir.
    d. yöntem gövdelerini saklamak için iyi bir yer oluştururlar.
16. Sınıfların yöntem tanımları
    a. bir başlık dosyası içine yerleştirilmelidir.
    b. bir başlık dosyası içine yerleştirilmemelidir.
    c. muhtemelen müşterilere dağıtılmamalıdır.
    d. genellikle müşterilere dağıtılmalıdır.
17. Doğru/Yanlış: Nitelik, başlıca sınıf ilişkilerinden biridir.
18. `A` sınıfı ve `B` sınıfı arasında bir bağlantı (association) olduğunu varsayalım. Ayrıca, `objA`, `A` sınıfının bir nesnesi, `objB` de `B` sınıfının bir nesnesi olsun. Bu durumda aşağıdakilerden hangisi söz konusu olur?
    a. `objA`, `objB`'ye bir mesaj gönderebilir.
    b. `B` sınıfı, `A` sınıfının bir alt kümesi olmalıdır; ya da tam tersi.
    c. `objB`, `A` sınıfının bir niteliği olmalıdır; ya da tam tersi.
    d. `objB`, `objA`'ya bir görevi yürütmesinde yardımcı olabilir.
19. `LANDLORD` programı aşağıdakilerin hangisinden yararlanır?
    a. genelleştirme.
    b. Bağlantı (association).
    c. Başkaldırı.
    d. Birlik (aggregation).
20. Doğru/Yanlış: Bir sınıf şemasında, bir bağlantı (association) nesneler arasındaki bir ilişki olarak gösterilir.
21. Bir sekans (sequential) şemasında
    a. zaman soldan sağa doğru ilerler.
    b. bağlantılar (associations) sağdan sola doğru gider.
    c. yatay oklar mesajları simgeler.
    d. düşey kesikli çizgiler yaşam süresini simgeler.

22. Bir sekans şeması, bir _____ diğerine gönderilen mesajları gösterir.
23. Doğru/Yanlış: Bir sekans şeması genellikle tek bir use case'i tanımlar.
24. Bir sekans şemasında yeni bir sınıf örneği oluşturulduğu zaman
    a. bir dikdörtgen uygun bir düşey koordinatta, ismiyle birlikte çizilir.
    b. büyük bir X, zaman içinde bir noktaya işaret eder.
    c. sınıf örneğinin faaliyet kutusu başlar.
    d. sınıf örneğinin yaşam çizgisi başlar.

Projeler

Bu bölümde yer alan projelere benzer projeleri alıştırmalara dahil etmek için bu kitapta yeterince yerimiz yok. Ancak, kendi kendinize uygulamak isteyebileceğiniz birkaç proje önerimizi burada listeliyoruz.

1. "İşaretçiler" adlı Bölüm 10'daki `HORSE` programının açıklamalarını yeniden okuyun; fakat, sakın koda *bakmayın*. Bu program için bir use case şeması, bir de sınıf şeması tasarlayın. Sonuçları, bir `.H` dosyası oluşturmak için kullanın ve sonuçlarınızı programınkiyle karşılaştırın. Birkaç tane doğru sonuç olabilir.
2. "Birden Fazla Dosya Kullanan Programlar" adlı Bölüm 13'te bahsindeki `ELEV` programının açıklamalarını yeniden okuyun; fakat, sakın koda *bakmayın*. Bu program için bir use case şeması, bir de sınıf şeması tasarlayın. Sonuçları, uygun `.H` dosyaları oluşturmak için kullanın. Sonuçlarınızı programınkiyle karşılaştırın.
3. Bildiğiniz bir ticari durum için bir use case şeması, bir de sınıf şeması tasarlayın. Bu bağlamda, at ticaretini, yazılım danışmanlığını veya ender bulunan mizah kitaplarının satışını düşünebilirsiniz.
4. Şimdiye kadar hep yazmak istediğiniz ama yazmak için bir türlü zaman bulamadığınız bir program için bir use case şeması, bir de sınıf şeması oluşturun. Eğer aklınıza bir şey gelmiyorsa, basit bir kelime işlemci programını, bir oyunu veya atalarınızla ilgili bilgileri girmenize imkan veren ve aile ağacını ekranda gösteren bir soy ağacı programını deneyin.

ASCII TABLOSU

TABLO A.1: IBM Karakter Kodları

Onluk Tabanda	Onaltılık Tabanda	Tuş	C'deki Kullanımı
0	00	Ctrl 2	
1	01	Ctrl A	
2	02	Ctrl B	
3	03	Ctrl C	
4	04	Ctrl D	
5	05	Ctrl E	
6	06	Ctrl F	
7	07	Ctrl G	Zil (bip)
8	08	Backspace	Backspace
9	09	Tab	Sekme
10	0A	Ctrl J	Linefeed (new line)
11	0B	Ctrl K	Düşey sekme
12	0C	Ctrl L	Form Feed
13	0D	Enter	Carriage Return
14	0E	Ctrl N	
15	0F	Ctrl O	
16	10	Ctrl P	
17	11	Ctrl Q	
18	12	Ctrl R	
19	13	Ctrl S	
20	14	Ctrl T	
21	15	Ctrl U	
22	16	Ctrl V	
23	17	Ctrl W	
24	18	Ctrl X	
25	19	Ctrl Y	
26	1A	Ctrl Z	
27	1B	Esc	
28	1C	Ctrl \	
29	1D	Ctrl]	
30	1E	Ctrl 6	
31	1F	Ctrl -	
32	20	Boşluk	
33	21	!	
34	22	"	
35	23	#	
36	24	$	

TABLO A.1: IBM Karakter Kodları

Onluk Tabanda	Onaltılık Tabanda	Tuş	C'deki Kullanımı
37	25	%	
38	26	&	
39	27	'	
40	28	(	
41	29	)	
42	2A	*	
43	2B	+	
44	2C	,	
45	2D	-	
46	2E	.	
47	2F	/	
48	30	0	
49	31	1	
50	32	2	
51	33	3	
52	34	4	
53	35	5	
54	36	6	
55	37	7	
56	38	8	
57	39	9	
58	3A	:	
59	3B	;	
60	3C	<	
61	3D	=	
62	3E	>	
63	3F	?	
64	40	@	
65	41	A	
66	42	B	
67	43	C	
68	44	D	
69	45	E	
70	46	F	
71	47	G	
72	48	H	
73	49	I	
74	4A	J	

TABLO A.1: IBM Karakter Kodları

Onluk Tabanda	Onaltılık Tabanda	Tuş	C'deki Kullanımı
75	4B	K	
76	4C	L	
77	4D	M	
78	4E	N	
79	4F	O	
80	50	P	
81	51	Q	
82	52	R	
83	53	S	
84	54	T	
85	55	U	
86	56	V	
87	57	W	
88	58	X	
89	59	Y	
90	5A	Z	
91	5B	[	
92	5C	\	
93	5D	]	
94	5E	^	
95	5F	_	
96	60	`	
97	61	a	
98	62	b	
99	63	c	
100	64	d	
101	65	e	
102	66	f	
103	67	g	
104	68	h	
105	69	i	
106	6A	j	
107	6B	k	
108	6C	l	
109	6D	m	
110	6E	n	
111	6F	o	
112	70	p	

TABLO A.1: IBM Karakter Kodları

Onluk Tabanda	Onaltılık Tabanda	Tuş	C'deki Kullanımı
113	71	q	
114	72	r	
115	73	s	
116	74	t	
117	75	u	
118	76	v	
119	77	w	
120	78	x	
121	79	y	
122	7A	z	
123	7B	{	
124	7C	|	
125	7D	}	
126	7E	~	
127	7F	Ctrl←	
128	80	Alt 128	
129	81	Alt 129	
130	82	Alt 130	
131	83	Alt 131	
132	84	Alt 132	
133	85	Alt 133	
134	86	Alt 134	
135	87	Alt 135	
136	88	Alt 136	
137	89	Alt 137	
138	8A	Alt 138	
139	8B	Alt 139	
140	8C	Alt 140	
141	8D	Alt 141	
142	8E	Alt 142	
143	8F	Alt 143	
144	90	Alt 144	
145	91	Alt 145	
146	92	Alt 146	
147	93	Alt 147	
148	94	Alt 148	
149	95	Alt 149	
150	96	Alt 150	

TABLO A.1: IBM Karakter Kodları

Onluk Tabanda	Onaltılık Tabanda	Tuş	C'deki Kullanımı
151	97	Alt 151	
152	98	Alt 152	
153	99	Alt 153	
154	9A	Alt 154	
155	9B	Alt 155	
156	9C	Alt 156	
157	9D	Alt 157	
158	9E	Alt 158	
159	9F	Alt 159	
160	A0	Alt 160	
161	A1	Alt 161	
162	A2	Alt 162	
163	A3	Alt 163	
164	A4	Alt 164	
165	A5	Alt 165	
166	A6	Alt 166	
167	A7	Alt 167	
168	A8	Alt 168	
169	A9	Alt 169	
170	AA	Alt 170	
171	AB	Alt 171	
172	AC	Alt 172	
173	AD	Alt 173	
174	AE	Alt 174	
175	AF	Alt 175	
176	B0	Alt 176	
177	B1	Alt 177	
178	B2	Alt 178	
179	B3	Alt 179	
180	B4	Alt 180	
181	B5	Alt 181	
182	B6	Alt 182	
183	B7	Alt 183	
184	B8	Alt 184	
185	B9	Alt 185	
186	BA	Alt 186	
187	BB	Alt 187	
188	BC	Alt 188	

TABLO A.1: IBM Karakter Kodları

Onluk Tabanda	Onaltılık Tabanda	Tuş	C'deki Kullanımı
189	BD	Alt 189	
190	BE	Alt 190	
191	BF	Alt 191	
192	C0	Alt 192	
193	C1	Alt 193	
194	C2	Alt 194	
195	C3	Alt 195	
196	C4	Alt 196	
197	C5	Alt 197	
198	C6	Alt 198	
199	C7	Alt 199	
200	C8	Alt 200	
201	C9	Alt 201	
202	CA	Alt 202	
203	CB	Alt 203	
204	CC	Alt 204	
205	CD	Alt 205	
206	CE	Alt 206	
207	CF	Alt 207	
208	D0	Alt 208	
209	D1	Alt 209	
210	D2	Alt 210	
211	D3	Alt 211	
212	D4	Alt 212	
213	D5	Alt 213	
214	D6	Alt 214	
215	D7	Alt 215	
216	D8	Alt 216	
217	D9	Alt 217	
218	DA	Alt 218	
219	DB	Alt 219	
220	DC	Alt 220	
221	DD	Alt 221	
222	DE	Alt 222	
223	DF	Alt 223	
224	E0	Alt 224	
225	E1	Alt 225	
226	E2	Alt 226	

TABLO A.1: IBM Karakter Kodları

Onluk Tabanda	Onaltılık Tabanda	Tuş	C'deki Kullanımı
227	E3	Alt 227	
228	E4	Alt 228	
229	E5	Alt 229	
230	E6	Alt 230	
231	E7	Alt 231	
232	E8	Alt 232	
233	E9	Alt 233	
234	EA	Alt 234	
235	EB	Alt 235	
236	EC	Alt 236	
237	ED	Alt 237	
238	EE	Alt 238	
239	EF	Alt 239	
240	F0	Alt 240	
241	F1	Alt 241	
242	F2	Alt 242	
243	F3	Alt 243	
244	F4	Alt 244	
245	F5	Alt 245	
246	F6	Alt 246	
247	F7	Alt 247	
248	F8	Alt 248	
249	F9	Alt 249	
250	FA	Alt 250	
251	FB	Alt 251	
252	FC	Alt 252	
253	FD	Alt 253	
254	FE	Alt 254	
255	FF	Alt 255	

"Ctrl" kelimesini içeren tuş çiftleri CTRL tuşu basılarak tuşlanabilir; CTRL tuşunu basılı tutarken belirtilen tuşun da aynı anda basılması gerekir. Bu tuş çiftleri, PC Personal Computer serisi klavyeler için tanımlanmış tuş çiftlerini temel alır. Diğer klavyelerde bu tuş çiftleri farklı biçimde tanımlanmış olabilir.

IBM Genişletilmiş ASCII karakterleri, ALT tuşuna basılarak ekranda gösterilebilir. ALT tuşu basılı tutulurken, yazılacak karakterin ondalık kodu sayısal tuş takımı üzerinde tuşlanır.

EK B

B

C++ ÖNCELİK SIRASI TABLOSU VE ANAHTAR KELİMELER

Öncelik Sırası Tablosu

Bu öncelik sırası tablosu, Bölüm 3'te gösterilen tablonun komple versiyonudur. Bu tablo bit tabanlı (bitwise) operatörleri de kapsar. Bit tabanlı operatörler, bu kitapta bu şekilde bahsedilmemiş olsalar da, << ve >> operatörleri akış giriş ve çıkışlarında kullanıldığından, aşırı yüklenebilirler.

Operatörün Tipi	Operatörler
Kapsam	::
Çeşitli	[], (), . (nokta), ->, sonek ++ ve --
Tekli	önek ++ ve --, &, *, +, -, !
Aritmetik	Çarpma özelliğine sahip *, /, %
	Toplama özelliğine sahip +, -
Bit tabanlı kaydırmalar	<<, >>
İlişkisel	Eşitsizlik <, >, <=, >=
	Eşitlik ==, !=
Bit tabanlı mantıksal	&, ^, |
Mantıksal	&&, ||
Koşul	?:
Atama	=, *=, /=, %=, +=, -=
	<<=, >>=, &=, ^=, |=
Sekans (sequence)	, (virgül)

Anahtar Kelimeler

Anahtar kelimeler, belirli C++ dil özelliklerini gerçeklerler. Anahtar kelimeler, değişkenler veya kullanıcı tarafından tanımlanan diğer programlama elemanları için isim olarak kullanılamazlar. Anahtar kelimelerin birçoğu hem C'de hem de C++'ta ortak olarak kullanılırlar; diğerleri ise C++'a özgüdür. Bazı derleyiciler ek anahtar kelimeler de destekleyebilirler. Bu anahtar kelimeler genellikle bir veya iki altçizgi karakteriyle başlar; `_cdecl` veya `__int16` örneklerinde olduğu gibi.

```
asm
auto

bool
break

case
catch
char
class
const
const_cast
continue

default
```



```
delete
do
double
dynamic_cast

else
enum
explicit
export
extern

false
float
for
friend

goto

if
inline
int

long

main
mutable

namespace
new

operator

private
protected
public

register
reinterpret_cast
return

short
signed
sizeof
static
static_cast
struct
switch

template
this
throw
true
```

```
try
typedef
typeid
typename

union
unsigned
using

virtual
void
volatile

wchar_t
while
```


EK C

MICROSOFT VISUAL C++

C

Bu ekte, Microsoft Visual C++ (MVC++) kullanarak, kitaptaki örneklerde olduğu gibi, konsol modunda uygulamalar geliştirmeyi göreceksiniz. Konular, MVC++ 6.0 sürümüne göre işlenmiştir.

MVC++'ın mevcut sürümü Standart C++'a (mükemmel olmasa da) gayet iyi bir şekilde uyar. Programın çeşitli sürümleri mevcuttur. Öğrenci sürümünün fiyatı 100 doların altındadır.

MVC++'ın sisteminizde yüklü olduğunu ve programı Windows Start (Başlat) düğmesi ve ilgili menü öğesini kullanarak çalıştırmayı bildiğinizi varsayacağız.

MVC++ ile çalışırken dosya uzantılarını (örneğin `.CPP`) bilmeniz gerekecek. Windows Explorer'da "Hide MS-DOS File Extensions for File Types That Are Registered" (Kayıtlı Dosya Tipleri İçin MS-DOS Dosya Uzantılarını Gizle) seçeneğinin işaretlenmemiş olmasına dikkat edin.

Ekran Öğeleri

MVC++ penceresi başlangıçta üç parçaya ayrılmıştır. Sol tarafta izleme panosu yer alır. Burada ClassView ve FileView adlı iki sekme vardır. Projeye başladıktan sonra ClassView sekmesinde programın sınıf hiyerarşisini, FileView sekmesinde de projede kullanılan dosyaları göreceksiniz. Hiyerarşileri açmak için artı işaretlerinin üzerine tıklayabilirsiniz. Dosyaları açmak için ise isimlerinin üzerine çift tıklamanız yeterli olacaktır.

Genelde ekranın en büyük kısmında doküman penceresi yer alır. Doküman penceresi çeşitli amaçlar için kullanılabilir. Örneğin, kaynak dosyalarınızı görüntülemek ve yardım dosyalarına bakmak için bu pencereyi kullanırsınız. Pencerenin altında ilave sekmeleri bulunan uzun pencere daha yer alır. Sekmelerin üzerinde Build, Debug ve diğer isimler vardır. Bu uzun pencere de program derleme gibi işlemler sırasında aldığınız mesajları gösterir.

Tek Dosyadan Oluşan Programlar

Microsoft Visual C++ kullanarak tek dosyadan oluşan bir konsol programını inşa etmek ve çalıştırmak gayet kolaydır. Burada karşımıza iki ihtimal çıkar: Kaynak metni ya hazırdır ya da bizim tarafımızdan yazılması gereklidir.

Her iki durumda da öncelikle açık projeniz olmamasını sağlayarak işe başlayın. (Projeleri birazdan ele alacağız.) File menüsüne tıklayın. Close Workspace öğesi etkin durumdaysa (gri değilse) bu öğeyi seçerek mevcut çalışma ortamını kapatın.

Mevcut Bir Dosyayı "Build" Etmek

.CPP kaynak dosyası mevcutsa, (bu kitaptaki örnek programların kaynaklarının elinizde olması gibi), File menüsünden Open'ı seçin. (Bunun Open Workspace ile aynı olmadığına dikkat edin.) İstediğiniz dosyayı bulmak için Open iletişim kutusunu kullanın. Dosyayı seçin ve Open düğmesine basın. Dosya doküman penceresine gelecektir. ("Fonksiyonlar" adlı Bölüm 5'deki `CIRCSTRC` ya da "Nesneler ve Sınıflar" adlı Bölüm 6'daki `CIRCLES` gibi Console Graphics Lite kullanan bir örnek programı derliyorsanız "Console Graphics Lite Programlarını İnşa Etmek" konusuna bakın.)

Kaynak dosyanızı derlemek (compile) ve bağlamak (link) için Build menüsünden Build öğesini seçin. Karşınıza çıkan iletişim kutusu bir Default Project Workspace (varsayılan proje çalışma alanı) oluşturmayı isteyip istemediğinizi soracaktır. Yes'e basın. Dosya derlenecek ve şayet varsa gerekli kütüphane dosyaları ile bağlanacaktır.

Programı çalıştırmak için Build menüsünden Execute öğesini seçin. Her şey yolunda giderse, bir ekranda konsol penceresi açılacak ve programın çıktısı burada görünecektir.

Program sonlandığında *Press any key to continue* (Devam etmek için herhangi bir tuşa basın) mesajını göreceksiniz. Derleyiciniz, her programın sonunda bu mesajın ekrana gelmesini sağlar. Bu sayede programın çıktısını incelemek için konsol penceresini istediğiniz kadar erkenden kapatabilirsiniz. Programla işiniz bittiğinde, çalışma ortamınızı File menüsünden Close Workspace öğesini seçerek kapatın. Tüm doküman pencerelerini kapatmayı isteyip istemediğiniz sorulduğunda uygun cevabı verin. Bu şekilde inşa ettiğiniz programları doğrudan MS-DOS'tan da çalıştırabilirsiniz. Bir MS-DOS penceresi açmak için Start düğmesine basın, Programs'dan MS-DOS Prompt öğesini seçin.

Karşınıza çıkan pencerede *C-prompt* denilen bir işaret göreceksiniz. Eğer Windows sürücünüz C ise bu işaret, C harfinin ardından mevcut klasörün adının yazılmasıyla oluşur. Bir klasörden diğerine geçmek için cd yazıp peşine de gitmek istediğiniz klasörün adını yazabilirsiniz. Burada cd, change directory (klasör değiştir) demektir. MVC++ ile derlenen programların `.EXE` dosyaları DEBUG adlı bir klasöre konulur. Bu klasör de proje dosyalarınızın bulunduğu klasörün altında yer alır. Bu kitaptaki örnekler dahil olmak üzere, bir programı çalıştırmak için `.EXE` dosyasının yer aldığı klasöre gittiğinize emin olun. Ardından MS-DOS imlecinde programın adını (uzantı kullanmadan) yazın. MS-DOS hakkında daha fazla bilgi edinmek için Windows'un yardım sistemine başvurabilirsiniz.

Yeni Bir Dosya Oluşturmak

MVC++'ta kendi .CPP dosyanızı yazmaya başlamak için öncelikle mevcut çalışma ortamınızı kapatın. Sonra File menüsünden New öğesini seçin ve Files sekmesine tıklayın. Buradan C++ Source File seçeneğini işaretleyin, dosyanın adını yazın. Ardından ya Location kutusuna dosyanın yolunu (path) yazın ya da dosyanın bulunmasını istediğiniz klasörü gezinerek işaretleyin. OK'e basın. Şimdi karşınıza boş bir doküman penceresi çıkmış olmalıdır. Programınızı bu pencereye yazın. Yeni dosyayı saklamak için File menüsünden Save As'i seçin. Daha önce olduğu gibi Build menüsünden Build öğesini seçin ve varsayılan çalışma alanını saklamayı isteyip istemediğinizle ilgili soruya Yes cevabı verin. Artık programınız derlenmiş ve bağlanmıştır.

Hatalar

Hata çıkarsa, bu durumla ilgili mesaj ekranın altındaki Build penceresinde belirir. (Bu pencereyi görünür kılmak için Build sekmesine tıklamanız gerekebilir.) Hata satırına çift tıklarsanız, kaynak dosyasında hatanın oluştuğu satırın yanında bir ok işareti belirecektir. Ayrıca, fare (mouse) imlecini Build penceresinde hata numarasının (örneğin C2143) üzerine getirip F1 tuşuna basarsanız, doküman penceresine hatayla ilgili bir açıklama gelecektir. Hata mesajı "0 error(s), 0 warning(s)" (0 hata, 0 uyarı) yazana kadar hataları düzeltip inşa işlemini tekrarlayabilirsiniz. Programı çalıştırmak için Build menüsünden Execute öğesini seçin.

Yeni bir program yazarken sık yapılan bir hata aşağıdaki ifadeyi unutmaktır:

```
using namespace std;
```

Eğer bu ifadeyi unutursanız, derleyici `cout` , `<<`, `endl` gibi ifadeleri tanımadığını söyleyecektir.

Yeni bir program üzerinde çalışmaya başlamadan önce File menüsünden Close Workspace öğesini seçmeyi unutmayın. Bu sayede temiz bir çalışma ortamı elde etmiş olursunuz. Daha önceden inşa etmiş olduğunuz bir programı açmak için File menüsünden Open Workspace öğesini seçin, programın bulunduğu klasöre gidin, `.DSW` uzantılı dosyanızı buradan seçin.

Run-Time Type Information (RTTI)

"Akışlar ve Dosyalar" adlı Bölüm 12'deki `EMPL_IO.CPP` gibi bazı programlar RTTI kullanırlar. MVC++'ta bu özelliğin çalışması için bir derleyici seçeneğini etkin duruma getirmeniz gerekli-dir. Project menüsünden Settings öğesini seçin ve C/C++ sekmesine tıklayın. Category adlı listeden C++ Language öğesini işaretleyin. Enable Run-Time Type Information kutusunu işaretleyln. OK'e basın. Bu sayede, bazıları yanıltıcı olan, çeşitli derleyici ve bağlayıcı hataları artık karşınıza çıkmayacaktır.

Birden Fazla Dosyadan Oluşan Programlar

Program inşa etmenin hızlı ve fazlaca rafine olmayan bir yolunu gördük. Bu yaklaşım, tek dosyadan oluşan programlar için geçerlidir. Birden fazla dosyadan oluşan projelerde işler bir parça karmaşıklaşır. Konuya öncelikle *workspace* (çalışma alanı) ve *project* (proje) terimlerini açıklayarak gireceğiz.

Projeler ve Çalışma Alanları

MVC++ *workspace* denilen bir kavram kullanmaktadır. Çalışma alanı anlamına gelen bu kavram, "proje"nin bir seviye üstüne denk düşen bir soyutlamadır, yani tanımdır. Bir çalışma alanında çok sayıda proje bulunabilir. Çalışma alanı, bir klasör ile çeşitli konfigürasyon dosyalarından oluşur. Her projenin kendi klasörü olabilir ya da tüm projelerin dosyaları çalışma alanının klasöründe durabilir.

Kavramsal olarak, en azından bu kitapta yer alan küçük programlar için, en kolay yaklaşım her projenin kendi çalışma alanında durduğunu varsaymak olacaktır. Burada bizim varsayımımız da bu yönde olacaktır.

Bir proje, geliştirmekte olduğunuz bir uygulamaya (programa) karşılık gelir. Proje; uygulamayı oluşturmak için gerekli tüm dosyalar ile bu dosyaların ne şekilde birleştirileceğine dair bilgiden ibarettir. Bir projeyi inşa etmenin neticesinde genellikle kullanıcının çalıştırabileceği tek bir `.EXE` dosyası oluşur. (Başka sonuçlar da mümkündür; örneğin bir `.DLL` dosyası da oluşturulabilir.)

Projeyi Geliştirmek

Varsayalım ki, yeni bir projeye dahil etmek istediğiniz dosyalar mevcut olsun ve bunlar belli bir klasörde bulunsunlar. File menüsünden New öğesini seçin ve New iletişim kutusunda Projects sekmesine tıklayın. Listeden Win32 Console Application öğesini seçin. Location kutusuna istediğiniz klasörün yolunu yazın, ama klasörün kendi adını yazmayın. Dosyaları içeren klasörün adını Project Name kutusuna yazın. (Location alanının sağındaki düğmeye basarak aradığınız klasörü gezinme yöntemiyle de bulabilirsiniz. Ancak klasörün yolunu bu şekilde bulduğunuzda Location kutusunda klasörün adını silmeniz ve geriye sadece klasörün yolunu bırakmanız gereklidir. Çünkü klasörün adını proje adı olarak zaten başka şekilde sağlıyorsunuz.) Create New Workspace kutusunun seçilmiş bulunduğuna emin olun ve OK'e basın.

Örneğin, dosyalar `C:\Book\Ch13\Elev` adresinde ise, önce Location kutusuna `C:\Book\Ch13\` yazmalı, Project Name alanına ise `Elev` girmelisiniz. Proje adını yazdığınızda bu projenin yoluna otomatik olarak eklenir. (Eğer proje adı yolun içinde zaten varsa, bunu bir de Project Name alanına yazarsanız `C:\Book\Ch13\Elev\Elev` gibi yanlış bir durum ortaya çıkar.) Bu sefer bir başka iletişim kutusu ile karşılaşacaksınız. An Empty Project düğmesinin seçilmiş bulunduğuna dikkat edin ve Finish'e basın. Bir sonraki iletişim kutusuna da OK deyin.

Bu aşamada projeyi tanımlayan `.DSP`, `.DSW` ve benzeri çeşitli dosyalar ile ortaya çıkacak nihai `.EXE` dosyasının duracağı `DEBUG` alt klasörü proje klasörüne eklenmiş olacaktır.

Kaynak Dosyalarını Eklemek

Şimdi de projeye kaynak dosyaları eklemeniz gerektiğini düşünelim. Bunlar, File sekmesinden bulabileceğiniz `.CPP` ve `.H` dosyaları olacaktır. Project menüsünden Add To Project öğesini seçin. İstediğiniz dosyaları ekleyin ve OK deyin. Seçtiğiniz dosyaları FileView sekmesine gidip projenin artı işaretine basarak inceleyebilirsiniz. ClassView sekmesine basarak da üye fonksiyonlar ve bunların özellikleri de dahil olmak üzere sınıf yapısını görebilirsiniz. Bir dosyayı incelemek ve düzenlemek (edit etmek) için FileView penceresinde o dosyaya ait simgenin üzerine çift tıklamanız gereklidir. Aynı işi, File menüsünden Open deyip dosyayı seçerek de yapabilirsiniz.

Başlık Dosyalarını Bulmak

Projeniz başlık (header) dosyaları kullanıyor olabilir. Örneğin Console Graphics Lite kullanan programlar `MSOFTCON.H` başlık dosyasına ihtiyaç duyar. (Başlık dosyaları genellikle `.H` uzantılıdır.) Başlık dosyalarının (bunları File sekmesi üzerinden izlemek istemediğiniz sürece) projeye eklenmesi gerekmez. Ancak derleyiciniz başlık dosyalarınızın nerede olduğunu bilmek zorundadır. Eğer bu dosyalar kaynak dosyalarınız ile aynı klasörde ise, yer belirtmenize gerek kalmaz. Başlık dosyalarınız bunun dışında bir yerde ise, o yeri derleyicinize bildirmeniz gereklidir.

Tools menüsünden Options öğesini seçin. Directories sekmesine gidin. Show Directories For listesinde Include Files seçeneğini işaretleyin. Burada derleyicinin kendi başlık dosyalarının bulunduğu klasörleri göreceksiniz. Listenin en altında yer alan noktalı kutuya çift tıklayın. Ardından, yandaki düğmeye basıp, başlık dosyalarınızın durduğu klasörü gezinerek bulun. Noktalı kutunun yerine yeni bir yol adı gelmiş olacaktır. OK'e basın. Alternatif olarak yolun tam şeklini Location kutusuna kendiniz yazabilirsiniz.

Projeleri Saklamak, Kapatmak ve Açmak

Projeyi saklamak için Save Workspace deyin. Projeyi kapatmak için Close Workspace'i seçin. (Close All Document Windows sorusuna Yes diye cevap verin.). Mevcut bir projeyi açmak için File menüsünden Open Workspace öğesini seçin, istediğiniz klasöre gidin, `.DSW` dosyasını seçin ve Open düğmesini tıklayın.

Derleme ve Bağlama

Tek dosyalı programlarda olduğu gibi, çok dosyalı programları derleme, bağlama ve çalıştırmanın en kolay yolu Build menüsünden Execute öğesini seçmektir. Alternatif olarak projenizi çalıştırmadan Build menüsünden Build'i seçerek derleyip bağlamanız da mümkündür.

Console Graphics Lite Programlarını İnşa Etmek

Console Graphics Lite ("Console Graphics Lite" adlı Ek E'de açıklanmıştır) kullanan programları inşa etmek için diğer örnek programlarımıza göre ilave bazı işlemler yapılması gerekmektedir. Her şeyden önce `MSOFTCON.H` ve `MSOFTCON.CPP` dosyalarına ihtiyacınız olacaktır. Kitabımıza özgü olan bu dosyalar, yayıncının Başlangıç bölümünde adresi verilen Web sitesinden çekilebilir.

- "Mevcut Bir Dosyayı 'Build' Etmek" başlığı altında açıklandığı üzere, kaynak dosyasını açın. Bu dosyada `#include msoftcon.h` satırı da yer almalıdır.
- Build menüsünden Build'i seçin. Varsayılan bir proje çalışma alanı oluşturmayı isteyip istemediğiniz sorulduğunda Yes diye cevap verin. Yeni bir proje oluşturulacaktır ama derleyici `MSOFTCON.H` adlı başlık dosyasını bulamadığından hata mesajı verecektir. Bu dosyada grafik fonksiyonlarının tanımları yer almaktadır.
- Kolay bir çözüm, `MSOFTCON.H` dosyasını projenizin klasörüne kopyalamaktır. Daha rafine bir yaklaşım ise derleyiciye bu başlık dosyasının yerini bildirmektir. Bunun için "Başlık Dosyalarını Bulmak" başlığı altında anlatılanları uygulayabilirsiniz.
- Şimdi dosyanızı yeniden inşa etmeye çalışın. Artık derleyici başlık dosyasını bulmaktadır ama bu kez de bağlayıcı grafik fonksiyonlarının yerini bilmediğinden çeşitli bağlayıcı hataları ortaya çıkmaktadır. Aranan kod `MSOFTCON.CPP` dosyasında yer almaktadır. Bu dosyayı "Kaynak Dosyalarını Eklemek" başlıklı paragrafta anlatıldığı gibi proje klasörünüzü ekleyin.

Artık programınız hatasız bir şekilde derleniyor ve bağlanıyor olmalıdır. Build menüsünden Execute seçeneğine giderek programı çalıştırabilirsiniz.

Hata Ayıklama

"Döngüler ve Kararlar" adlı Bölüm 3'te döngülerin nasıl çalıştığını aydınlatmak için hata ayıklayıcı (debugger) kullanımını tavsiye ediyoruz. Microsoft Visual C++'ta bunun nasıl yapılacağını şimdi burada ele alacağız. Aynı adımları, programınızın hatalı çalıştığı durumlarda da bu kez hataları ayıklamak için kullanabilirsiniz. Burada sadece tek dosyalı programları ele alacağız. Ancak, belli bazı değişikliklerin dışında buradaki yaklaşım daha büyük ve çok dosyalı programlarda da geçerlidir.

Önce programınızı normal bir şekilde inşa edin. Ortaya çıkan derleyici ve bağlayıcı hatalarını giderin. Programın kaynak kodunun Edit penceresinde gösterildiğinden emin olun.

Adım Adım İlerleme

Hata ayıklayıcıyı çalıştırmak için F10 tuşuna basmanız yeterli olacaktır. Şimdi, kaynak kodunun yan tarafında main'in arkasından gelen açılış parantezine işaret eden sarı bir ok göreceksiniz.

Programın başından değil de başka bir noktadan başlamak istiyorsanız, imleci başlamak istediğiniz yere getirin. Sonra (Build menüsünün yerine geçmiş olan) Debug menüsünden Start Debug öğesini ve ardından da Run to Cursor öğesini seçin. Sarı ok, başlangıç almak istediğiniz ifadenin yanında belirecektir.

F10 tuşuna basın. Bu, hata ayıklayıcının bir sonraki çalıştırılabilir (executable) ifadeye geçmesini sağlar. Sarı ok bulunduğunuz yeri göstermeye devam eder. F10'a her basışınızda bir sonraki ifadeye gidersiniz. Şayet bir döngüde iseniz, sarı okun döngüdeki ifadeleri işletirken aşağı-yukarı ilerlediğini görürsünüz.

Değişkenleri İzlemek

Programınızdaki değişkenlerin adım adım işleyişte aldığı değerleri izleyebilirsiniz. Yerel değişkenlerin değerlerini görmek için ekranın sol alt köşesindeki pencerede Locals sekmesini tıklayın. Auto sekmesinde derleyicinin seçtiği değişkenleri görebilirsiniz.

Kendi seçtiğiniz değişkenleri izlemek istiyorsanız, bunların adlarını ekranın sağ alt köşesindeki Watch penceresine ekleyin. Bunu yapmak için kaynak kodunda bir değişkenin üzerine gi-

din ve farenin sağ tuşuna basın. Karşınıza çıkan menüden QuickWatch öğesini seçin. Oluşan QuickWatch iletişim kutusunda AddWatch'a basın. Değişkeniniz ve onun mevcut değeri Watch penceresine eklenmiş olacaktır. Şayet bir değişken geçerli olduğu alanının dışında ise, örneğin tanımlandığı noktadan önce onun değerini izlemeye çalışıyorsanız, Watch penceresinde değişkenin değeri yerine hata mesajı görürsünüz.

Fonksiyonların İçine Girmek

Programınızda fonksiyon kullanılıyorsa, F11 tuşuna basarak fonksiyona *step into* yapabilirsiniz. (Yani fonksiyonun içine girip onun ifadelerini adım adım işletebilirsiniz.) Fonksiyon adının üzerinde iken F10 tuşuna basarsanız *step over* olur; yani fonksiyon tek bir ifade olarak değerlendirilir, içine girmeden çalıştırılır ve bir sonraki ifadeye geçilir. `cout <<` gibi kütüphane fonksiyonlarının nasıl çalıştığını izlemek için F11'e basarsanız, kütüphane fonksiyonunun kaynak kodu üzerinden ilerleyebilirsiniz. Bu uzunca bir işlem olabilir, dolayısıyla gerçekten ilginizi çekmiyorsa bu yola gitmekten kaçınmanızda yarar var. Yani, söz konusu fonksiyonun nasıl çalıştığını araştırmayı isteyip istemediğinize bağlı olarak F11 ve F10 arasında mantıklı bir geçiş yapmanız gereklidir.

Durak Noktaları

Durak noktaları (breakpoint) programı istediğiniz noktada durdurmanıza imkan verir. Peki bunun yararı nedir? "Run to Cursor" ile programı belli bir noktaya kadar çalıştırmayı daha önce öğrenmiştik. Bazen öyle durumlar olur ki, programı birden fazla yerde durdurmanız gerekir. Örneğin, programı belli bir `if` ifadesinden ve de ona karşılık gelen `else` ifadesinden sonra durdurmanız gerekebilir. İşte durak noktaları bu problemi çözer, çünkü ihtiyacınız olduğu kadar çok sayıda durak noktası kullanabilirsiniz. (Durak noktalarının, burada ayrıntısına girmeyeceğimiz başka gelişkin özellikleri de vardır.)

Kaynak kodunuza bir durak noktası eklemek için önce imleci durmak istediğiniz satırın üzerine getirin. Sonra sağ fare tuşuna basın. Karşınıza çıkan menüden Insert/Remove Breakpoint öğesini seçin. Kodun sol tarafında kırmızı bir daire ortaya çıkacaktır. Artık programı ne zaman çalıştırsanız (örneğin Debug/Go ile), program belirlediğiniz durak noktasında duracaktır. Program bu şekilde durduğu zaman değişkenleri inceleyebilirsiniz, kodu adım adım çalıştırabilirsiniz ya da bir başka durak noktasına kadar programı devam ettirebilirsiniz.

Bir durak noktasını kaldırmak için üzerine gidin, sağ fare tuşuna basın ve menüden Remove Breakpoint öğesini seçin.

Debugger'ın daha başka pek çok özelliği vardır ama burada anlattıklarımız başlangıç için size yeterli olacaktır.

BORLAND C++BUILDER

Bu ekte, Borland C++Builder kullanarak, kitaptaki örneklerde olduğu gibi, konsol modunda uygulamalar geliştirmeyi göreceksiniz.

C++Builder, Borland'ın en ileri geliştirme ürünüdür. Kitabımız basıma hazırlandığı sırada, C++Builder, piyasada Standard C++'a en fazla uyan ürün konumundaydı. Programın öğrenci sürümü 100 doların altında bir fiyata satılmaktadır. (Sadece derleyici içeren bir sistemi Borland'ın Web sitesinden ücretsiz olarak çekmek de mümkündür. Bu ücretsiz derleyiciyi kullanırken kaynak kodunu yazmak için Notepad ya da ona benzer bir metin düzenleyicisine ihtiyacınız olur.) Bu bölümdeki bilgiler, C++Builder 5.0 sürümüne göredir.

C++Builder'ın sisteminizde yüklü olduğunu ve programı Windows Start (Başlat) düğmesi ve ilgili menü öğesini kullanarak çalıştırmayı bildiğinizi varsayacağız.

C++Builder ile çalışırken dosya uzantılarını (örneğin `.CPP`) bilmeniz gerekecek. Windows Explorer'da "Hide MS-DOS File Extensions for File Types That Are Registered" (Kayıtlı Dosya Tipleri İçin MS-DOS Dosya Uzantılarını Gizle) seçeneğinin işaretlenmemiş olmasına dikkat edin.

Örnek Programları C++Builder'da Çalıştırmak

Kitaptaki programları C++Builder'da çalıştırmak için küçük bazı değişiklikler gereklidir. Şimdi bu değişiklikleri kısaca özetleyelim.

Örnek programların çoğunluğunu herhangi bir değişiklik yapmadan MS-DOS penceresinde (Start/Programs/MS-DOS Prompt) derleyip çalıştırabilirsiniz. Ancak, aynı programları C++Builder'ın içinde Run menüsünden Run komutunu seçerek çalıştırmak isterseniz, konsol penceresini yeterli bir süre açık tutabilmek için programın sonuna bir ifade eklemeniz gerekir. Bunu iki adımda yapabilirsiniz:

- `main()`'de programın son `return` ifadesinden hemen önce bir `getch()` ifadesi ekleyin. Bu sayede programın çıktısını görebilirsiniz.
- `main()`'in başına `#include <conio.h>` ifadesini ekleyin. Bu, `getch()` için gereklidir.

Şayet inşa ettiğiniz program Console Graphics Lite fonksiyonlarını (bu fonksiyonlar Ek E'de "Console Graphics Lite" başlığı altında anlatılmıştır) kullanıyorsa, birkaç ilave adım daha atmanız gerekecektir. Bu adımlar bu ek içinde özetlenmiştir.

Bu ekin devamında, bu konuları daha ayrıntılı olarak ele alacağız ve konsol modu programlarını düzenlemek, derlemek, bağlamak ve çalıştırmak için C++Builder'ın nasıl kullanıldığından bahsedeceğiz.

Ekranı Temizlemek

C++Builder çalıştırılmaya başlandığında, konsol modundaki programlar için ihtiyacınız olmayacak bazı ekran nesnelerini gösterir. Ekranın sağında Form1 olarak adlandırılan bir pencere göreceksiniz. Bu pencereyi kapatmak için kapatma düğmesini (sağ üst köşedeki X'e) tıklayın. Aynı şekilde, Object Inspector'a da ihtiyaç duymayacaksınız; bu nedenle, bunun da kapatma düğmesine tıklayın. C++Builder'ı her çalıştırışınızda bu unsurlardan kurtulmanız gerekecektir.

Form1 penceresini kaldırınca, bu pencerenin altında, içinde bir miktar C++ kodu olan bir başka pencereyle karşılaşacaksınız. Bu pencere *kod editör* penceresi olarak adlandırılır. Kod editör penceresi, kaynak dosyalara bakabileceğiniz ve kendi programlarınızı yazacağınız yerdir. Ancak, kod editör penceresi `UNIT1` adında bir dosya ile yüklenir ve bu dosya sizin için gerekli

değildir. Bu pencereyi ortadan kaldırmak için kapatma düğmesine tıklayın; eğer iletişim kutusu değişiklikleri kaydetmek isteyip istemediğinizi sorarsa No yanıtını verin.

C++Builder, ihtiyacınızdan daha fazla araç çubuğu ile çalışmaya başlar. Muhtemelen Standard ve Debug araç çubuklarına ihtiyacınız olur. View menüsünün en altında yer alan Toolbars'ı seçerek ve istemediğiniz araç çubuklarının işaretlerini kaldırarak diğerlerinden kurtulun.

Yeni Bir Proje Geliştirmek

C++Builder (diğer modern derleyiciler gibi) programları geliştirirken *proje* bazında düşünür. Bir proje, bir veya daha fazla kaynak dosyadan oluşur. Proje ayrıca, burada ilgilenmemiz gerekmeyen başka türde birçok dosyayı daha içerebilir, mesela sistem kaynaklarıyla ilgili dosyalar ve tanım dosyaları. Bir projenin neticesi genellikle kullanıcının çalıştırabildiği tek bir `.EXE` dosyasıdır.

Yeni bir proje başlatmak için File menüsünden New... seçeneğini seçin. Ekranda New Items adında bir iletişim kutusu göreceksiniz. New sekmesine tıklayın (eğer gerekiyorsa). Sonra, Console Wizard simgesine çift tıklayın. Karşınıza çıkan iletişim kutusunda Source Type'ın C++ olduğundan ve Console Application seçeneğinin seçili olduğundan emin olun. Use VCL, Multi Threaded ve Specify Project Source seçeneklerinin işaretini kaldırın. OK'e tıklayın. Eğer bir başka iletişim kutusu değişiklikleri `PROJECT1`'e kaydetmek isteyip istediğinizi sorarsa No'ya tıklayın. Yeni kod editör penceresinde aşağıdaki kaynak dosyanın göründüğünü göreceksiniz:

```
//---------------------------------------------------------------

#pragma hdrstop

//---------------------------------------------------------------

#pragma argsused
int main(int argc, char **argv [])
{
      return 0;
}
//---------------------------------------------------------------
```

Bu, konsol modundaki bir programının iskeletidir. Bu programdaki bazı satırlara ihtiyaç duymayacaksınız; bazı yeni satırları da eklemeniz gerekecek. Biz bu değişiklikleri sizin adınıza yapacağız ve programın çalışıp çalışmadığını görmeniz için ekrana birkaç satır yazan bir ifade ekleyeceğiz. İşte sonuç:

```
// test1.cpp
#include <iostream>
#include <conio.h>
using namespace std;
//#pragma hdrstop          //gerekli degil
//#pragma argsused         //gerekli degil
int main()                 //argumanlar gerekli degil
   {
   cout <<"Happy are they whose programs"
        <<"compile the first time.";
   getch();
   return 0;
   }
```

`#pragma` direktiflerinin ikisi de gerekli değildir; `main()`'in argümanlarına da ihtiyacınız yoktur.

Orijinal iskelet programını bu değişiklikleri yapmadan çalıştırırsanız, konsol penceresinin, içindekileri görebilecek kadar yeterli bir süre kalmadığını fark edeceksiniz. Önceden bahsettiğimiz gibi, programın sonuna, `return`'den hemen önce şu ifadeyi

```
getch();
```

ekleyerek bu sorunu giderebilirsiniz. Bu, programın bir tuş basılana kadar beklemesine neden olur, böylece konsol penceresi, bir tuşa basılana kadar kalır. `getch()` fonksiyonu `CONIO.H` başlık dosyasını gerektirir, dolayısıyla bu başlık dosyasını programınızın başına dahil etmeniz gerekecek.

Eğer kendi programınızı geliştiriyorsanız, bu iskelet programla başlayabilir ve bunun üzerinde değişikler yapabilirsiniz. Eğer mevcut bir dosya ile başlıyorsanız, "Mevcut Dosyalar ile Başlamak" başlıklı konuyu okuyun.

Bir Projeye İsim Verip, Projeyi Kaydetmek

Hem kaynak dosyanızı hem de kaynak dosyanızın içinde olduğu projenizi kaydetmeniz ve yeniden adlandırmanız gerekecektir. Derleyici, kaynak dosyayı otomatik olarak `UNIT1.CPP` olarak adlandırır. Bunu kaydetmek ve yeniden adlandırmak için File menüsünden Save As seçeneğini seçin, projenizin yer aldığı klasöre yönlenin, dosyaya isim verin (`.CPP` uzantısını koruyarak) ve Save'i tıklayın.

Bir projeyle ilgili bilgiler `.BPR` uzantılı bir dosya içine kaydedilir. Yani, bir projeyi kaydettiğiniz zaman aslında hem `.CPP` dosyasını (veya dosyalarını) hem de `.BPR` dosyasını kaydediyorsunuz demektir. Yeni bir projeyi ilk kez oluşturduğunuzda, bu proje `PROJECT1` (daha yüksek numaralı da olabilir) olarak adlandırılır.

Projeyi kaydedip, ismini değiştirmek için File menüsünden Save Project As seçeneğini seçin, dosyayı saklamak istediğiniz klasöre doğru yönlenin, projeye vermek istediğiniz ismi yazın (`.BPR` uzantısı bu ismin peşinden gelmelidir) ve Save'i tıklayın.

Mevcut Dosyalarla Başlamak

Şimdi de, bu kitaptaki örneklerde olduğu gibi (bu örnekleri Web'den indirdiğinizi varsayıyoruz), kaynak dosyası önceden mevcut iken bir projeyi nasıl oluşturacağınızı görelim. Burada ele alacağımız durumlar, bir tek kaynak dosyadan ibaret programlara uygulanır.

Kaynak dosyanızın `MYPROG.CPP` olarak adlandırıldığını varsayalım. Bu dosyanın, dosyayı inşa etmek istediğiniz klasörün içinde bulunduğundan emin olun. File menüsünden Open seçeneğini seçin, dosyayı seçin ve Open'a basın. Dosya, kod editör penceresinde görünecektir. Bir iletişim kutusu, "Would you like to create a project so this file can compile and run? (Bu dosyanın derlenip çalıştırılabilmesi için bir proje oluşturmak ister misiniz?)" diye sorar. Buna Yes yanıtını verin. Artık, bir proje oluşturulmuştur.

Şimdi bu projeye yeni biri isim vermeniz gerekir; bu nedenle, File menüsünden Save Project As seçeneğini seçin. `PROJECT1.BPR` ismini projenizin ismi ile değiştirin. Projenizin ismi genellikle programınızın ismidir: `MYPROG.BPR`. Save'i tıklayın. Tüm yapmanız gereken, bundan ibaret.

Derleme, Bağlama ve Çalıştırma

Çalıştırılabilir bir program inşa etmek için Project menüsünden Make veya Build'i seçin. Bu, `.CPP` dosyanızın `.OBJ` dosyası olarak derlenmesini ve `.OBJ` dosyasının bir `.EXE` dosyasına bağlanmasını (çeşitli kütüphane dosyalarıyla birlikte) sağlar. Örneğin, eğer `MYPROG.CPP` dosyasını derliyorsanız, sonuç `MYPROG.EXE` olacaktır. Derleyici veya bağlayıcı hataları varsa bunlar ekranda gösterilir. Bu hataların tümünü ortadan kaldırana kadar programınızı düzenleyin.

C++Builder'dan Çalıştırmak

Şayet programınızı önceden bahsettiğimiz gibi `getch()` ekleyerek düzeltmişseniz, programınızı Run menüsünden Run seçeneğini işaretleyerek doğrudan C++Builder'ın içinden derler, bağlar ve çalıştırabilirsiniz. Eğer hiç hata yoksa, programınızın çıktısının yer aldığı konsol penceresi ekranda görünecektir.

MS-DOS'tan Çalıştırmak

Programları doğrudan MS-DOS'tan da çalıştırabilirsiniz. Windows'ta Start düğmesine tıklayıp, önce Programs'ı, ardından da MS-DOS Prompt seçeneğini seçerek MS-DOS için bir pencere elde edebilirsiniz.

Elde ettiğiniz pencerede C-*prompt* denilen bir şey göreceksiniz. C-prompt, C harfinin ardından genellikle mevcut klasörün isminin yazılmasıyla oluşur. Bir klasörden diğerine `cd` (Change Directory –Klasör değiştir, anlamında) ile birlikte yeni klasörün adını yazarak gezinebilirsiniz. Bir program çalıştırmak için (bu kitaptaki örnekler de buna dahildir) ilgili `.EXE` dosyası ile aynı klasörde olduğunuzdan emin olun ve programın ismini tuşlayın (dosya uzantısını yazmayın). Windows yardım sistemini kullanarak MS-DOS hakkında daha fazla bilgi edinebilirsiniz.

Önceden Derlenmiş Başlık Dosyaları

Derleme işlemini heyecan verici ölçüde hızlandırabilirsiniz. Bunun için Project menüsünden Options seçeneğini seçin, ardından Compiler sekmesini seçin ve Use Precompiled Headers'ın üzerine tıklayın. Kısa bir programda derleme zamanının büyük kısmı C++ başlık dosyalarını (mesela `iostream`) derlemekle geçer. Precompiled Headers seçeneğini kullanmak bu başlık dosyalarının programınızla birlikte her seferinde yeniden derlenmesi yerine, sadece bir kez derlenmesine neden olur.

Projeleri Kapatmak ve Açmak

Bir projeyle işiniz bittiği zaman File menüsünden Close All seçeneğini seçerek projenizi kapatabilirsiniz. Önceden kaydedilmiş bir projeyi açmak için ise File menüsünden Open'ı seçin, ilgili `.BPR` dosyasına yönelin ve bu dosyayı çift tıklayın.

Projenize Bir Başlık Dosyası Eklemek

Ciddi C++ programlarının bir çoğu bir veya daha fazla, kullanıcı (tarafından yazılmış) başlık dosyalarını (birçok kütüphane dosyasına, mesela `IOSTREAM` ve `CONIO.H`'e ek olarak) kullanırlar. Şimdi bir başlık dosyasının nasıl oluşturulduğunu görelim.

Yeni Bir Başlık Dosyası Hazırlamak

File menüsünden New... seçeneğini seçin, New sekmesinin seçili olduğundan emin olun ve Text simgesine çift tıklayın. `FILE1.TXT` başlıklı bir dosyanın içinde yer aldığı bir kod editör penceresi göreceksiniz. Dosyanızın metnini bu pencere içinde yazın ve File menüsünden Save As seçeneğini kullanarak, uygun bir isim ve peşinden `.H` dosya uzantısı vererek dosyanızı kaydedin. Bu başlık dosyasını kaynak (`.CPP`) dosyalarınızla aynı klasör içine kaydedin. Yeni dosya ismi, kod editör penceresindeki diğer dosyaların yanında bir sekme üzerinde yer alacaktır. Sekmelere tıklayarak bir dosyadan diğerine geçebilirsiniz.

Mevcut Bir Başlık Dosyasını Düzenlemek

Mevcut bir başlık dosyasını açmak için File menüsünden Open'ı seçin ve Files of Type listesinden de Any File (*.*) seçeneğini seçin. Bundan sonra, başlık dosyanızı listeden seçebilirsiniz.

Başlık dosyası için `.CPP` dosyanıza `#include` ifadesini yazdığınız zaman dosya ismini çift tırnak içine aldığınızdan emin olun:

```
#include "myHeader.h"
```

Çift tırnak işareti, başlık dosyasını kaynak dosyalarınızla aynı klasörde aramasını derleyiciye bildirir.

C++Builder'a Başlık Dosyasının Yerini Söylemek

Bir `.H` dosyası ekliyorsanız, derleyici bunu nerede bulacağını bilmelidir. Eğer bu dosya diğer dosyalarınızla aynı klasördeyse bir şey yapmanıza gerek kalmaz.

Ancak, eğer `.H` dosyanız farklı bir klasördeyse, C++Builder'a bunu nerede bulması gerektiğini söylemeniz gerekir. (Bu, konsol modunda grafikler için gerekli olan `BORLACON.H` dosyası için de geçerlidir. Tabii, bu dosyayı eğer proje dosyanızın içine kopyalamamışsanız bu işlem söz konusudur.) Project menüsündeki Options seçeneğine gidin ve Directories/Conditionals sekmesini seçin. Directories bölümünde Include Path listesinin sağındaki üç noktalı düğmeye tıklayın. Directories iletişim kutusu ekranda görünecektir.

Directories iletişim kutusunun alt kısmındaki alana `.H` dosyasının bulunduğu klasörün dosya yolunu (pathname) tam olarak yazın. Bu dosya yolunu, (include) dosya yolu listesine yerleştirmek için Add düğmesine tıklayın. Sonra da, iletişim kutularını kapatmak için OK'e iki kere daha tıklayın.

Başlık dosyalarını Project menüsündeki Add to Project seçeneği ile projenize eklemeye çalışmayın.

Birden Fazla Kaynak Dosyasından Oluşan Projeler

Gerçek uygulamalar ve bu kitaptaki bazı örnekler, birden fazla kaynak (`.CPP`) dosyası gerektirir. Bu arada, C++Builder'da kaynak dosyalar genellikle *birim* (unit) olarak adlandırılır; birim, bu ürüne özgü bir terimdir. Birçok C++ geliştirme ortamında ise dosyalar, dosya veya *modül* olarak adlandırılır.

Ek Kaynak Dosyalarını Hazırlamak

Ek `.CPP` dosyalarını da başlık dosyalarını oluşturduğunuz gibi hazırlıyorsunuz: File/New seçeneğini seçin ve New iletişim kutusunda Text simgesine çift tıklayın. Kaynak kodu yazın ve dosyayı kaydetmek için Save As'i kullanın. Save As'i kullanırken Type listesi olarak Save File'dan C++Builder Unit'i (`.CPP`) seçtiğinizden emin olun. Bu otomatik olarak `.CPP` uzantısını ekleyecektir, yani sizin yazmanız gereken sadece dosyanın isminden ibarettir. Eğer bunu başaramazsanız ve dosya isminden sonra `.CPP` uzantısını eklerseniz, dosya C++Builder birimi olarak tanınmayacaktır.

Projenize Mevcut Kaynak Dosyalarını Eklemek

Az önce bahsedildiği gibi yeni bir kaynak dosyası oluşturmuş olabilirsiniz ya da zaten mevcut olan bir tanesini kullanıyor olabilirsiniz, mesela, Console Graphics Lite programları için kullanılan `BORLACON.CPP` dosyası gibi. Projenize bir kaynak dosyası eklemek için Project menüsünden Add to Project seçeneğini seçin, ilgili klasöre yönelin (eğer gerekiyorsa) ve listeden dosya ismini seçin. Sonra Open'a tıklayın. Bu işlem, bu dosyanın, projenin bir parçası olduğunu C++Builder'a bildirir.

Birden fazla kaynak dosyası (eğer ekranı kaplayan pencerelerin içindelerse) Edit penceresinde sekmelerle gösterilir, böylece bir dosyadan diğerine çabucak geçebilirsiniz. Bu dosyaları tek tek açıp kapayabilirsiniz; bu nedenle, dosyaların tümünün aynı anda ekranda olması gerekmez.

Proje Yöneticisi

View menüsünden Project Manager'ı seçerek hangi kaynak dosyaların projenin parçası olduğunu görebilirsiniz. Windows Explorer'da gösterilene benzer şekilde dosyalar arası ilişkilerin şemasını göreceksiniz. Proje simgesinin yanındaki artı işaretine tıklayınca projenin tüm kaynak dosyaları ekranda gösterilecektir. Projeye henüz eklediğiniz dosya da bunların içinde olmalıdır.

Project Manager'da bir dosyanın üzerine sağ fare tuşuyla tıklarsanız, kontekst menüsü Open, Save, Save As ve Compile'ı da içeren bir takım seçenekler sunacaktır. Bu, her kaynak dosya üzerinde bu işlemleri ayrı ayrı gerçekleştirmek için kullanışlı bir yöntemdir.

Birden fazla dosyadan oluşan bir programda, Project menüsünden Compile Unit seçeneğini seçerek dosyaları tek tek derleyebilirsiniz. Project menüsünden Make'i seçerek kaynak dosyalarının tümünü derleyip bağlayabilirsiniz. Bu, son derleme işleminden bu yana yalnızca değişen kaynak dosyalarının yeniden derlenmesine neden olacaktır.

Console Graphics Lite Programları

Şimdi de, Console Graphics Lite paketini kullanan programları nasıl inşa edeceğinizi görelim. Bu, "Fonksiyonlar" adlı Bölüm 5'teki `CIRCSTRC` programı ve "Nesneler ve Sınıflar" adlı Bölüm 6'daki `CIRCLES` programı türünde programları kapsar.

- Program ismini proje ismi olarak kullanarak, fakat bir de `.BPR` uzantısı ekleyerek önceden anlatıldığı gibi, yeni bir proje oluşturun.
- Kaynak dosyası içinde `#include <msoftcon.h>` ifadesini `#include <borlacon.h>` olacak şekilde değiştirin.

- **BORLACON.H** ve **BORLACON.CPP** dosyalarını proje dosyanızın içine kopyalayın (sadece taşımakla kalmayın.) (Ya da önceden belirtildiği gibi, derleyiciye başlık dosyasının nerede olduğunu söyleyin.)
- Önceki bölümde "Mevcut Kaynak Dosyaları Projenize Eklemek" başlıklı ekte anlatılan direktifleri izleyerek **BORLACON.CPP** kaynak dosyasını projenize ekleyin.
- Görüntüyü ekranda muhafaza etmek için **main()**'in sonundaki **return** ifadesinin hemen önüne **getch()** ekleyin.
- **getch()**'i desteklemek için programınızın başına **#include <conio.h>** satırını ekleyin.

Artık Console Graphics Lite programlarını tıpkı diğer programlar gibi derler, bağlar ve çalıştırabilirsiniz.

Hata Ayıklama

"Döngüler ve Kararlar" adlı Bölüm 3'te döngülerin nasıl çalıştığını aydınlatmak için hata ayıklayıcı (debugger) kullanımını tavsiye ediyoruz. C++Builder'da bunun nasıl yapılacağını şimdi burada ele alacağız. Aynı adımları, programınızın hatalı çalıştığı durumlarda da bu kez hataları ayıklamak için kullanabilirsiniz. Burada sadece tek dosyalı programları ele alacağız. Ancak, belli bazı değişikliklerin dışında buradaki yaklaşım daha büyük ve çok dosyalı programlarda da geçerlidir.

Önce programınızı normal bir şekilde inşa edin. Ortaya çıkan derleyici ve bağlayıcı hatalarını giderin. Programın kaynak kodunun Edit penceresinde gösterildiğinden emin olun.

Adım Adım İlerleme

Hata ayıklayıcıyı başlatmak için sadece F8'e basmanız yeterlidir. Bu sayede, program yeniden derlenecektir ve programın ilk satırı – ki bu genellikle **main()** deklarasyonudur – parlak hale gelir. F8'e tekrar tekrar basmak kontrolün, programdaki ifadelerin üzerinden tek tek, sırayla ilerlemesine neden olur. Bir döngüye girdiğiniz zaman parlaklığın döngü boyunca aşağıya doğru ilerlediğini, sonra bir sonraki tekrarda yeniden döngünün başına döndüğünü göreceksiniz.

Değişkenleri İzlemek

Program boyunca adım adım ilerlerken değişken değerlerinin nasıl değiştiğini görmek için Run menüsünden Add Watch seçeneğini seçin. Watch Properties iletişim kutusu ekranda belirecektir. Bu iletişim kutusunun Expression alanına izlemek istediğiniz değişkenin ismini yazın, sonra karşılık gelen değişken tipini seçin ve OK'i tıklayın. Watch List adında bir pencere karşınıza çıkacaktır. Add Watch iletişim kutusunu tekrar tekrar kullanarak istediğiniz kadar çok sayıda değişkeni Watch listesine ekleyebilirsiniz.

Edit penceresini ve Watch listesini aynı anda her ikisini de görecek şekilde konumlandırırsanız, program içinde adım adım ilerlerken değişkenlerin değişen değerlerini de izleyebilirsiniz. Eğer değişken geçerli olduğu alanın dışında ise, örneğin tanımlandığı noktadan önce onun değerini izlemeye çalışıyorsanız, Watch listesinde değişkenin değer yerine bir hata mesajı görürsünüz.

CUBELIST programı gibi özel bir örnekte, **cube** değişkeni döngü içinde tanımlandığında izleme mekanizması **cube** değişkeninin geçerliliğini fark edemez. Bu programı, **cube** değişkeni

döngüden önce tanımlanacak şekilde yeniden yazın. Böylece, değişkenin değeri Watch List'te düzgün olarak gösterilecektir.

Fonksiyonların İçine Girmek

Programınızda fonksiyon kullanılıyorsa, F7 tuşuna basarak bu fonksiyonlara *trace into* yapabilirsiniz. (Yani fonksiyonun içine girip onun ifadelerini adım adım işletebilirsiniz.) Fonksiyon adının üzerinde iken F8 tuşuna basarsanız *step over* olur; yani fonksiyon tek bir ifade olarak değerlendirilir, içine girmeden çalıştırılır ve bir sonraki ifadeye geçilir. **cout <<** gibi kütüphane fonksiyonlarının nasıl çalıştığını izlemek için F7'ye basarsanız, kütüphane fonksiyonunun kaynak kodu üzerinden ilerleyebilirsiniz. Bu uzunca bir işlem olabilir, dolayısıyla gerçekten ilgilenizi çekmiyorsa bu yola gitmekten kaçınmanızda yarar var. Yani, söz konusu fonksiyonun nasıl çalıştığını araştırmayı isteyip istemediğinize bağlı olarak F7 ve F8 arasında mantıklı bir geçiş yapmanız gereklidir.

Durak Noktaları

Durak noktaları (breakpoint) programı istediğiniz noktada durdurmanıza imkan verir. Peki bunun yararı nedir? Run menüsünden "Run to Cursor" seçeneğini seçerek programı belli bir noktaya kadar çalıştırmayı daha önce öğrenmiştik. Bazen öyle durumlar olur ki, programı birden fazla yerde durdurmanız gerekir. Örneğin, programı belli bir **if** ifadesinden ve de ona karşılık gelen **else** ifadesinden sonra durdurmanız gerekebilir. İşte durak noktaları bu problemi çözer, çünkü ihtiyacınız olduğu kadar çok sayıda durak noktası kullanabilirsiniz. (Durak noktalarının, burada ayrıntısına girmeyeceğimiz başka gelişkin özellikleri de vardır.)

Program listenize bir durak noktası eklemek kolaydır. Program listenizi Edit penceresine yükleyin. Çalıştırılabilir program satırlarının her birinin karşısındaki sol kenar boşluğunda bir nokta göreceksiniz. Durak noktası eklemek için durak noktası eklemek istediğiniz yerdeki bu noktaya sol fare tuşu ile tıklamanız yeterlidir. Kodun sol tarafında kırmızı bir daire ortaya çıkacaktır ve program satırı parlak hale gelecektir. Artık programınızı ne zaman tam hız çalıştırırsanız her seferinde (örneğin, Run menüsünden Run'ı seçerek) program durak noktasında duracaktır. Böylece, değişkenleri inceleyebilir, kod içinde adım adım ilerleyebilir veya bir başka durak noktasına kadar programınızı çalıştırabilirsiniz.

Durak noktasını kaldırmak için durak noktası üzerinde tekrar sol fare tuşuna tıklayın. Durak noktası ortadan kalkacaktır.

Debugger'ın daha başka pek çok özelliği vardır ama burada anlattıklarımız başlangıç için size yeterli olacaktır.

CONSOLE GRAPHICS LITE

Örnek programları bazı grafik öğeleriyle renklendirebilmek iyi olur; dolayısıyla, bazı grafik tabanlı örnekleri bu kitaba dahil ettik. Standart C++ grafik spesifikasyonlarını içermez ama, grafikleri de elbette engellemez; Windows ise çeşitli grafik türlerini destekler.

Microsoft Visual C++ ve Borland C++, grafikler için farklı kütüphane fonksiyonları kullanırlar ve bunların ikisi de istediğimiz her şeyi gerçekleştirmezler. Her grafik örneğinin iki versiyonuna sahip olmayı önlemek için ve ekstra beceri elde etmek için Console Graphics Lite adında kendimize ait birtakım grafik fonksiyonları kullanırız. Bu fonksiyonlar, programınızı inşa etmek için hangi dosyaları kullandığınıza bağlı olarak Microsoft veya Borland versiyonuna dönüştürülürler: Microsoft için `MSOFTCON.H` ve `MSOFTCON.CPP`; Borland için ise `BORLACON.H` ve `BORLACON.CPP`. (Microsoft derleyicisi için kullanılan dosyaların diğer derleyicilerle çalışması da mümkündür.)

Console Graphics Lite dosyaları yayıncının Web sitesinden indirilerek temin edilebilir. Örnek programların kaynak kodlarını indirmişseniz, bu grafik dosyalarını da zaten temin etmişsiniz demektir. Örnek programları indirmediyseniz, bu işlemle ilgili uymanız gereken direktifler Giriş bölümünde anlatılmıştır. Bu dosyaların listelerini bu ekin sonunda görebilirsiniz.

Grafik rutinlerimiz konsol grafiklerini kullanır. Konsol, genellikle 80 sütun ve 25 satır olarak düzenlenen karakter modunda bir ekrandır. Bu kitaptaki grafiksel olmayan örnek programların bir çoğu metinleri konsol penceresine yazıyorlar. Bir konsol programı Windows'ta kendisine ait bir pencere içinde çalıştırılabilir veya tek başına bir MS-DOS programı olarak da çalıştırılabilir.

Konsol grafiklerinde dikdörtgenler, daireler ve diğer unsurlar piksel yerine karakterlerden (örneğin, 'X' harfinden veya küçük boyutlu bloklardan) oluşur. Sonuçlar kaba görünse de, örnek programlarda işe yararlar.

Console Graphics Lite Rutinlerini Kullanmak

Grafik unsurları kullanan bir program inşa etmek için normal "build" prosedürüne birkaç adım daha eklemelisiniz. Bu adımlar şu şekilde sıralanabilir:

- Örnek programın kaynak (`.CPP`) dosyası içine uygun başlık dosyasını (`MSOFTCON.H` veya `BORLACON.H`) dahil edin.
- İlgili kaynak dosyayı (`MSOFTCON.CPP` veya `BORLACON.CPP`) projenize ekleyin, böylece söz konusu kaynak dosya örnek programınızla bağlanabilsin.
- Derleyicinin ilgili başlık dosyasını ve kaynak dosyayı bulabileceğinden emin olun.

Başlık dosyaları, Console Graphics Lite fonksiyonlarının deklarasyonlarını içerirler. Kaynak dosyalar ise bu fonksiyonların tanımlarını içerirler. Bu nedenle, uygun kaynak dosyayı derlemeniz ve bu derleme neticesinde elde edilen `.OBJ` dosyasını programınızın kalan kısmıyla bağlamanız gerekir. Kaynak dosyayı projenize eklerseniz bu işlemler "build" işlemi sırasında otomatik olarak yapılır.

Projenize bir dosya eklemeyi öğrenmek için "Microsoft Visual C++" adlı Ek C'ye ya da "Borland C++Builder" adlı Ek D'ye bakın. Sonra bu işlemleri kaynak dosyanıza uygulayın.

Derleyicinizin başlık dosyasını bulmasını temin etmek için Directories (klasörler) seçeneğinde ilgili klasörün yolunu belirtmeniz gerekebilir. Bunun için de yine ilgili Ek bölümünü okuyun.

Sadece bu kitaptaki grafik örneklerini çalıştırmak istiyorsanız, bilmeniz gerekenler bundan ibarettir. Eğer Console Graphics Lite'ı kendi programlarınızda kullanma arzunuz varsa, o zaman aşağıdaki bölümleri okumaya devam edin.

Console Graphics Lite Fonksiyonları

Console Graphics Lite fonksiyonları 80 sütuna 25 satırlık bir konsol ekranı olduğunu varsayar. Ekranın sol üst köşesi (1, 1) noktası, sağ alt köşesi de (80, 25) noktası olarak tanımlanmıştır.

Bu fonksiyonlar, kitabımızdaki örnek programlar için özel olarak tasarlanmışlardır ve spesifik olarak esnek ya da gelişkin bir yönleri yoktur. Şayet bu fonksiyonları kendi programlarınızda kullanırsanız, tüm şekilleri 80x25'lik karakter ekranının sınırları içine çizmeye dikkat edin. Geçersiz koordinatlar verildiğinde bu fonksiyonların nasıl davranacağı tanımlanmamıştır. Fonksiyonlar, Tablo E.1'de sıralanmıştır.

TABLO E.1: Console Graphics Lite İçin Fonksiyonlar

Fonksiyon Adı	Amacı
`init_graphics()`	Grafik sistemini ilk kullanıma hazırlar
`set_color()`	Ön plan ve arka plan (fon) renklerini belirler
`set_cursor_pos()`	İmleci belli bir satır ve sütuna konumlandırır
`clear_screen()`	Tüm konsol ekranını temizler
`wait(n)`	Programı n milisaniye duraklatır
`clear_line()`	Bir satırı tümden siler
`draw_rectangle()`	Dikdörtgenin üst, sol, alt ve sağ koordinatları belirler
`draw_circle()`	Çemberin merkezini (x,y) ver yarıçapını belirler
`draw_line()`	Doğrunun uç noktaları olan (x1,y1) ve (x2,y2)'yi belirler
`draw_pyramid()`	Piramidin üst koordinatını (x,y) ve yüksekliğini belirler
`set_fill_style()`	Dolgu karakterini belirler

Herhangi bir grafik fonksiyonunu kullanmadan önce `init_graphics()` fonksiyonunu çağırmalısınız. Bu fonksiyon, dolgu karakterini belirler; ayrıca, Microsoft versiyonunda konsol grafik sisteminin diğer önemli parçalarını da ilk kullanıma hazırlar.

`set_color()` fonksiyonu bir ya da iki argüman kullanabilir. İlk argüman, ardarda görüntülenen karakterlerin ön plan (foreground) rengini; ikinci argüman ise karakterin arka plan (background) rengini belirler. Genellikle arka planı siyah olarak saklamak tercih edilir.

```
set_color(cRED);          //on plani kirmizi olarak ayarlar
set_color(cWHITE,cBLUE);  //on plan beyaz, arka plan mavi
```

Ön plan ya da arka plan için kullanılabilecek renk sabitlerinin işte bir listesi:

`set_color()` için Renk Sabitleri

`cBLACK`	`cDARK_GRAY`
`cDARK_BLUE`	`cBLUE`
`cDARK_GREEN`	`cGREEN`
`cDARK_CYAN`	`cCYAN`
`cDARK_RED`	`cRED`
`cDARK_MAGENTA`	`cMAGENTA`
`cBROWN`	`cYELLOW`
`cLIGHT_GRAY`	`cWHITE`

`draw_` ile başlayan fonksiyonlar *dolgu karakteri* denilen özel bir karakter kullanarak şekiller ve çizgiler çizer. Bu karakterin varsayılan değeri içi dolu bir bloktur, ama bu değer `set_fill_style()` fonksiyonu kullanılarak değiştirilebilir. İçi dolu bloğun yanı sıra, büyük harf olarak 'X' veya 'O' karakterlerini veya gölgeli üç blok karakterinden herhangi birini kullanabilirsiniz. Dolgu sabitlerinin listesi şu şekildedir:

`set_fill_style()` İçin Dolgu Sabitleri

`SOLID_FILL`	`LIGHT_FILL`
`X_FILL`	`MEDIUM_FILL`
`O_FILL`	`DARK_FILL`

`wait()` fonksiyonu saniye cinsinden bir argüman alır ve bu kadar bir süre bekler.

```
wait(3000);     //3 saniye bekler
```

Diğer fonksiyonlar büyük ölçüde kendi kendilerini açıklarlar. Bunların işleyişleri grafik unsurlar kullanan örneklerde görülebilir.

Console Graphics Lite Fonksiyonlarının Uygulanması

Console Graphics Lite için kullanılan bu rutinler nesne tabanlı değildir; hatta, C++ yerine C dilinde de yazılmış olabilirlerdi. Bu sebeple, doğru ve çember çizmek gibi hızlı ve basit grafik işlemlerinin peşinde değilseniz bu rutinleri incelemenizi gerektiren bir neden yok demektir. Buradaki amaç, iş görecek en az sayıda rutin geliştirmekti. Şayet merak ediyorsanız, bu ekin sonundaki kaynak dosyaları inceleyebilirsiniz.

Microsoft Derleyicileri

Microsoft derleyicilerinde birkaç yıl öncesine kadar mevcut olan kendilerine ait konsol grafik rutinleri artık mevcut değildir. Yine de, Windows'un kendisi, imleci konumlandırmak veya metnin rengini değiştirmek gibi basit konsol grafik işlemleri için bir takım rutinler sağlar. Microsoft derleyicilerinde Console Graphics Lite fonksiyonları, bu hazır Windows konsol fonksiyonlarına erişirler. (Bu çözümü önerdiği için André LaMothe'a teşekkürler.)

Konsol grafik fonksiyonlarını kullanmak için Ek C'de anlatılan "Win32 Console Application" tipinde bir proje kullanmalısınız.

Windows konsol fonksiyonları, konsol grafik sistemini ilk kullanıma hazırlamadığınız sürece çalışmayacaklardır. Bu nedenle, Microsoft derleyicisini kullanıyorsanız, `init_graphics()` fonksiyonunu çağırmak şarttır.

Borland Derleyicileri

Borland C++ hem konsol mod grafikleri için hem de piksel grafikleri için standart grafik fonksiyonlarına halen sahiptir. `BORLACON.CPP` dosyasını kullanıyorsanız, Console Graphics Lite fonksiyonları, çok yakından benzedikleri Borland konsol fonksiyonlarına dönüştürülürler.

Windows'un içine monte edilmiş konsol fonksiyonlarına erişmek için niçin Borland derleyicisini kullanamadığınızı merak edebilirsiniz. Problem, Borland C++'ta bir konsol mod programı geliştirmek için ya bir EasyWin ya da bir DOS hedefi kullanmanız gerektiğidir -ki,

bunların her ikisi de 16 bit sistemlerdir. Windows konsol fonksiyonları 32 bit sistemlerdir; bu nedenle, Borland'ın konsol modunda kullanılamazlar.

Borland C++ kullandığınız zaman I/O için `iostream` yaklaşımı (`cout <<`) farklı renkler üretmez. Bu nedenle, `HORSE.CPP` gibi bazı örnek programlar `CONIO.H` dosyasında bulunan `cputs()` ve `putch()` gibi konsol mod fonksiyonlarını kullanırlar.

Kaynak Kod Listeleri

Console Graphics Lite içinde kullanılan dört dosyanın listeleri aşağıda görülüyor. Microsoft için `MSOFTCON.H` ve `MSOFTCON.CPP`; C++ Builder için `BORLACON.H` ve `BORLACON.CPP`. Normal olarak bu dosyaların içindekilerle ilgili kaygılanmanızı gerektirecek hiçbir sebep olamaz. Bunlar sadece referans olarak burada gösterilmiştir.

MSOFTCON.H İçin Program Listesi

```
// msoftcon.h
// Lafore 'un konsol grafik fonksiyonlari icin deklarasyonlar
// Windows'un konsol fonksiyonlarini kullanir

#ifndef _INC_WCONSOLE          //bu dosyanin ayni kaynak dosya icine
#define _INC_WCONSOLE          //iki kez dahil edilmesine engel olun

#include <windows.h>           //Windows konsol fonksiyonlari icin
#include <conio.h>             //kbhit(),getche() icin
#include <math.h>              //sin,cos icin

enum fstyle { SOLID_FILL, X_FILL,         O_FILL,
              LIGHT_FILL, MEDIUM_FILL,    DARK_FILL };
enum color {
   cBLACK=0,      cDARK_BLUE=1,     cDARK_GREEN=2, cDARK_CYAN=3,
   cDARK_RED=4,   cDARK_MAGENTA=5,  cBROWN=6,      cLIGHT_GRAY=7,
   cDARK_GRAY=8,  cBLUE=9,          cGREEN=10,     cCYAN=11,
   cRED=12,       cMAGENTA=13,      cYELLOW=14,    cWHITE=15 };
//-----------------------------------------------------------------
void init_graphics();
void set_color(color fg, color bg = cBLACK);
void set_cursor_pos(int x, int y);
void clear_screen();
void wait(int milliseconds);
void clear_line();
void draw_rectangle(int left, int top, int right, int bottom);
void draw_circle(int x, int y, int rad);
void draw_line(int x1, int y1, int x2, int y2);
void draw_pyramid(int x1, int y1, int height);
void set_fill_style(fstyle);
#endif   /*_INC_WCONSOLE */
```

MSOFTCON.CPP İçin Program Listesi

```
// msoftcon.cpp
// Windows konsol fonksiyonlarina erismek icin gerekli rutinleri sunar

// derleyiCi bu dosyayi bulabilmelidir
// MCV++'ta, /Tools/Options/Directories/Include/dizin yolunu yazin

#include "msoftcon.h"
```

```
   HANDLE hConsole;      //konsol yoneticisi
   char fill_char;       //dolgu icin kullanilacak karakter
   //-------------------------------------------------------------
   void init_graphics()
      {
      COORD console_size = {80, 25};
      //i/o kanalini konsol ekranina ac
      hConsole = CreateFile("CONOUT$", GENERIC_WRITE | GENERIC_READ,
                            FILE_SHARE_READ | FILE_SHARE_WRITE,
                        0L, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0L);
      //80x25 ekran boyutuna ayarla
      SetConsoleScreenBufferSize(hConsole, console_size);
      //metni siyah uzerine beyaz olarak ayarla
      SetConsoleTextAttribute( hConsole, (WORD)((0 << 4) | 15) );

      fill_char = '\xDB';  //varsayilan dolgu karakteri, ici dolu bloktur
      clear_screen();
      }
//-------------------------------------------------------------
void set_color(color foreground, color background)
   {
   SetConsoleTextAttribute(hConsole,
                           (WORD)((background << 4) | foreground) );
   }     //setcolor()'in sonu

/* 0 Siyah          8 Koyu gri
   1 Koyu mavi      9 Mavi
   2 koyu yesil     10 Yesil
   3 Koyu turkuaz   11 Turkuaz
   4 Koyu kirmizi   12 Kirmizi
   5 Koyu bordo     13 Bordo
   6 Kahverengi     14 Sari
   7 Acik gri       15 Beyaz
*/
//-------------------------------------------------------------
void set_cursor_pos(int x, int y)
   {
   COORD cursor_pos;       //sol ust kose, baslangic
   cursor_pos.X = x - 1;   //Windows (0,0)'dan basliyor
   cursor_pos.Y = y - 1;   //biz (1,1)'de basliyoruz
   SetConsoleCursorPosition(hConsole, cursor_pos);
   }
//-------------------------------------------------------------
void clear_screen()
   {
   set_cursor_pos(1, 25);
   for(int j=0; j<25; j++)
      putch('\n');
   set_cursor_pos(1, 1);
   }
//-------------------------------------------------------------
void wait(int milliseconds)
   {
   Sleep(milliseconds);
   }
//-------------------------------------------------------------
void clear_line()    //satirin sonuna kadar 80 bosluk kullanarak
   {                 //Satiri temizle
   //.....12345678901234567890123456789012345678901234567890
```



```
   //.....0.........1.........2.........3.........4
   cputs("                                          ");
   cputs("                                          ");
   }
//-------------------------------------------------------------
void draw_rectangle(int left, int top, int right, int bottom)
   {
   char temp[80];
   int width = right - left + 1;

   for(int j=0; j<width; j++)
      temp[j] = fill_char;          //kareler katari
   temp[j] = 0;                     //null

   for(int y=top; y<=bottom; y++)   //karakter katarlari yigini
      {
      set_cursor_pos(left, y);
      cputs(temp);
      }
   }
//-------------------------------------------------------------
void draw_circle(int xC, int yC, int radius)
   {
   double theta, increment, xF, pi = 3.14159;
   int x, xN, yN;

   increment = 0.8 /static_cast<double>(radius);
   for(theta=0; theta<=pi/2; theta+=increment)     //ceyrek daire
      {
      xF = radius * cos(theta);
      xN = static_cast<int>(xF * 2 / 1);          //piksellerin karesi alinmaz
      yN = static_cast<int>(radius * sin(theta) + 0.5);
      x = xC-xN;
      while(x <= xC+xN)                  //her biri bir yarim daire icin
         {                               //iki yatay satir doldur
         set_cursor_pos(x,   yC-yN); putch(fill_char);   //en ust
         set_cursor_pos(x++, yC+yN); putch(fill_char);   //en alt
         }
      }//for'un sonu
   }
//-------------------------------------------------------------
void draw_line(int x1, int y1, int x2, int y2)
   {
   int w, z, t, w1, w2, z1, z2;
   double xDelta=x1-x2, yDelta=y1-y2, slope;
   bool isMoreHoriz;

   if( fabs(xDelta) > fabs(yDelta) )       //biraz daha yatay
      {
      isMoreHoriz = true;
      slope = yDelta / xDelta;
      w1=x1; z1=y1; w2=x2, z2=y2;          //w=x, z=y
      }
   else                                    //biraz daha dusey
      {
      isMoreHoriz = false;
      slope = xDelta / yDelta;
      w1=y1; z1=x1; w2=y2, z2=x2;          //w=y, z=x
      }
```

```
                                        //w'nun arkasindaysa,
    if(w1 > w2)
       {                                //bunu (w1, z1)
       t=w1; w1=w2; w2=t;               //bununla (w2, z2) takas yap
       t=z1; z1=z2; z2=t;
       }
    for(w=w1; w<=w2; w++)
       {
       z = static_cast<int>(z1 + slope * (w-w1));
       if( !(w==80 && z==25) )          //80,25 konumunda ekranin kaymasini onle
          {
          if(isMoreHoriz)
             set_cursor_pos(w, z);
          else
             set_cursor_pos(z, w);
          putch(fill_char);
          }
       }
    }
//---------------------------------------------------------------
void draw_pyramid(int x1, int y1, int height)
   {
   int x, y;
   for(y=y1; y<y1+height; y++)
      {
      int incr = y -y1;
      for(x=x1-incr; x<=x1+incr; x++)
         {
         set_cursor_pos(x, y);
         putch(fill_char);
         }
      }
   }
//---------------------------------------------------------------
void set_fill_style(fstyle fs)
   {
   switch(fs)
      {
      case SOLID_FILL:     fill_char = '\xDB';break;
      case DARK_FILL:      fill_char = '\xB0';break;
      case MEDIUM_FILL:    fill_char = '\xB1';break;
      case LIGHT_FILL:     fill_char = '\xB2';break;
      case X_FILL:         fill_char = 'X';   break;
      case O_FILL:         fill_char = 'O';   break;
      }
   }
//---------------------------------------------------------------
```

BORLACON.H İçin Program Listesi

```
// borlacon.h
// Console Graphics Lite fonksiyonlarinin deklarasyonları
// Borland'in konsol fonksiyonlarini kullanir
#ifndef _INC_WCONSOLE      //bu dosyanin ayni kaynak dosyasi icine
#define _INC_WCONSOLE      //iki kez dahil edilmesine engel ol

#include <windows.h>       //Sleep() icin
#include <conio.h>         //kbhit(), getche() icin
```



```
#include <math.h>                  //sin, cos icin

enum fstyle {SOLID_FILL,X_FILL,O_FILL,
             LIGHT_FILL,MEDIUM_FILL,DARK_FILL };

enum color {
   cBLACK=0,      cDARK_BLUE=1,                cDARK_GREEN=2, cDARK_CYAN=3,
   cDARK_RED=4,   cDARK_MAGENTA=5,             cBROWN=6,      cLIGHT_GRAY=7,
   cDARK_GRAY=8,  cBLUE=9,                     cGREEN=10,     cCYAN=11,
   cRED=12,       cMAGENTA=13,                 cYELLOW=14,    cWHITE=15 };
//---------------------------------------------------------------
void init_graphics();
void set_color(color fg, color bg = cBLACK);
void set_cursor_pos(int x, int y);
void clear_screen();
void wait(int milliseconds);
void clear_line();
void draw_rectangle(int left, int top, int right, int bottom);
void draw_circle(int x, int y, int rad);
void draw_line(int x1, int y1, int x2, int y2);
void draw_pyramid(int x1, int y1, int height);
void set_fill_style(fstyle fstyle);
#endif   //_INC_WCONSOLE
```

BORLACON.CPP İçin Program Listesi

```
// borlacon.cpp
// Borland konsol fonksiyonlari icin cizim rutinleri sunar
#include "borlaCon.h"

char fill_char;              //dolgu icin kullanilacak karakter
//---------------------------------------------------------------
void init_graphics()
   {
   textcolor(WHITE);         //siyah uzerine beyaz metin
   textbackground(BLACK);
   fill_char = '\xDB';       //varsayilan dolgu karakteri, ici dolu blok
   clrscr();
   }
//---------------------------------------------------------------
void set_color(color foreground, color background)
   {
   textcolor( static_cast<int>(foreground) );
   textbackground( static_cast<int>(background) );
   }
//---------------------------------------------------------------
void set_cursor_pos(int x, int y)
   {
   gotoxy(x, y);
   }
//---------------------------------------------------------------
void clear_screen()
   {
   clrscr();
   }
//---------------------------------------------------------------
void wait(int milliseconds)
   {
```

```
      Sleep(milliseconds);
      }
   //-------------------------------------------------------------
   void clear_line()       //satirin sonuna kadar 80 bosluk kullanarak
      {                    //satiri temizle
      //.....12345678901234567890123456789012345678901234567890
      //.....0........1.........2.........3.........4
      cputs("                                                  ");
      cputs("                                                  ");
      }     //clreol()'in sonu
   //-------------------------------------------------------------
   void draw_rectangle(int left, int top, int right, int bottom)
      {
      int j;
      char temp[80];
      int width = right - left + 1;

      for(j=0; j<width; j++)          //kareler karakter katari
         temp[j] = fill_char;
      temp[j] = 0;                    //null

      for(int y=top; y<=bottom; y++)  //karakter katarlari yigini
         {
         set_cursor_pos(left, y);
         cputs(temp);
         }
      }     //rectangle'in sonu
   //-------------------------------------------------------------
   void draw_circle(int xC, int yC, int radius)
      {
      double theta, increment, xF ,pi=3.14159;
      int x, xN, yN;

      increment = 0.8 / static_cast<double>(radius);
      for(theta=0; theta<=pi/2; theta+=increment)     //ceyrek daire
         {
         xF = radius * cos(theta);
         xN = static_cast<int>(xF * 2 /1);           //piksellerin karesi alinmaz
         yN = static_cast<int>(radius * sin(theta) + 0.5);
         x = xC-xN;
         while(x <= xC+xN)                    //her biri bir yarim daire icin
            {                                 //iki yatay satir doldur
            set_cursor_pos(x,   yC-yN); putch(fill_char);   //en ust
            set_cursor_pos(x++, yC+yN); putch(fill_char);   //en alt
            }
         }     //for'un sonu
      }     //circle()'in sonu
   //-------------------------------------------------------------
   void draw_line(int x1, int y1, int x2, int y2)
      {

      int w, z, t, w1, w2, z1, z2;
      double xDelta=x1-x2, yDelta=y1-y2, slope;
      bool isMoreHoriz;

      if( fabs(xDelta) > fabs(yDelta))       //biraz daha yatay
         {
         isMoreHoriz = true;
         slope = yDelta / xDelta;
```



```
         w1=x1; z1=y1; w2=x2, z2=y2;
         }                                    //w=x, z=y
      else
         {                                    //biraz daha dusey
         isMoreHoriz = false;
         slope = xDelta / yDelta;
         w1=y1; z1=x1; w2=y2, z2=x2;
         }                                    //w=y, z=x

      if(w1 > w2)
         {                                    //w'nun arkasindaysa,
         t=w1; w1=w2; w2=t;
         t=z1; z1=z2; z2=t;                   //bunu (w1,z1)
         }                                    //bununla (w2,z2) takas yap
      for(w=w1; w<=w2; w++)
         {
         z = static_cast<int>(z1 + slope * (w-w1));
         if( !(w==80 && z==25) )              //80,25 konumunda ekranin kaymasini onle
            {
            if(isMoreHoriz)
               set_cursor_pos(w, z);
            else
               set_cursor_pos(z, w);
            putch(fill_char);
            }
         }
      }
   //-------------------------------------------------------------
   void draw_pyramid(int x1, int y1, int height)
      {
      int x, y;
      for(y=y1; y<y1+height; y++)
         {
         int incr = y - y1;
         for(x=x1-incr; x<=x1+incr; x++)
            {
            set_cursor_pos(x, y);
            putch(fill_char);
            }
         }
      }
   //-------------------------------------------------------------
   void set_fill_style(fstyle fs)
      {
      switch(fs)
         {
         case SOLID_FILL:  fill_char ='\xDB'; break;
         case DARK_FILL:   fill_char ='\xB0'; break;
         case MEDIUM_FILL: fill_char ='\xB1'; break;
         case LIGHT_FILL:  fill_char ='\xB2'; break;
         case X_FILL:      fill_char ='X';    break;
         case O_FILL:      fill_char ='O';    break;
         }
      }
   //-------------------------------------------------------------
```

EK F

STL ALGORİTMALARI VE ÜYE FONKSİYONLARI

F

Bu ekte, Standart Şablon Kütüphanesi'nde (STL) mevcut olan algoritmaları ve konteyner üye fonksiyonlarını içeren tablolar yer alıyor. Bu bilgiler, Alexander Stepanov ve Ming Lee tarafından geliştirilen (1995) *Standart Şablon Kütüphanesi*'ne (The Standard Template Library) dayanır. Fakat, biz bunu büyük ölçüde sıkıştırdık ve yeniden gözden geçirdik; anlaşılmalarını hızlandırmak adına orijinal formülasyonlarında bir takım değişiklikler yapma cesaretini gösterdik.

Algoritmalar

Tablo F.1'de, STL'de mevcut olan algoritmalar yer alıyor. Bu tablodaki tanımlamalar, algoritmaların ne yaptığı konusunda hızlı ve yoğun açıklamalar sağlarlar. Ancak, bu tanımlamalar ciddi matematiksel tanımlar olarak tasarlanmamıştır. Argümanlar ve dönüş değerleri için kullanılan doğru veri tipleri de dahil olmak üzere daha fazla bilgi için, konuyla ilgili kitapları araştırabilirsiniz.

Tablonun ilk sütununda fonksiyon ismi yer alır; ikinci sütunda algoritmanın amacı açıklanır; üçüncü sütunda ise argümanlar belirtilir. Dönüş değerleri sistematik olarak belirtilmez. Bunlardan bazılarına Amaç sütununda değiniliyor; bir çoğu ise ya çok barizdir ya da algoritmayı kullanırken hayati öneme sahip değildir.

Argüman sütunundaki `first`, `last`, `first1`, `last1`, `first2`, `last2`, `first3` ve `middle`, bir konteyner içindeki belirli yerlerdeki iteratörleri simgelerler. Sayı içeren isimler (örneğin, `first1`) çok sayıda konteyneri ayırt etmek için kullanılır. `first1` ve `last1` birinci aralığı, `first2` ve `last2` ise ikinci aralığı sınırlandırır. `function`, `predicate`, `op` ve `comp` argümanları birer fonksiyon nesnesidir. `value`, `old`, `new`, `a`, `b` ve `init` argümanları bir konteyner içinde saklanan nesnelerin değerleridir. Bu değerler ya sıralanmıştır ya da < veya == operatörleri baz alınarak veya `comp` fonksiyon nesnesi kullanılarak karşılaştırılır. `n` argümanı bir tamsayıdır.

Amaç sütununda, taşınabilir iteratörler `iter`, `iter1` ve `iter2` terimleriyle belirtilir. `iter1` ve `iter2` beraber kullanıldığında, bu iteratörlerin kendi konteynerleri içinde (ya da aynı konteyner içinde muhtemelen iki farklı aralıkta) birlikte adım adım ilerleyecekleri farz edilir.

TABLO F.1: Algoritmalar

İsim	Amaç	Argümanlar
	Değişmeyen Sekans İşlemleri	
`for_each`	Her nesneye uygulanır.	`first, last, function`
`find`	value değerine eşit olan ilk nesneye bir iteratör döndürür.	`first, last, value`
`find_if`	predicate'ın true değerine sahip olduğu ilk nesneye bir iteratör döndürür.	`first, last, predicate`
`adjacent_find`	Yan yana ve eşit değere sahip nesne çiftleri: n ilkine bir iteratör döndürür.	`first, last`
`adjacent_find`	predicate'ı sağlayan yan yana nesne çiftlerinin ilkine bir iteratör döndürür.	`first, last, predicate`
`count`	value değerine sahip nesnelerin sayısını n'e ekler.	`first, last, value, n`
`count_if`	Predicate'ı sağlayan nesnelerin sayısını n'e ekler.	`first, last, predicate, n`
`mismatch`	İki aralık içinde, karşılık gelen ve birbirine eşit olmayan ilk nesne çiftini döndürür.	`first1, last1, first2`

TABLO F.1: Algoritmalar

İsim	Amaç	Argümanlar
`mismatch`	İki aralık içinde, birbirine karşılık gelen ve predicate'i sağlamayan ilk nesne çiftini döndürür.	`first1, last1, first2, predicate`
`equal`	İki aralık içinde, karşılık gelen nesnelerin tümü birbirine eşitse true döndürür.	`First1, last1, first2`
`equal`	İki aralık içinde, karşılık gelen nesnelerin tümü predicate'i sağlarsa true döndürür.	`first1, last, first2, Predicate`
`search`	İkinci aralığın birinci aralık içinde bulunup bulunmadığını kontrol eder. Eğer eşleşme söz konusu ise, eşleşmenin başlangıcını döndürür; aksi halde, last1'i döndürür.	`first1, last1, first2, last2`
`search`	İkinci aralığın birinci aralık içinde bulunup bulunmadığını kontrol eder. Eşitlik predicate'e göre belirlenir. Eğer eşleşme söz konusu ise, eşleşmenin başlangıcını döndürür; aksi halde, last1'i döndürür.	`First1, last1, first2, last2, predicate`
	Değişen Sekans İşlemler	
`copy`	1. aralıktaki nesneleri 2. aralığa kopyalar.	`First1, last1, first2`
`copy_backward`	Nesneleri 1. aralıktan 2. aralığa kopyalar. Kopyalama sırasında last2'den first2'ye doğru, nesneleri ters sırada ekler.	`First1, last1, first2`
`swap`	İki nesneyi takas eder.	`a, b`
`iter_swap`	İki iteratör tarafından işaret edilen nesneleri takas eder.	`iter1, iter2`
`swap_ranges`	İki aralık içinde karşılık gelen öğeleri takas eder.	`first1, last1, first2`
`transform`	operator'ı uygulayarak 1. aralıktaki nesneleri 2. aralıkta yeni nesnelere dönüştürür.	`first1, last1, first2, operator`
`transform`	operator'ı uygulayarak 1. ve 2. aralıktaki nesneleri birleştirip, 3. aralıkta yeni nesnelere dönüştürür.	`first1, last1, first2, first3, operator`
`replace`	old'a eşit olan tüm nesneleri new'a eşit olan nesnelerle değiştirir.	`first, last, old, new`
`replace_if`	predicate'ı sağlayan tüm nesneleri, new'a eşit olan nesnelerle değiştirir.	`first, last, predicate, new`
`replace_copy`	old'a eşit olan nesnelerin tümünü new'a eşit olan nesnelerle değiştirerek 1. aralıktan 2. aralığa kopyalar.	`first1, last1, first2, old, new`
`replace_copy_if`	Predicate'i sağlayan nesnelerin tümünü new'a eşit olan nesnelerle değiştirerek 1. aralıktan 2. aralığa kopyalar.	`first1, last1, first2, predicate, new`
`fill`	Aralık içindeki tüm nesnelere value değerini atar.	`first, last, value`
`fill_n`	first'ten first+N'e kadarki nesnelerin tümüne value değerini atar.	`first, n, value`

TABLO F.1: Algoritmalar

İsim	Amaç	Argümanlar
`generate`	Aralığı, gen fonksiyonuna yapılan peş peşe çağrılarla elde edilen değerlerle doldurur.	`first, last, gen`
`generate_n`	gen fonksiyonuna yapılan peş peşe çağrılarla üretilen değerleri kullanarak first'ten first+n'e uzanan aralığı doldurur.	`first, n, gen`
`remove`	value değerine eşit olan herhangi bir nesneyi aralıktan çıkarır.	`first, last, value`
`remove_if`	predicate'i sağlayan herhangi bir nesneyi aralıktan çıkarır.	`first, last, predicate`
`remove_copy`	value değerine eşit olanlar hariç nesneleri 1. aralıktan 2. aralığa kopyalar.	`first1, last1, first2, value`
`remove_copy_if`	pred'i sağlayanlar hariç, nesneleri 1. aralıktan 2. aralığa kopyalar.	`first1, last1, first2, pred`
`unique`	Aynı değere sahip herhangi ardışık nesne sekansından ilk nesne hariç diğerlerini yok eder.	`first, last`
`unique`	predicate'ı sağlayan herhangi ardışık nesne sekansından ilk nesne hariç diğerlerini yok eder.	`first, last, predicate`
`unique_copy`	Nesneleri 1. aralıktan 2. aralığa kopyalar. Fakat, peş peşe gelen eşit nesneler sekansından yalnızca, sekansın başında bulunan ilk nesne kopyalanır. Diğerleri kopyalanmaz.	`first1, last1, first2`
`unique_copy`	Nesneleri 1. aralıktan 2. aralığa kopyalar. Fakat, peş peşe gelen ve predicate'i sağlayan nesneler sekansından yalnızca, sekansın başında bulunan ilk nesne kopyalanır. Diğerleri kopyalanmaz.	`first1, last1, first2, predicate`
`reverse`	Aralık içindeki nesnelerin sırasını tersine çevirir.	`first,last`
`reverse_copy`	1. aralıktaki nesneleri sıralarını ters çevirerek 2. aralığa kopyalar.	`first1, last1, first2`
`rotate`	Bir nesne sekansını middle iteratörü etrafında döndürür.	`first, last, middle`
`rotate_copy`	Bir nesne sekansını middle1 iteratörü etrafında döndürerek 1. aralıktan 2. aralığa kopyalar.	`first1, middle1, last1, first2`
`random_shuffle`	Aralık içindeki nesneleri rasgele karıştırır.	`first, last`
`random_shuffle`	Rasgele sayı üreten rand fonksiyonunu kullanarak aralık içindeki nesneleri rasgele karıştırır.	`first, last, rand`
`partition`	predicate'ı sağlayan tüm nesneleri, predicate'ı sağlamayanların önüne gelecek şekilde taşır.	`first, last, predicate`
`stable_partition`	predicate'ı sağlayan tüm nesneleri, predicate'ı sağlamayanların önüne gelecek şekilde taşır. Bu işlem sırasında her iki grup içindeki nispi sıralamayı korur.	`first, last, predicate`
	Sıralama ve İlgili İşlemler	
`sort`	Aralık içindeki nesneleri sıralar.	`first, last`

TABLO F.1: Algoritmalar

İsim	Amaç	Argümanlar
`sort`	Karşılaştırma fonksiyonu olarak comp'u kullanarak aralık içindeki öğeleri sıralar.	`first, last, comp`
`stable_sort`	Eşit öğelerin sırasını muhafaza ederek aralık içindeki nesneleri sıralar.	`first, last`
`stable_sort`	Eşit öğelerin sırasını muhafaza ederek ve karşılaştırma fonksiyonu olarak comp'u kullanarak aralık içindeki nesneleri sıralar.	`first, last, comp`
`partial_sort`	Aralık içindeki nesnelerin tümünü sıralar; first ve middle arasına sığacak kadar çok sayıda sıralı değeri bu aralığa yerleştirir. middle ve last arasındaki nesnelerin sırası belirsizdir.	`first, middle, last`
`partial_sort`	Aralık içindeki nesnelerin tümünü sıralar; first ve middle arasına sığacak kadar çok sayıda sıralı değeri bu aralığa yerleştirir. middle ve last arasındaki nesnelerin sırası belirsizdir. Sıralamayı tanımlamak için predicate'i kullanır.	`first, middle, last, predicate`
`partial_sort_copy`	partial_sort(first, middle, last) ile aynıdır; fakat, sonuçta elde edilen sekansı 2. aralığa yerleştirir.	`first1, last1, first2, last2`
`partial_sort_copy`	partial_sort(first, middle, last, predicate) ile aynıdır; fakat, sonuçta elde edilen sekansı 2. aralığa yerleştirir.	`first1, last1, first2, last2, comp`
`nth_element`	n. nesneyi, eğer tüm aralık sıralanmış olsaydı, bu nesneye karşılık gelmesi gereken konuma yerleştirir.	`first, nth, last`
`nth_element`	n. nesneyi, eğer tüm aralık sıralanmış olsaydı, bu nesneye karşılık gelmesi gereken konuma yerleştirir. Karşılaştırmalar için comp kullanılır.	`first, nth, last, comp`
`lower_bound`	Sıralamayı bozmadan value'nun eklenebileceği ilk konuma bir iteratör döndürür.	`first, last, value`
`lower_bound`	comp'u baz alan sıralamayı bozmadan value'nun eklenebileceği ilk konuma bir iteratör döndürür.	`first, last, value, comp`
`upper_bound`	Sıralamayı bozmadan value'nun eklenebileceği son konuma bir iteratör döndürür.	`first, last, value`
`upper_bound`	comp'u baz alan sıralamayı bozmadan value'nun eklenebileceği son konuma bir iteratör döndürür.	`first, last, value, comp`
`equal_range`	Sıralamayı bozmadan bir value'nun eklenebileceği aralığın alt ve üst sınırlarını kapsayan bir çift değer döndürür.	`first, last, value`
`equal_range`	comp'u baz alan sıralamayı bozmadan bir value'nun eklenebileceği aralığın alt ve üst sınırlarını kapsayan bir çift değer döndürür.	`first, last, value`
`binary_search`	value aralık içindeyse true döndürür.	`first, last, value`
`binary_search`	Sıralamanın comp tarafından belirlendiği aralık içinde value mevcutsa true döndürür.	`first, last, value, comp`

TABLO F.1: Algoritmalar

İsim	Amaç	Argümanlar
merge	Sıralı 1. ve 2. aralığı birleştirerek sıralı bir 3. aralık oluşturur.	first1, last1, first2, last2, first3
merge	Sıralı 1. ve 2. aralığı birleştirerek sıralı bir 3. aralık oluşturur. Sıralama comp ile belirlenir.	first1, last1, first2, last2, first3, comp
inplace_merge	Ardışık, sıralı iki aralığı; first, middle ve middle, last, birleştirerek first, last aralığını oluşturur.	first, middle, last
inplace_merge	Ardışık, sıralı iki aralığı; first, middle ve middle, last, birleştirerek first, last aralığını oluşturur. Sıralama comp'u baz alır.	first, middle, last, comp
includes	first2, last2 aralığındaki nesnelerin tümü aynı zamanda first1, last1 aralığındaysa true döndürür.	first1, last1, first2, last2
includes	first2, last2 aralığındaki nesnelerin tümü aynı zamanda first1, last1 aralığındaysa true döndürür. Sıralama comp'u baz alır. (Sadece, kümeler ve çoklu kümeler için geçerlidir.)	first1, last1, first2, last2, comp
set_union	1. ve 2. aralıktaki elemanların sıralı bileşimini oluşturur. (Kümeler ve çoklu kümeler için geçerlidir.)	first1, last1, first2, last2, first3
set_union	1. ve 2. aralıktaki elemanların sıralı bileşimini oluşturur. Sıralama comp'u baz alır. (Kümeler ve çoklu kümeler için geçerlidir.)	first1, last1, first2, last2, first3, comp
set_interaction	1. ve 2. aralıktaki elemanların sıralı kesişimini oluşturur. (Kümeler ve çoklu kümeler için geçerlidir.)	first1, last1, first2, last2, first3
set_interaction	1. ve 2. aralıktaki elemanların sıralı kesişimini oluşturur. Sıralama comp'u baz alır. (Kümeler ve çoklu kümeler için geçerlidir.)	first1, last1, first2, last2, first3, comp
set_difference	1. ve 2. aralıktaki elemanların sıralı farkını oluşturur. (Kümeler ve çoklu kümeler için geçerlidir.)	first1, last1, first2, last2, first3
set_difference	1. ve 2. aralıktaki elemanların sıralı farkını oluşturur. Sıralama comp'u baz alır. (Kümeler ve çoklu kümeler için geçerlidir.)	first1, last1, first2, last2, first3, comp
set_symmetric_difference	1. ve 2. aralıktaki elemanların sıralı simetrik farkını oluşturur. (Kümeler ve çoklu kümeler için geçerlidir.)	first1, last1, first2, last2, first3
set_symmetric_difference	1. ve 2. aralıktaki elemanların sıralı simetrik farkını oluşturur. Sıralama comp'u baz alır. (Kümeler ve çoklu kümeler için geçerlidir.)	first1, last1, first2, last2, first3, comp
push_heap	Değerleri last-1'den itibaren, ortaya çıkan heap'te first, last arasına yerleştirir.	first, last
push_heap	Değerleri last-1'den itibaren, ortaya çıkan heap'te first, last arasına yerleştirir. Sıralama comp ile belirlenir.	first, last, comp

TABLO F.1: Algoritmalar

İsim	Amaç	Argümanlar
pop_heap	first ve last-1'deki değerleri takas eder; first, last-1 aralığını katmana çevirir.	first, last
pop_heap	first ve last-1'deki değerleri takas eder; first, last-1 aralığını katmana çevirir. Sıralama comp ile belirlenir.	first, last, comp
make_heap	first, last aralığından bir katman oluşturur.	first, last
make_heap	comp ile belirlenen sıralamaya dayanarak first, last aralığından bir katman oluşturur.	first, last, comp
sort_heap	first, last ile ifade edilen katmandaki elemanları sıralar.	first, last
sort_heap	comp ile belirlenen sıralamaya dayanarak first, last ile ifade edilen katmandaki elemanları sıralar.	first, last, comp
min	İki nesneden küçük olanı döndürür.	a, b
min	İki nesneden küçük olanı döndürür. Sıralama comp ile belirlenir.	a, b, comp
max	İki nesneden büyük olanı döndürür.	a, b
max	İki nesneden büyük olanı döndürür. Sıralama comp ile belirlenir.	a, b, comp
max_element	Aralık içindeki en büyük elemana bir iteratör döndürür.	first, last
max_element	comp ile belirlenen bir sıralama yardımıyla aralık içindeki en büyük elemana bir iteratör döndürür.	first, last, comp
min_element	Aralık içindeki en küçük elemana bir iteratör döndürür.	first, last
min_element	comp ile belirlenen bir sıralama yardımıyla aralık içindeki en küçük elemana bir iteratör döndürür.	first, last, comp
lexicographical_compare	1. aralıktaki sekans, 2. aralıktaki sekanstan alfabetik olarak önce geliyorsa true döndürür.	first1, last1, first2, last2
lexicographical_compare	comp ile belirlenen sıralamayı baz alarak, 1. aralıktaki sekans, 2. aralıktaki sekanstan alfabetik olarak önce geliyorsa true döndürür.	first1, last1, first2, last2, comp
next_permutation	Aralık içindeki sekans üzerinde bir permütasyon gerçekleştirir.	first, last
next_permutation	Aralık içindeki sekans üzerinde bir permütasyon gerçekleştirir. Sıralama comp ile belirlenir.	first, last, comp
prev_permutation	Aralık içindeki sekans üzerinde bir ters permütasyon gerçekleştirir.	first, last
prev_permutation	Aralık içindeki sekans üzerinde bir ters permütasyon gerçekleştirir. Sıralama comp ile belirlenir.	first, last, comp
	Genelleştirilmiş Nümerik İşlemler	
accumulate	Aralık içindeki nesnelerin her birine arka arkaya init= init + *iter işlemini uygular.	first, last, init

TABLO F.1: Algoritmalar

İsim	Amaç	Argümanlar
accumulate	Aralık içindeki nesnelerin her birine arka arkaya init= op(init, *iter)işlemini uygular.	first, last, init, op
inner_product	1. ve 2. aralıktaki karşılıklı değerlere arka arkaya init= init + (*iter1)*(*iter2) işlemini uygular.	first1, last1, first2, init
inner_product	1. ve 2. aralıktaki karşılıklı değerlere arka arkaya init= op1(init, op2(*iter1, *iter2)) işlemini uygular.	first1, last1, first2, init, op1, op2
partial_sum	1.aralığın başından başlayarak mevcut iteratöre kadar değerleri toplar; toplamları 2. aralıkta karşılık gelen iteratöre yerleştirir.*iter2= sum (*first1, *(first1+1), *(first1+2), .. *iter1)	first1, last1, first2
partial_sum	1.aralıktaki first1 ve mevcut iteratör arasındaki nesnelere peş peşe op işlemini uygular; sonuçları 2. aralıkta karşılık gelen iteratöre yerleştirir.answer= *first;for (iter=first+1; iter!=iter1; iter++) op(answer, *iter);*iter2= answer;	first1, last1, first2, op
adjacent_difference	1.aralıktaki yan yana nesnelerin farkını alır; farkları 2. aralığa yerleştirir.*iter2= * (iter1+1) - *iter1;	first1, last1, first2
adjacent_difference	1.aralıktaki yan yana nesnelere peş peşe op işlemini uygular; sonuçları 2. Aralığa yerleştirir.*iter2= op (* (iter1+1), *iter1);	first1, last1, first2, op

Üye Fonksiyonlar

Farklı konteynerlerde aynı işlevlere sahip üye fonksiyonlar için aynı isimler kullanılmıştır. Bununla birlikte, mevcut üye fonksiyonların tümünü birden içeren bir konteyner sınıfı bulunmaz. Tablo F.2 ile amaçlanan, her konteyner için hangi üye fonksiyonların mevcut olduğunu göstermektir. İki sebepten ötürü fonksiyonlar için açıklama yapılmamıştır: Fonksiyonların işlevleri, ya az-çok isimlerinden bellidir ya da zaten kitaptaki metinlerde açıklanmıştır.

TABLO F.2: Üye Fonksiyonlar

	Vektör	Liste	Çift Uçlu Kuyruk	Küme	Çoklu küme	eşleme	Çoklu eşleme	Yığın	Kuyruk	Öncelik Kuyruğu
operator==	x	x	x	x	x	x	x	x	x	
operator!=	x	x	x	x	x	x	x	x	x	
operator<	x	x	x	x	x	x	x	x	x	
operator>	x	x	x	x	x	x	x	x	x	
operator<=	x	x	x	x	x	x	x	x	x	
operator>=	x	x	x	x	x	x	x	x	x	
operator=	x	x	x							

TABLO F.2: Üye Fonksiyonlar

	Vektör	Liste	Çift Uçlu Kuyruk	Küme	Çoklu küme	eşleme	Çoklu eşleme	Yığın	Kuyruk	Öncelik Kuyruğu
operator[]	x		x							
operator*		x	x			x				
operator->		x	x							
operator()				x	x	x				
operator +			x							
operator -			x							
operator++		x	x							
operator--		x	x							
operator +=			x							
operator -=			x							
begin	x	x	x	x	x	x	x			
end	x	x	x	x	x	x	x			
rbegin	x	x	x	x	x	x	x			
rend	x	x	x	x	x	x	x			
empty	x	x	x	x	x	x	x	x	x	x
size	x	x	x	x	x	x	x	x	x	x
ma_size	x	x	x	x	x	x	x			
front	x	x	x						x	
back	x	x	x						x	
push_front		x	x							
push_back	x	x	x							
pop_front		x	x							
pop_back	x	x	x							
swap	x	x	x	x	x	x	x			
insert	x	x	x	x	x	x	x			
erase	x	x	x	x	x	x	x			
find				x	x	x	x			
count				x	x	x	x			
lower_bound				x	x	x	x			
upper_bound				x	x	x	x			
equal_range				x	x	x	x			
top								x		x
push								x	x	x
pop								x	x	x
capacity	x									
reserve	x									

TABLO F.2: Üye Fonksiyonlar

	Vektör	Liste	Çift Uçlu Kuyruk	Küme	Çoklu küme	eşleme	Çoklu eşleme	Yığın	Kuyruk	Öncelik Kuyruğu
splice		x								
remove		x								
unique		x								
merge		x								
reverse		x								
sort		x								

İteratörler

Tablo F.3, her algoritma için gerekli olan iteratör tiplerini listeler.

TABLO F.3: Algoritmalar İçin Gerekli Olan İteratör Tipleri

	Giriş	Çıkış	İleri	İki Yönlü	Rasgele Erişim
for_each	X				
find	X				
find_if	X				
adjacent_find	X				
count	X				
count_if	X				
mismatch	X				
equal	X				
search			X		
copy	X	X			
copy_backward	X	X			
iter_swap			X		
swap_ranges			X		
transform	X	X			
replace			X		
replace_if			X		
replace_copy	X	X			
fill			X		
fill_n		X			
generate			X		
generate_n		X			
remove			X		

TABLO F.3: Algoritmalar İçin Gerekli Olan İteratör Tipleri

	Giriş	Çıkış	İleri	İki Yönlü	Rasgele Erişim
remove_if			X		
remove_copy	X	X			
remove_copy_if	X	X			
unique			X		
unique_copy	X	X			
reverse				X	
reverse_copy		X			
rotate			X		
rotate_copy		X	X		
random_shuffle					X
partition				X	
stable_partition				X	
sort					X
stable_sort					X
partial_sort					X
partial_sort_copy					X
nth_element					X
lower_bound			X		
upper_bound			X		
equal_range			X		
binary_search			X		
merge	X	X			
inplace_merge				X	
includes	X				
set_union	X	X			
set_intersection	X	X			
set_difference	X	X			
set_symmetric_difference	X				
push_heap					X
pop_heap					X
make_heap					X
sort_heap					X
max_element	X				
min_element	X				
lexicographical_comparison	X				

TABLO F.3: Algoritmalar İçin Gerekli Olan İteratör Tipleri

	Giriş	Çıkış	İleri	İki Yönlü	Rasgele Erişim
net_permutation				X	
prev_permutation				X	
accumulate	X				
inner_product	X				
partial_sum	X	X			
adjacent_ difference	X	X			

G

SORULARIN VE ALIŞTIRMALARIN YANITLARI

Bölüm 1

Soruların Yanıtları

1. prosedürel, nesne yönelimli
2. b
3. veriler, işlem yapan
4. a
5. veri gizliliği (data hiding)
6. a, d
7. nesneler
8. yanlış; programların organizasyonuyla ilgili prensipler farklıdır
9. encapsulation
10. d
11. yanlış; kod satırlarının birçoğu C ve C++'ta aynısıdır
12. çok biçimlilik
13. d
14. b
15. b, d

Bölüm 2

Soruların Yanıtları

1. b, c
2. parantez
3. küme parantezleri {}
4. Program başladığında çalıştırılan ilk fonksiyon `main()`'dir.
5. İfade
6.
```
// bu bir açıklamadır
/* bu bir açıklamadır */
```
7. a, d
8.
   a. 4
   b. b. 10
   c. 4
   d. 4
9. yanlış
10.
   a. tamsayı sabit
   b. karakter sabiti
   c. kayan noktalı sabit
   d. değişken ismi veya tanımlayıcı (identifier)
   e. fonksiyon ismi
11.

   a. `cout << 'X';`
   b. `cout << "Jim";`
   c. `cout << 509;`
12. yanlış; ifade çalıştırılan kadar eşit değillerdir.
13. `cout << setw(10) << george;`
14. `IOSTREAM`
15. `cin >> temp;`
16. `IOMANIP`
17. karakter kataı sabitleri, önişlemci direktifleri
18. doğru
19. 2
20. atama (=) ve aritmetik (örneğin, + veya -)
21.
```
temp += 23;
temp= temp + 23;
```
22. 1
23. 2020
24. Kütüphane fonksiyonları, aşırı yüklenmiş operatörler ve nesneler için gerekli deklarasyonları ve diğer verileri sağlamak için.
25.
26. kütüphane

Alıştırmaların Çözümleri

1.
```
// ex2_1.cpp
// galonu fit kube cevirir
#include <iostream>
using namespace std;

int main()
   {
   float gallons,cufeet;

   cout << "\nEnter quantity in gallons: ";
   cin >> gallons;
   cufeet = gallons / 7.481;
   cout << "Equivalent in cublic feet is " << cufeet << endl;
   return 0;
   }
```
2.
```
// ex2_2.cpp
// tablo uretir
#include <iostream>
#include <iomanip>
using namespace std;

int main()
```

```
    {
    cout << 1990 << setw(8) << 135 << endl
         << 1991 << setw(8) << 7290 << endl
         << 1992 << setw(8) << 11300 << endl
         << 1993 << setw(8) << 16200 << endl;
    return 0;
    }
```

3.

```
// ex2_3.cpp
// aritmetik atama ve eksiltme operatorleriyle ilgili uygulamalar
#include <iostream>
using namespace std;

int main()
   {
   int var = 10;
   cout << var << endl;        //var, 10
   var *= 2;                   //var, 20 olur
   cout << var-- << endl;      //var'i gosterir,sonra bir kez azaltir
   cout << var << endl;        //var, 19
   return 0;
   }
```

Bölüm 3

Soruların Yanıtları

1. b, c
2. **george !=sally**
3. -1 doğru; sadece 0 hatalı.
4. İlk değer atama deyimi, döngü değişkenine ilk değeri atar; koşul sınama deyimi, döngü değişkenini test eder; artırım deyimi döngü değişkenini değiştirir.
5. c, d
6. doğru
7.

```
for(int j=100; j<=110; j++)
cout << endl << j;
```

8. küme parantezleri
9. c
10.

```
int j = 100;
while( j <= 110 )
cout << endl << j++;
```

11. yanlış
12. en azından bir kere
13.


```
int j = 100;
do
   cout << endl << j++;
while(j <= 110);
```

14.

```
if(age > 21)
   cout << "Yes";
```

15. d
16.

```
if(age > 21)
   cout << "Yes";
else
   cout << "No";
```

17. a, c
18. '\r'
19. bir önceki, küme parantezleri
20. yeniden biçimlendirilmesi
21.

```
switch(ch)
   {
case 'y':
   cout << "Yes";
   break;
case 'n':
   cout << "No";
   break;
default:
   cout << "Unknown response";
}
```

22. **ticket = (speed > 55) ? 1 : 0;**
23. d
24. **limit == 55 && speed > 55**
25. birli, aritmetik, ilişkisel, mantıksal, koşul, atama
26. d
27. döngünün başına
28. b

Alıştırmaların Çözümleri

1.

```
// ex3_1.cpp
// bir sayinin katlarini gosterir
#include <iostream>
#include <iomanip>                    //setw() icin
using namespace std;
```

```
int main()
   {
   unsigned long n;                        //sayi

   cout << "\nEnter a number: ";
   cin >> n;                               //sayiyi al
   for(int j=1; j<=200; j++)               //1'den 200'ye kadar tekrarla
      {
      cout << setw(5) << j*n << " ";       //n'in katini yazdir
      if( j%10 == 0)                       //her 10 sayida bir,
         cout << endl;                     //yeni satirdan basla
      }
   return 0;
   }
```

2.

```
// ex3_2.cpp
// fahrenheit'i santigrada cevirir ya da
// santigradi fahrenheit'a cevirir
#include <iostream>
using namespace std;

int main()
   {
   int response;
   double temper;

   cout << "\nType 1 to convert fahrenheit to celsius,"
        << "\n     2 to convert celsius to fahrenheit: ";
   cin >> response;
   if( response == 1 )
      {
      cout << "Enter temperature in fahrenheit: ";
      cin >> temper;
      cout << "In celsius that's " << 5.0/9.0 * (temper-32.0);
      }
   else
      {
      cout << "Enter temperature in celsius: ";
      cin >> temper;
      cout << "In fahrenheit that's " << 9.0/5.0 * temper + 32.0;
      }
   cout << endl;
   return 0;
   }
```

3.

```
// ex3_3.cpp
// rakamlardan bir sayi olusturur
#include <iostream>
using namespace std;
#include <conio.h>                         //getche() icin

int main()
   {
```



```
   char ch;
   unsigned long total = 0;                //Sayiyi bu tutar

   cout << "\nEnter a number: ";
   while( (ch=getche()) != '\r' )          //Enter'la cik
      total = total*10 + ch - '0';         //rakamı total*10'a ekle
   cout << "\nNumber is: " << total << endl;
   return 0;
   }
```

4.

```
// ex3_4.cpp
// dort fonksiyonlu bir hesap makinesini modeller
#include <iostream>
using namespace std;

int main()
   {
   double n1, n2, ans;
   char oper, ch;

   do {
      cout << "\nEnter first number, operator, second number: ";
      cin >> n1 >> oper >> n2;
      switch(oper)
         {
         case '+': ans = n1 + n2; break;
         case '-': ans = n1 - n2; break;
         case '*': ans = n1 * n2; break;
         case '/': ans = n1 / n2; break;
         default: ans = 0;
         }
      cout << "Answer = " << ans;
      cout << "\nDo another (Enter 'y' or 'n')? ";
      cin >> ch;
      } while( ch != 'n' );
   return 0;
   }
```

Bölüm 4

Soruların Yanıtları

1. b, d
2. doğru
3. noktalı virgül
4.

```
struct time
   {
   int hrs;
   int mins;
   int secs;
   };
```

5. yanlış; sadece değişken tanımı ile bellekten yer ayrılır.
6. c
7. `time2.hrs = 11;`
8. 16 bit sistemlerde, 18 (3 yapı çarpı 3 tamsayı çarpı 2 byte) ya da 32 bit sistemlerde, 36
9. `time time1 = { 11, 10, 59 };`
10. doğru
11. `temp = fido.dogs.paw;`
12. c
13. `enum players { B1, B2, SS, B3, RF, CF, LF, P, C };`
14.

```
players joe, tom;
joe = LF;
tom = P;
```

15.
    a. geçersiz
    b. geçerli
    c. geçersi
    d. geçerli

16. 0, 1, 2
17. `enum speeds { obsolete=78, single=45, album=33 };`
18. çünkü yanlış (false) 0 ile simgelenmelidir (no (hayır) yanlışa karşılık gelir)

Alıştırma Çözümleri

1.

```
// ex4_1.cpp
// telefon numarasini saklamak icin yapi kullanir
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct phone
   {
   int area;                  //alan kodu (3 basamak)
   int exchange;              //santral kodu (3 basamak)
   int number;                //sayi (4 basamak)
   };
////////////////////////////////////////////////////////////////
int main()
   {
   phone ph1 = { 212, 767, 8900 };  //telefon numarasini ilk kullanima hazirla
   phone ph2;                       //telefon numarasini tanimla
                                    //tel. numarasini kullanicidan al
   cout << "\nEnter your area code, exchange, and number";
   cout << "\n(Don 't use leading zeros): ";
   cin >> ph2.area >> ph2.exchange >> ph2.number;

   cout << "\nMy number is "              //sayilari goster
        << '(' << ph1.area << ")"
        << ph1.exchange << '-' << ph1.number;
```



```
   cout << "\nYour number is "
        << '(' << ph2.area << ")"
        << ph2.exchange << '-' << ph2.number << endl;
   return 0;
   }
```

2.

```
// ex4_2.cpp
// yapi, duzlem uzerindeki noktalari modeller
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct point
   {
   int xCo;              //X koordinati
   int yCo;              //Y koordinati
   };
////////////////////////////////////////////////////////////////
int main()
   {
   point p1, p2, p3;                        //3 nokta tanimla

   cout << "\nEnter coordinates for p1: ";
   cin >> p1.xCo >> p1.yCo;                 //kullanicidan
   cout << "Enter coordinates for p2: ";    //2 nokta al
   cin >> p2.xCo >> p2.yCo;

   p3.xCo = p1.xCo + p2.xCo;                //p1 ve p2'nin
   p3.yCo = p1.yCo + p2.yCo;                //toplamini bul

   cout << "Coordinates of p1+p2 are: "     //toplami ekranda goster
        << p3.xCo << "," << p3.yCo << endl;
   return 0;
   }
```

3.

```
// ex4_3.cpp
// odanin hacmini modellemek icin yapi kullanir
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct Distance
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////////
struct Volume
   {
   Distance length;
   Distance width;
   Distance height;
   };
////////////////////////////////////////////////////////////////
int main()
```

```
    {
    float l, w, h;
    volume room1 = { { 16, 3.5 }, {12, 6.25 }, {8, 1.75 } };

    l = room1.length.feet   + room1.length.inches / 12.0;
    w = room1.width.feet    + room1.width.inches  / 12.0;
    h = room1.height.feet   + room1.height.inches / 12.0;

    cout << "Volume = " << l*w*h << " cubic feet\n";
    return 0;
    }
```

Bölüm 5

Soruların Yanıtları

1. d (b için de yarım puan)
2. tanımı
3.

```
void foo()
   {
   cout << "foo";
   }
```

4. deklarasyon, prototip
5. gövdesini
6. çağrısı
7. deklaratör
8. c
9. yanlış
10. Argümanların amacını netleştirmek için
11. a, b, c
12. Boş parantezler, fonksiyonun argüman almadığını gösterir
13. bir tane
14. doğru
15. deklarasyon ve deklaratörün en başında
16. `void`
17.

```
main()
   {
   int times2(int);          //prototip
   int alpha = times2(37);   //fonksiyon cagrisi
   }
```

18. d
19. orijinal argümanı değiştirmek için (ya da büyük bir argümanı kopyalamaya gerek kalmaması için)
20. a, c
21.


```
int bar(char);
int bar(char, char);
```

22. hızlı, çok
23. `inline float foobar(float fvar)`
24. a, b
25. `char blyth(int,float=3.14159);`
26. erişilebilirliği, yaşam süresi
27. değişken tanımının ardından tanımlanan fonksiyonlar
28. yerel değişkenin içinde tanımlı olduğu fonksiyon
29. b, d
30. eşittir işaretinin sol tarafına

Alıştırmaların Çözümleri

1.

```
// ex5_1.cpp
// fonksiyon, dairenin alanini hesaplar
#include <iostream>
using namespace std;
float circarea(float radius);

int main()
   {
   double rad;
   cout << "\nEnter radius of circle: ";
   cin >> rad;
   cout << "Area is " << circarea(rad) << endl;
   return 0;
   }
//-------------------------------------------------------------
float circarea(float r)
   {
   const float PI=3.14159F;
   return r * r * PI;
   }
```

2.

```
// ex5_2.cpp
// fonksiyon, bir sayinin ussunu bulur
#include <iostream>
using namespace std;
double power(double n, int p=2);   //p'nin varsayilan degeri 2'dir

int main()
   {
   double number, answer;
   int pow;
   char yeserno;

   cout << "\nEnter number: ";      //sayiyi al
   cin >> number;
   cout << "Want to enter a power (y/n)? ";
```

```
    cin >> yeserno;
    if( yeserno == 'y' )       //kullanici 2'den farkli bir us istiyor mu?
       {
       cout << "Enter power: ";
       cin >> pow;
       answer = power(number, pow);     //sayinin, "pow" ussunu bul
       }
    else
       answer = power(number);          //sayini karesini al
    cout << "Answer is " << answer << endl;
    return 0;
    }
//-------------------------------------------------------------
// power()
// n sayisinin p ussunu dondurur
double power(double n, int p)
   {
   double result = 1.0;       //1 ile basla
   for(int j=0; j<p; j++)     //p kere
      result *= n;            //n ile carp
   return result;
   }
```

3.

```
// ex5_3.cpp
// fonksiyon, iki sayidan kucuk olanina 0 degerini verir
#include <iostream>
using namespace std;

int main()
   {
   void zeroSmaller(int&, int&);
   int a=4, b=7, c=11, d=9;

   zeroSmaller(a, b);
   zeroSmaller(c, d);

   cout << "\na=" << a << " b=" << b
        << " c=" << c << " d=" << d;
   return 0;
   }
//-------------------------------------------------------------
// zeroSmaller()
// iki sayidan kucuk olanina 0 degerini verir
void zeroSmaller(int& first, int& second)
   {
   if( first < second )
      first = 0;
   else
      second = 0;
   }
```

4.

```
// ex5_4.cpp
// fonksiyon, iki mesafeden buyuk olanini dondurur
```



```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct Distance          //Ingiliz olcu sistemine gore uzaklik
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////////
Distance bigengl(Distance, Distance); //deklarasyonlar
void engldisp(Distance);

int main()
   {
   Distance d1, d2, d3;
                                 //uc uzaklik tanimla
                                 //d1 uzakligini kullanicidan al
   cout << "\nEnter feet: "; cin >> d1.feet;
   cout << "Enter inches: "; cin >> d1.inches;
                                 //d2 uzakligini kullanicidan al
   cout << "\nEnter feet: "; cin >> d2.feet;
   cout << "Enter inches: "; cin >> d2.inches;

   d3 = bigengl(d1, d2);         //d3, d1 ve d2'den buyuk olanidir
                                 //butun mesafeleri goster
   cout << "\nd1="; engldisp(d1);
   cout << "\nd2="; engldisp(d2);
   cout << "\nlargest is "; engldisp(d3); cout << endl;
   return 0;
   }
//-------------------------------------------------------------
// bigengl()
// Distance tipinde iki yapiyi karsilastirir, buyuk olani dondurur
Distance bigengl(Distance dd1, Distance dd2)
   {
   if(dd1.feet > dd2.feet)       //fit degerleri farkliysa,
      return dd1;                //buyuk fit degerine sahip olani dondur
   if(dd1.feet < dd2.feet)
      return dd2;
   if(dd1.inches > dd2.inches)   //inc degerleri farkliysa,
      return dd1;                //buyuk inc degerine sahip olani dondur,
   else                          //eger esitse, dd2'yi dondur
      return dd2;
   }
//-------------------------------------------------------------
// engldisp()
// Distance tipindeki yapiyi fit ve inc cinsinden goster
void engldisp(Distance dd)
   {
   cout << dd.feet << "\'-" << dd.inches << "\"";
   }
```

Bölüm 6

Soruların Yanıtları

1. Sınıf tanımı, bu sınıfa ait nesnelerin oluşturuldukları zaman nasıl görüneceklerini ifade eder.

2. sınıf, nesne
3. c
4.

```
class leverage
   {
   private:
      int crowbar;
   public:
      void pry();
   };
```

5. yanlış; hem veriler hem de fonksiyonlar private veya public olabilir
6. `leverage lever1;`
7. d
8. `lever1.pry();`
9. `inline`'dır (ayrıca `private`'dır)
10.

```
int getcrow()
   { return crowbar; }
```

11. tanımlandığında
12. üyesi olduğu sınıfın
13.

```
leverage()
   { crowbar = 0; }
```

14. doğru
15. a
16. `int getcrow();`
17.

```
int leverage::getcrow()
   { return crowbar; }
```

18. üye fonksiyonların ve verilerin yapı içinde varsayılan durumda public, sınıf içinde private olmalarıdır
19. üç, bir
20. nesnenin üye fonksiyonlarından birini çağırmak
21. b, c, d
22. yanlış; deneme-yanılma yöntemi gerekli olabilir
23. d
24. doğru
25. `void aFunc(const float jerry)const;`

Alıştırmaların Çözümleri

1.


```
// ex6_1.cpp
// bir tamsayi veri tipini modellemek icin bir sinif kullanir
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Int                                   //(int ile ayni degil)
   {
   private:
      int i;
   public:
      Int()
         { i = 0; }                         //bir Int olustur
      Int(int ii)
         { i = ii; }                        //bir Int tanimla ve baslangic degeri ata
      void add(Int i2, Int i3)              //iki Int'i topla
         { i = i2.i + i3.i; }
      void display()                        //Int'i ekranda goster
         { cout << i; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Int Int1(7);                             //bir Int tanimla, baslangic degeri ata
   Int Int2(11);                            //bir Int tanimla, baslangic degeri ata
   Int Int3;                                //bir Int tanimla

   Int3.add(Int1, Int2);                    //iki Int'i topla
   cout << "\nInt3="; Int3.display(); //sonucu ekranda goster
   cout << endl;
   return 0;
   }
```

2.

```
// ex6_2.cpp
// para odeme gisesini modellemek icin sinif kullanir
#include <iostream>
using namespace std;
#include <conio.h>
const char ESC = 27;                        //ESC tusunun ASCII kodu
const double TOLL = 0.5;                    //kopruden gecis ucreti 50 cent
////////////////////////////////////////////////////////////////
class tollBooth
   {
   private:
      unsigned int totalCars;               //bugun kopruden gecen toplam araba sayisi
      double totalCash;                     //bugun toplanan toplam para miktari
   public:                                  //kurucu fonk.
      tollBooth() : totalCars(0), totalCash(0.0)
         { }
      void payingCar()                      //odeme yapan araba
         { totalCars++; totalCash += TOLL; }
      void nopayCar()                       //odeme yapmayan araba
         { totalCars++; }
      void display() const                  //toplamlari ekranda goster
         { cout << "\nCars=" << totalCars
                << ", cash=" << totalCash
                << endl;}
```

```
   };
////////////////////////////////////////////////////////////////
int main()
   {
   tollBooth booth1;                  //para odeme gisesi tanimla
   char ch;

   cout << "\nPress 0 for each non-paying car,"
        << "\n 1 for each paying car,"
        << "\n Esc to exit the program.\n";

   do {
      ch = getche();                  //bir karakter al
      if( ch == '0' )                 //eger 0 ise, araba odeme yapmadi demektir
         booth1.nopayCar();
      if( ch == '1')                  //1 ise, araba odeme yapmistir
         booth1.payingCar();
      } while( ch != ESC );           //ESC ile donguden cik
   booth1.display();                 //toplamlari ekranda goster
   return 0;
   }
```

3.

```
// ex6_3.cpp
// saat veri tipini modellemek icin sinif kullanir
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class time
   {
   private:
      int hrs, mins, secs;
   public:
      time() : hrs(0), mins(0), secs(0)    //argumansiz kurucu fonk.
         { }
                                           //3-argumanli kurucu fonk.
      time(int h, int m, int s) : hrs(h), mins(m), secs(s)
         { }
      void display() const                 //format 11:59:59
         { cout << hrs << ":" << mins << ":" <<secs; }
      void add_time(time t1, time t2)      //iki saati topla
         {
         secs = t1.secs + t2.secs;         //saniyeleri topla
         if( secs >59 )                    //tasma varsa,
            { secs -= 60; mins++; }        //elde bir dakika tasi
         mins += t1.mins + t2.mins;        //dakikalari topla
         if( mins >59 )                    //tasma varsa,
            { mins -= 60; hrs++; }         //elde bir saat tasi
         hrs += t1.hrs + t2.hrs;           //saatleri topla
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   const time time1(5, 59, 59);     //iki saat tanimlar ve
   const time time2(4, 30, 30);     //ilk kullanima hazirlar
   time time3;                      //bir baska saat tanimlar
```



```
   time3.add_time(time1, time2);    //iki saati toplar
   cout << "time3="; time3.display(); //sonucu ekranda gosterir
   cout << endl;
   return 0;
   }
```

Bölüm 7

Soruların Yanıtları

1. d
2. aynı
3. `double doubleArray [100];`
4. 0, 9
5. `cout << doubleArray[j];`
6. c
7. `int coins[] = { 1, 5, 10, 25, 50, 100 };`
8. d
9. `twoD [2][4]`
10. doğru
11. `float flarr[3][3] = { {52, 27, 83}, {94, 73, 49}, {3, 6, 1} };`
12. bellek adresini
13. a, d
14. `employee` yapısından veya sınıfından 1000 elemanlık bir dizi
15. `emplist [16].salary`
16. d
17. `bird manybirds [50];`
18. yanlış
19. `manybirds[26].cheep();`
20. `char`, dizi
21. `char city[21]`(sıfır –null karakteri için ek bir byte gereklidir.)
22. `char dextrose []="C6H1206-H20 ";`
23. doğru
24. d
25. `strcpy(blank,name);`
26.

```
class dog
   {
   private:
      char breed[80];
      int age;
   };
```

27. yanlış
28. b, c
29. `int n = s1.find("cat");`
30. `s1.insert(12, "cat");`

Alıştırmaların Yanıtları

1.

```
// ex7_1.cpp
// bir C karakter katarini tersine cevirir
#include <iostream>
#include <cstring>                  //strlen() icin
using namespace std;

int main()
   {
   void reversit( char[] );         //prototip
   const int MAX = 80;              //dizi buyuklugu
   char str[MAX];                   //karakter katari

   cout << "\nEnter a string: ";    //kullanicidan karakter katarini al
   cin.get(str, MAX);
   reversit(str);                   // karakter katarini tersine cevir

   cout << "Reversed string is: "; //ekranda goster
   cout << str << endl;
   return 0;
   }
//--------------------------------------------------------------
// reversit()
// arguman olarak aktarilan karakter katarini tersine ceviren fonksiyon
void reversit( char s[] )
   {
   int len = strlen(s);             //karakter katarinin uzunlugunu bul
   for(int j=0; j <len/2; j++)      //ilk yaridaki her karakteri
      {                             //ikinci
      char temp = s[j];             //yaridakilerle
      s[j] = s[len-j-1];            //degistir
      s[len-j-1] = temp;
      }
   }
```

2.

```
// ex7_2.cpp
// employee nesnesi veri olarak bir karakter katari kullanir
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class employee
   {
   private:
      string name;
      long number;
   public:
      void getdata()                //kullanicidan veri al
         {
         cout << "\nEnter name: "; cin >> name;
         cout << "Enter number: "; cin >> number;
         }
```



```
      void putdata()
         {                          //veriyi goruntule
         cout << "\nName: " << name;
         cout << "\nNumber: " << number;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   employee emparr[100];
   int n = 0;                       //employee dizisi
   char ch;                         //kac calisan var
                                    //kullanici yaniti
   do {
      cout << "\nEnter data for employee number " << n+1;
      emparr[n++].getdata();        //kullanicidan veri al
      cout << "Enter another (y/n)? "; cin >> ch;
      } while( ch != 'n' );

   for(int j=0; j<n; j++)           //dizideki veriyi goruntule
      {
      cout << "\nEmployee number " << j+1;
      emparr[j].putdata();
      }
   cout << endl;
   return 0;
   }
```

3.

```
// ex7_3.cpp
// kullanici tarafindan girilen Distance nesneleri dizisinin ortalamasini bulur
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                  //Ingiliz olcu sistemine gore uzaklik sinifi
   {
   private:
      int feet;
      float inches;
   public:
      Distance()                            //kurucu fonk. (argumansiz)
         { feet = 0; inches = 0; }
      Distance(int ft, float in)            //kurucu fonk. (iki argumanli)
         { feet = ft; inches = in; }
      void getdist()                        //kullanicidan uzakligi al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }

      void showdist()                       //uzakligi ekranda goster
         { cout << feet << "\'-" << inches << '\"'; }

      void add_dist(Distance, Distance);    //deklarasyonlar
      void div_dist(Distance, int);
   };
//--------------------------------------------------------------
```

```
                                        // d2 ve d3 uzakliklarini topla
void Distance::add_dist(Distance d2, Distance d3)
   {
   inches = d2.inches + d3.inches;      //inches'leri topla
   feet = 0;                            //(muhtemel elde icin)
   if(inches >= 12.0)                   //eger toplam 12.0'yi gecerse,
      {                                 //inches'i 12.0 azalt
      inches -= 12.0;                   //ve
      feet++;                           //feet'i 1
      }                                 //artir
   feet += d2.feet + d3.feet;           //feet'leri topla
   }
//-----------------------------------------------------------
                                        //Distance'i tamsayi ile bol
void Distance::div_dist(Distance d2, int divisor)
   {
   float fltfeet = d2.feet + d2.inches/12.0;    //float'a cevir
   fltfeet /= divisor;                          //bolmeyi yap
   feet = int(fltfeet);                         //feet kismini al
   inches = (fltfeet-feet) * 12.0;              //inches kismini al
   }
//////////////////////////////////////////////////////////////
int main()
   {
   Distance distarr[100];               //100 Distance'lik bir dizi
   Distance total(0, 0.0), average;     //diger Distance'lar
   int count = 0;                       //Distance girdilerini sayar
   char ch;                          //kullanicinin yanitini tutacak karakter

   do {
      cout << "\nEnter a Distance ";    //kullanicidan Distance'lari al
      distarr[count++].getdist();       //bir dizinin
      cout << "\nDo another (y/n)? ";   //icine yerlestir
      cin >> ch;
      } while( ch != 'n' );

   for(int j=0; j<count; j++)           //tum Distance'lari
      total.add_dist( total, distarr[j] ); //total'e ekle
   average.div_dist(total, count);      //sayi ile bol

   cout << "\nThe average is: ";        //ortalamayi ekranda goster
   average.showdist();
   cout << endl;
   return 0;
   }
```

Bölüm 8

Soruların Yanıtları

1. a, c
2. `x3.subtract(x2,x1);`
3. `x3=x2 -x1;`
4. doğru
5. `void operator --(){count--;}`
6. hiç argüman gerekmez

7. b, d
8.

```
void Distance::operator ++()
   {
   ++feet;
   }
```

9.

```
Distance Distance::operator ++()
   {
   int f = ++feet;
   float i = inches;
   return Distance(f, i);
   }
```

10. Değişkeni, kullanmadan önce artırır. Tıpkı aşırı yüklenmiş ++ operatörü gibi davranır.
11. c, e, b, a, d
12. doğru
13. b, c
14.

```
String String::operator ++()
   {
   int len = strlen(str);
   for(int j=0; j<len; j++)
      str[j] = toupper(str[j] )
   return String(str);
   }
```

15. d
16. eğer bir dönüşüm rutini mevcutsa, yanlış; aksi halde, doğru
17. b
18. doğru
19. kurucu fonksiyon
20. doğru, fakat insanların bunu anlaması güç olacaktır
21. d
22. nitelikler, işlemler
23. yanlış
24. a

Alıştırmaların Çözümleri

1.

```
// ex8_1.cpp
// asiri yuklenen '-' operatoru iki Distance'i birbirinden cikarir
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance          //Ingiliz olcu sistemine gore Distance sinifi
```

```
   {
   private:
      int feet;
      float inches;
   public:                                  //kurucu fonk. (argumansiz)
      Distance() : feet(0), inches(0.0)
         { }                                //kurucu fonk. (iki argumanli)
      Distance(int ft, float in) : feet(ft), inches(in)
         { }
      void getdist()                        //kullanicidan uzakligi al
         {
         cout << "\nEnter feet: "; cin >> feet;
         cout << "Enter inches: "; cin >> inches;
         }
      void showdist()                       //uzakligi ekranda goster
         { cout << feet << "\'-" << inches << '\"'; }

      Distance operator + (Distance);       //iki uzakligi topla
      Distance operator - (Distance);       //iki uzakligi birbirinden cikar
   };
//--------------------------------------------------------------
                                            //d2'yi bu uzakliga ekle
Distance Distance::operator + (Distance d2)  //toplami dondur
   {
   int f = feet + d2.feet;                  //feet'leri topla
   float i = inches + d2.inches;            //inches'leri topla
   if(i >= 12.0)                            //eger toplam 12.0'yi gecerse,
      {                                     //inches'i 12 azalt
      i -= 12.0;                            //ve
      f++;                                  //feet'i 1 artir
      }                                     //toplamin degeri atanmis,
   return Distance(f, i);                   //gecici bir Distance dondur
   }
//--------------------------------------------------------------
                                            //bu dist'ten d2'yi cikart
Distance Distance::operator - (Distance d2)  //farki dondur
   {
   int f = feet - d2.feet;                  //feet'leri cikart
   float i = inches - d2.inches;            //inches'leri cikart
   if(i < 0)                                //inches 0'dan kucukse,
      {                                     //inches'i 12.0 artir
      i += 12.0;                            //ve
      f--;                                  //feet'i 1 azalt
      }                                     //fark degeri atanmis,
   return Distance(f, i);                   //gecici bir Distance dondur
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist3;                   //uzakliklari tanimla
   dist1.getdist();                         //kullanicidan dist1'i al

   Distance dist2(3, 6.25);                 //dist2'yi tanimla,ilk kulanima hazirla

   dist3 = dist1 - dist2;                   //cikart
                                            //tum uzakliklari ekranda goster
   cout << "\ndist1="; dist1.showdist();
   cout << "\ndist2="; dist2.showdist();
   cout << "\ndist3="; dist3.showdist();
   cout << endl;
   return 0;
   }
```


2.

```
// ex8_2.cpp
// asiri yuklenen '+=' operatoru, karakter katarlarini pes pese ekler
#include <iostream>
#include <cstring>
using namespace std;                //strcpy(),strlen() icin
#include <process.h>
////////////////////////////////////////////////////////////////
                                    //exit() icin
class String      //kullanici tarafindan tanımlanan karakter katari sinifi
   {
   private:
      enum { SZ = 80 };
      char str[SZ];                 //String nesnelerinin boyutu
   public:                          //C karakter katari tutmak icin
      String()
         { strcpy(str, ""); }       //argumansiz kurucu fonk.
      String( char s[] )
         { strcpy(str, s); }        //1-argumanli kurucu fonk.
      void display()
         { cout << str; }           //String'i ekranda goster
      String operator += (String ss)  //buna bir String ekle
         {                            //sonuc bunda kalsin
         if( strlen(str) + strlen(ss.str) >= SZ)
            { cout << "\nString overflow "; exit(1); }
         strcat(str, ss.str);       //arguman karakter katari ekle
         return String(str);        //gecici String'i dondur
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   String s1 = "Merry Christmas!";     //1-argumanli kurucu fonk. kullanir
   String s2 = "Happy new year!";      //1-argumanli kurucu fonk. kullanir
   String s3;                          //argumansiz kurucu fonk. kullanir

   s3 = s1 += s2;                      //s2'yi s1'e ekle, s3'e ata

   cout << "\ns1="; s1.display();       //s1'i goster
   cout << "\ns2="; s2.display();       //s2'yi goster
   cout << "\ns3="; s3.display();       //s3'u goster
   cout << endl;
   return 0;
   }
```

3.

```
// ex8_3.cpp
// asiri yuklenen '+' operatoru, iki saati toplar
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class time
```

```
    {
    private:
       int hrs, mins, secs;
    public:
       time() : hrs(0), mins(0), secs(0)          //argumansiz kurucu fonk.
          { }                                     //3-argumanli kurucu fonk.
       time(int h, int m, int s) : hrs(h), mins(m), secs(s)
          { }
       void display()                             //format 11:59:59
          { cout << hrs << ":" << mins << ":" << secs; }
       time operator + (time t2)                  //iki saati topla
          {
          int s = secs + t2.secs;          //saniyeleri ekle
          int m = mins + t2.mins;          //dakikalari ekle
          int h = hrs + t2.hrs;            //saatleri ekle
          if( s > 59 )                     //eger saniyelerde tasma varsa,
             { s -= 60; m++;}              //   elde bir dakika tasi
          if( m > 59)                      //eger dakikalarda tasma varsa,
             { m -= 60; h++;}              //   elde bir saat tasi
          return time(h, m, s);            //gecici degeri dondur
          }
    };
//////////////////////////////////////////////////////////////
int main()
    {
    time time1(5, 59, 59);                 //iki adet saat tanimla ve
    time time2(4, 30, 30);                 //bunlari ilk kullanima hazirla
    time time3;                            //bir baska saat tanimla

    time3 = time1 + time2;                     //iki saati topla
    cout << "\ntime3="; time3.display();       /sonucu ekranda goster
    cout << endl;
    return 0;
    }
```

4.

```
// ex8_4.cpp
// Int tipi ile calisan asiri yuklenmis aritmetik operatorler
#include <iostream>
using namespace std;
#include <process.h>       //exit() icin
//////////////////////////////////////////////////////////////
class Int
   {
   private:
      int i;
   public:
      Int() : i(0)                         //argumansiz kurucu fonk.
         { }
      Int(int ii) : i(ii)                  //1-argumanli kurucu fonk.
         { }                               //(int'ten Int'e)
      void putInt()                        //Int'i ekranda goster
         { cout << i; }
      void getInt()
         { cin >> i; }                     //klavyeden Int oku
      operator int()                       //donusum operatoru
         { return i; }                     //(Int'ten int'e)
```



```
      Int operator + (Int i2)
         { return checkit( long double(i)+long double(i2) );}   //toplama
      Int operator - (Int i2)
         { return checkit( long double(i)-long double(i2) );}   //cikartma
      Int operator * (Int i2)
         { return checkit( long double(i)*long double(i2) );}   //carpma
      Int operator / (Int i2)
         { return checkit( long double(i)/long double(i2) );}   //bolme

      Int checkit(long double answer) //sonuclari kontrol et
         {
         if( answer > 2147483647.0L || answer < -2147483647.0L)
            { cout << "\nOverflow Error\n"; exit(1);}
         return Int( int(answer) );
         }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   Int alpha = 20;
   Int beta = 7;
   Int delta, gamma;

   gamma = alpha + beta;                        //27
   cout << "\ngamma="; gamma.putInt();
   gamma = alpha - beta;                        //13
   cout << "\ngamma="; gamma.putInt();
   gamma = alpha * beta;                        //140
   cout << "\ngamma="; gamma.putInt();
   gamma = alpha / beta;                        //2
   cout << "\ngamma="; gamma.putInt();

   delta = 2147483647;
   gamma = delta + alpha;                       //tasma hatasi
   delta = -2147483647;
   gamma = delta -alpha;                        //tasma hatasi

   cout << endl;
   return 0;
   }
```

Bölüm 9

Soruların Yanıtları

1. a, c
2. türetilmiştir
3. b, c, d
4. `class Bosworth : public Alphonso`
5. yanlış
6. `protected`
7. evet (`basefunc`'ın private olmadığını varsayarsak)
8. `BosworthObj.alfunc();`
9. doğru
10. türetilmiş sınıf içindeki

11. `Bosworth() : Alphonso(){}`
12. c, d
13. doğru
14. `Derv(int arg) : Base(arg)`
15. a
16. doğru
17. c
18. `class Tire : public Wheel, public Rubber`
19. `Base::func();`
20. yanlış
21. genelleştirme
22. d
23. yanlış
24. kuvvetlendirilmiş, birlik

Alıştırmaların Çözümleri

1.

```
// ex9_1.cpp
// publication sinifi ve turetilmis siniflar
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class publication                          //temel sinif
   {
   private:
      string title;
      float price;
   public:
      void getdata()
         {
         cout << "\nEnter title: "; cin >> title;
         cout << "Enter price: "; cin >> price;
         }
      void putdata() const
         {
         cout << "\nTitle: " << title;
         cout << "\nPrice: " << price;
         }
   };
////////////////////////////////////////////////////////////////
class book : private publication           //turetilmis sinif
   {
   private:
      int pages;
   public:
      void getdata()
         {
         publication::getdata();
         cout << "Enter number of pages: "; cin >> pages;
         }
      void putdata() const
         {
```



```
         publication::putdata();
         cout << "\nPages: " << pages;
         }
   };
////////////////////////////////////////////////////////////////
class tape : private publication           //turetilmis sinif
   {
   private:
      float time;
   public:
      void getdata()
         {
         publication::getdata();
         cout << "Enter playing time: "; cin >> time;
         }
      void putdata() const
         {
         publication::putdata();
         cout << "\nPlaying time: " << time;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   book book1;
   tape tape1;                             //yayinlari tanimla

   book1.getdata();
   tape1.getdata();                        //bunlar icin verileri al

   book1.putdata();
   tape1.putdata();                        //verilerini ekranda goster
   cout << endl;
   return 0;
   }
```

2.

```
// ex9_2.cpp
// String sinifindan aktarilan kalitim
#include <iostream>
#include <cstring>                         //strcpy() vs icin
using namespace std;
////////////////////////////////////////////////////////////////
class String                               //temel sinif
   {
   protected:                              //Dikkat: private olamaz
      enum { SZ = 80 };                    //tum String nesnelerinin buyuklugu
      char str[SZ];                        //C karakter katari tutmak icin
   public:
      String()                             //0. kurucu fonk., argumansiz
         { str[0] = '\0'; }
      String( char s[] )                   //1.kurucu fonk., bir argumanli
         { strcpy(str, s); }               // karakter katarini String'e cevir
      void display()const                  //String'i ekranda goster
         { cout << str; }
      operator char*()                     //donusum fonksiyonu
         { return str; }                   //String'i C karakter katarina cevir
```

```
   };
//////////////////////////////////////////////////////////////
class Pstring : public String                  //turetilmis sinif
   {
   public:
      Pstring( char s[] );                     //kurucu fonk.
   };
//--------------------------------------------------------------
Pstring::Pstring( char s[] )                   //Pstring icin kurucu fonk.
   {
   if(strlen(s) > SZ-1)                        //eger asiri uzunsa,
      {
      for(int j=0; j<SZ-1; j++)                //ilk SZ-1 karakteri "elle"
         str[j] = s[j];                        //kopyala
      str[j] = '\0';                           //null karakterini ekle
      }
   else                                        //asiri uzun degilse,
      String(s);                               //normal bicimde olustur
   }
//////////////////////////////////////////////////////////////
int main()
   {                                           //String'i tanimla
   Pstring s1 = "This is a very long string which is probably "
                "no, certainly--going to exceed the limit set by SZ.";
   cout << "\ns1="; s1.display();              //String'i goster

   Pstring s2 = "This is a short string."; //String'i tanimla
   cout << "\ns2="; s2.display();              //String'i goster
   cout << endl;
   return 0;
   }
```

3.

```
// ex9_3.cpp
// publication sinifi ile coklu kalitim
#include <iostream>
#include <string>
using namespace std;
//////////////////////////////////////////////////////////////
class publication
   {
   private:
      string title;
      float price;
   public:
      void getdata()
         {
         cout << "\nEnter title: "; cin >> title;
         cout << " Enter price: "; cin >> price;
         }
      void putdata() const
         {
         cout << "\nTitle: " << title;
         cout << "\n Price: " << price;
         }
   };
//////////////////////////////////////////////////////////////
```



```
class sales
   {
   private:
      enum { MONTHS = 3 };
      float salesArr[MONTHS];
   public:
      void getdata();
      void putdata() const;
   };
//--------------------------------------------------------------
void sales::getdata()
   {
   cout << "   Enter sales for 3 months\n";
   for(int j=0; j<MONTHS; j++)
      {
      cout << "   Month " << j+1 << ": ";
      cin >> salesArr[j];
      }
   }
//--------------------------------------------------------------
void sales::putdata() const
   {
   for(int j=0; j<MONTHS; j++)
      {
      cout << "\n   Sales for month " << j+1 << ": ";
      cout << salesArr[j];
      }
   }
//////////////////////////////////////////////////////////////
class book : private publication, private sales
   {
   private:
      int pages;
   public:
      void getdata()
         {
         publication::getdata();
         cout << "   Enter number of pages: "; cin >> pages;
         sales::getdata();
         }
      void putdata() const
         {
         publication::putdata();
         cout << "\n   Pages: " << pages;
         sales::putdata();
         }
   };
//////////////////////////////////////////////////////////////
class tape : private publication, private sales
   {
   private:
      float time;
   public:
      void getdata()
         {
         publication::getdata();
         cout << " Enter playing time: "; cin >> time;
         sales::getdata();
         }
```

```
    void putdata() const
       {
       publication::putdata();
       cout << "\n   Playing time: " << time;
       sales::putdata();
       }
  };
////////////////////////////////////////////////////////////////
int main()
   {
   book book1;                    //yayinlari tanimla
   tape tape1;

   book1.getdata();               //yayinlar icin gerekli verileri al
   tape1.getdata();

   book1.putdata();               //yayinlarla ilgili verileri ekranda goster
   tape1.putdata();
   cout << endl;
   return 0;
   }
```

Bölüm 10

Soruların Yanıtları

1. `cout <<&testvar;`
2. 4 byte'tır
3. c
4. `&var, *var, var&, char*`
5. sabit; değişkendir
6. `float* ptrtofloat;`
7. ismi
8. `*testptr`
9. işaret eder; işaret edilen değişkenin içeriği
10. b, c, d
11. Hayır. `&intvar` adresi, erişilmeden önce `intptr` işaretçisine yerleştirilmelidir.
12. herhangi veri tipine
13. Her ikisi de aynı görevi görür.
14.

```
for(int j=0; j<77; j++)
   cout << endl << *(intarr+j);
```

15. çünkü dizi ismi, dizinin adresini simgeler; dizinin adresi ise bir sabittir, değiştirilemez.
16. referans; işaretçi
17. a, d
18. `void func(char*);`
19.

```
for(int j=0; j<80; j++)
   *s2++ = *s1++;
```


20. b
21. `char* revstr(char*);`
22. `char* numptrs[] = { "One", "Two", "Three" };`
23. a, c
24. israfına
25. daha fazla ihtiyaç duyulmayacak belleği
26. `p->exclu();`
27. `objarr[7].exclu();`
28. a, c
29. `float* arr[8];`
30. b
31. 0..9 bir uçta; 3..* diğerinde
32. b
33. yanlış
34. a

Alıştırmaların Çözümleri

1.

```
// ex10_1.cpp
// Kullanici tarafindan girilen sayilarin ortalamasini hesaplar
#include <iostream>
using namespace std;

int main()
   {
   float flarr[100];              //sayilar icin dizi
   char ch;                       //kullanicinin karari
   int num = 0;                   //girilen sayilari sayar
   do
      {
      cout << "Enter number: ";   //kullanici 'n' girene kadar
      cin >> *(flarr+num++);      //kullanicidan sayilari alir
      cout << "   Enter another (y/n)? ";
      cin >> ch;
      }
   while( ch != 'n' );

   float total = 0.0;             //total, 0'dan baslar
   for(int k=0; k<num; k++)       //sayilari total'e ekle
      total += *(flarr+k);
   float average = total / num;   //ortalamayi hesapla ve goster
   cout << "Average is " << average << endl;
   return 0;
   }
```

2.

```
// ex10_2.cpp
// uye fonksiyonlar, String nesnelerini buyuk harflere cevirir
#include <iostream>
#include <cstring>                //strcpy() icin
#include <cctype>                 //toupper() icin
```

```
using namespace std;
//////////////////////////////////////////////////////////////
class String              //kullanici tarafindan tanimlanan karakter katari tipi
   {
   private:
      char* str;                    // karakter katarina isaret eden isaretci
   public:
      String(char* s)               //kurucu fonk.,bir argumanli
         {
         int length = strlen(s);    // karakter katari argumaninin uzunlugu
         str = new char[length+1];  //bellekte yer ayir
         strcpy(str, s);            //argumani bu bellege kopyala
         }
      ~String()                     //yok edici fonk.
         { delete str; }
      void display()                //String'i goster
         { cout << str; }
      void upit();                  //String'i buyuk harfe cevir
   };
//--------------------------------------------------------------
void String::upit()                 //her karakteri buyuk harfe cevir
   {
   char* ptrch = str;               //bu karakter katarini gosteren isaretci
   while( *ptrch )                  //null'a kadar,
      {
      *ptrch = toupper(*ptrch);     //her karakteri buyuk harfe cevir
      ptrch++;                      //bir sonraki karaktere gec
      }
   }
//////////////////////////////////////////////////////////////
int main()
   {
   String s1 = "He who laughs last laughs best.";

   cout << "\ns1=";                //karakter katarini goster
   s1.display();
   s1.upit();                       //karakter katarini buyuk harfe cevir
   cout << "\ns1=";                //karakter katarini goster
   s1.display();
   cout << endl;
   return 0;
   }
```

3.

```
// ex10_3.cpp
// karakter katarlarina isaret eden isaretci dizisini siralar
#include <iostream>
#include <cstring>                  //strcmp() vs. icin
using namespace std;
const int DAYS = 7;                 //dizi icindeki isaretci sayisi

int main()
   {
   void bsort(char**, int);         //prototip
                                    //char'a isaret eden isaretci dizisi
   char* arrptrs[DAYS] = { "Sunday", "Monday", "Tuesday",
                           "Wednesday", "Thursday",
```



```
                           "Friday", "Saturday" };
   cout << "\nUnSorted:\n";
   for(int j=0; j<DAYS; j++)
      cout << *(arrptrs+j) << endl;  //siralanmamis karakter katarlarini goster

   bsort(arrptrs, DAYS);
                                    // karakter katarlarini sirala
   cout << "\nSorted:\n";
   for(j=0; j<DAYS; j++)
      cout << *(arrptrs+j) << endl;  //sirali karakter katarlarini goster
   return 0;
   }
//--------------------------------------------------------------
void bsort(char** pp, int n)  //karakter katarlarini gosteren isaretcileri sirala
   {
   void order(char**, char**);      //prototip
   int j, k;                        //dizinin indeksleri

   for(j=0; j<n-1; j++)             //dis dongu
      for(k=j+1; k<n; k++)          //ic dongu, dis dongunun kaldigi yerden baslar
   order(pp+j, pp+k);               //isaretci iceriklerini sirala
   }
//--------------------------------------------------------------
void order(char** pp1, char** pp2)  //iki isaretciyi siralar
   {                                //1. karakter katari 2.den
   if( strcmp(*pp1, *pp2) > 0)      //buyukse,
      {
      char* tempptr = *pp1;         //isaretcileri takas et
      *pp1 = *pp2;
      *pp2 = tempptr;
      }
   }
```

4.

```
// ex10_4.cpp
// bagli liste bir yok edici fonksiyon icerir
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////////
struct link                         //listenin bir elemani
   {
   int data;                        //veri ogesi
   link* next;                      //bir sonraki baglantiyi gosteren isaretci
   };
//////////////////////////////////////////////////////////////
class linklist                      //baglantilar listesi
   {
   private:
      link* first;                  //ilk baglantiyi gosteren isaretci
   public:
      linklist()                    //argumansiz kurucu fonk.
         { first = NULL; }          //ilk baglanti mevcut degil
      ~linklist();                  //yok edici fonk.
      void additem(int d);          //veri ogesini ekle (bir baglanti)
      void display();               //baglantilarin tumunu goster
   };
```

```
//--------------------------------------------------------------
void linklist::additem(int d)          //veri ogesini ekle
   {
   link* newlink = new link;           //yeni bir baglanti olustur
   newlink->data = d;                  //verilerini yukle
   newlink->next = first;              //bu, bir sonraki baglantiyi gostersin
   first = newlink;                    //artik, first bunu gosteriyor
   }
//--------------------------------------------------------------
void linklist::display()               //tum baglantilari goster
   {
   link* current = first;              //isaretci, ilk baglantiyi gostersin
   while( current != NULL )            //son baglantiyla cik
      {
      cout << endl << current->data;   //verileri yazdir
      current = current->next;         //bir sonraki baglantiya gec
      }
   }
//--------------------------------------------------------------
linklist::~linklist()                  //yok edici fonk.
   {
   link* current = first;              //isaretci, ilk baglantiyi gostersin
   while( current != NULL )            //son baglantiyla cik
      {
      link* temp = current;            //bu baglantiyi gosteren isaretciyi sakla
      current = current->next;         //isaretciyi bir sonraki baglantiya kaydir
      delete temp;                     //bu baglantiyi sil
      }
   }
////////////////////////////////////////////////////////////////
int main()
   {
   linklist li;                        //bagli liste olustur

   li.additem(25);                     //listeye dort oge ekle
   li.additem(36);
   li.additem(49);
   li.additem(64);

   li.display();                       //tum listeyi goster
   cout << endl;
   return 0;
   }
```

Bölüm 11

Soruların Yanıtları

1. d
2. doğru
3. temel
4. `virtual void dang(int);` veya `void virtual dang(int);`
5. gecikmiş bağlama veya dinamik bağlama
6. türetilmiş
7. `virtual void aragorn()=0;` veya `void virtual aragorn()=0;`
8. a, c

9. `dong* parr[10];`
10. c
11. doğru
12. c, d
13. `friend void harry(George);`
14. a, c, d
15. `friend class harry;` veya `friend harry;`
16. c
17. Üyeleri tek tek kopyalar.
18. `zeta& operator = (zeta&);`
19. a, b, d
20. yanlış; derleyici bir varsayılan kurucu fonksiyon sağlar.
21. a, d
22. `Bertha(Bertha&);`
23. doğru, eğer bunun için bir sebep varsa
24. a, c
25. doğru; eğer referans olarak döndürürse problem çıkar.
26. Aynı şekilde işlev görürler.
27. a, b
28. onu kullanan fonksiyonun üye olduğu nesneye
29. hayır; `this` bir işaretçi olduğu için `this->da=37;` kullanmanız gerekir
30. `return *this;`
31. c
32. bağlantılar (linkler)
33. doğru
34. a, b, c

Alıştırmaların Yanıtları

1.

```
// ex11_1.cpp
// publication sinifi ve turetilmis siniflar
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class publication
   {
   private:
      string title;
      float price;
   public:
      virtual void getdata()
         {
         cout << "\nEnter title: "; cin >> title;
         cout << "Enter price: "; cin >> price;
         }
      virtual void putdata()
         {
         cout << "\n\nTitle: " << title;
         cout << "\nPrice: " << price;
```

```
      }
   };
//////////////////////////////////////////////////////////////
class book : public publication
   {
   private:
      int pages;
   public:
      void getdata()
         {
         publication::getdata();
         cout << "Enter number of pages: "; cin >> pages;
         }
      void putdata()
         {
         publication::putdata();
         cout << "\nPages: " << pages;
         }
   };
//////////////////////////////////////////////////////////////
class tape : public publication
   {
   private:
      float time;
   public:
      void getdata()
         {
         publication::getdata();
         cout << "Enter playing time: "; cin >> time;
         }
      void putdata()
         {
         publication::putdata();
         cout << "\nPlaying time: " << time;
         }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   publication* pubarr[100];  //yayinlara isaret eden isaretci dizisi
   int n = 0;                 //bir dizi icindeki yayin sayisi
   char choice;               //kullanicinin secimi

   do {
      cout << "\nEnter data for book or tape (b/t)? ";
      cin >> choice;
      if( choice == 'b' )        //kitap nesnesi tanimla
         pubarr[n] = new book;   // diziye yerlestir
      else                       //teyp nesnesi olustur
         pubarr[n] = new tape;   // diziye yerlestir
      pubarr[n++]->getdata();    //nesne icin verileri al
      cout << " Enter another (y/n)? ";    //bir baska yayin var mi?
      cin >> choice;
      }
   while( choice == 'y' );       //'y ' olmayana kadar tekrarla

   for(int j=0; j<n; j++)        //tum yayinlar icin tekrarla
      pubarr[j]->putdata();      //yayinin verilerini yazdir
   cout << endl;
```



```
   return 0;
   }
```

2.

```
// ex11_2.cpp
// Distance icin friend square()fonksiyonu
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////////
class Distance              //Ingiliz olcu sistemine gore Distance sinifi
   {
   private:
      int feet;
      float inches;
   public:
      Distance()                          //kurucu fonk. (argumansiz)
         { feet = 0; inches = 0.0; }
      Distance(float fltfeet)             //kurucu fonk. (bir argumanli)
         {                                //fit, tamSayi kismi
         feet = static_cast<int>(fltfeet);
         inches = 12*(fltfeet-feet);  //geriye kalanlar, inc
         }                                //kurucu fonk. (iki argumanli)
      Distance(int ft, float in) : feet(ft), inches(in)
         { }
      void showdist()                     //uzakligi goster
         { cout << feet << "\'-" << inches << '\"'; }
      friend Distance operator * (Distance,Distance);     //friend
   };
//--------------------------------------------------------------
                                          //d1'i d2 ile carp
Distance operator * (Distance d1, Distance d2)
   {
   float fltfeet1 = d1.feet + d1.inches/12;       //float'a cevir
   float fltfeet2 = d2.feet + d2.inches/12;
   float multfeet = fltfeet1 * fltfeet2;          //carpimi bul
   return Distance(multfeet);                     //gecici Distance'i dondur
   }
//////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1(3, 6.0);                        //birkac uzaklik olustur
   Distance dist2(2, 3.0);
   Distance dist3;

   dist3 = dist1 * dist2;                         //carpma

   dist3 = 10.0 * dist3;                          //carp ve donustur
                                                  //tum uzakliklari goster
   cout << "\ndist1="; dist1.showdist();
   cout << "\ndist2="; dist2.showdist();
   cout << "\ndist3="; dist3.showdist();
   cout << endl;
   return 0;
   }
```

3.

```
// ex11_3.cpp
// dizi sinifi olusturur
// atama operatorunu ve kopyalama kurucu fonksiyonunu asiri yukler
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Array
   {
   private:
      int* ptr;                          //"dizi " icerigine isaret eder
      int size;                          //dizinin buyuklugu
   public:
      Array() : ptr(0), size(0)          //argumansiz kurucu fonk.
         { }
      Array(int s) : size(s)             //bir argumanli kurucu fonk.
         { ptr = new int[s]; }
      Array(Array&);                     //kopyalama kurucu fonk.
      ~Array()                           //yok edici fonk.
         { delete[] ptr; }
      int& operator [] (int j)           //asiri yuklenen indeks operatoru
         { return *(ptr+j); }
      Array& operator = (Array&);        //asiri yuklenen = operatoru
   };
//--------------------------------------------------------
Array::Array(Array& a)                   //kopyalama kurucu fonk.
   {
   size = a.size;                        //yenisi ayni boyutta
   ptr = new int[size];                  //icerik icin yer ayir
   for(int j=0; j<size; j++)             //icerigi yeni yere kopyala
      *(ptr+j) = *(a.ptr+j);
   }
//--------------------------------------------------------
Array&Array::operator = (Array&a)        //asiri yuklenen = operatoru
   {
   delete[] ptr;                         //eski icerigi sil (eger varsa)
   size = a.size;                        //bu nesne ayni buyuklukte olsun
   ptr = new int[a.size];                //yeni icerik icin yer ayir
   for(int j=0; j<a.size; j++)           //icerigi bu nesneye kopyala
      *(ptr+j) = *(a.ptr+j);
   return * this;                        //bu nesneyi dondur
   }
////////////////////////////////////////////////////////////////
int main()
   {
   const int ASIZE = 10;                 //dizinin buyuklugu

   Array arr1(ASIZE);                    //bir dizi olustur

   for(int j=0; j<ASIZE; j++)            //icini karelerle doldur
      arr1[j] = j*j;

   Array arr2(arr1);                     //kopyalama kurucu fonksiyonunu kullan
   cout << "\narr2: ";
   for(j=0; j<ASIZE; j++)                //calistigini kontrol et
      cout << arr2[j] << " ";
```



```
   Array arr3, arr4;
   arr4 = arr3 = arr1;                   //iki bos Array nesnesi tanimla
   cout << "\narr3: ";                    //atama operatorunu kullan
   for(j=0; j<ASIZE; j++)
      cout << arr3[j] << " ";            //bunun arr3 uzerinde calistigini kontrol et
   cout << "\narr4: ";
   for(j=0; j<ASIZE; j++)
      cout << arr4[j] << " ";            //bunun arr4 uzerinde calistigini kontrol et
   cout << endl;
   return 0;
   }
```

Bölüm 12

Soruların Yanıtları

1. b, c
2. **ios**
3. **ifstream**, **ofstream** ve **fstream**
4. **ofstream salefile ("SALES.JUN");**
5. doğru
6. **if(foobar)**
7. d
8. **fileOut.put(ch);** (**ch, yazilacak karakterdir**)
9. c
10. **ifile.read((char*)buff, sizeof(buff));**
11. a, b, d
12. Bir sonraki okuma veya yazma işleminin gerçekleştireceği konum
13. yanlış; dosya işaretçisi, mevcut konum ile eş anlamlı olabilir
14. **.f1.seekg(-13, ios::cur);**
15. b
16. b, c
17. **skipws** girdideki görünmeyen karakterlerin dikkate alınmamasını sağlar, böylece **cin**, girdinin sona erdiğini farz etmeyecektir.
18. **int main(int argc, char *argv[])**
19. **PRN, LPT1**
20. **istream& operator >> (istream&, Sample&)**

Alıştırmaların Çözümleri

1.

```
// ex12_1.cpp
// dosyaya yaz
#include <iostream>
#include <fstream>                       //dosya akislari icin
using namespace std;
////////////////////////////////////////////////////////////////
class Distance      //Ingiliz olcu sistemine gore Distance sinifi
   {
   private:
      int feet;
```

```
      float inches;
   public:
      Distance() : feet(0), inches(0.0)     //kurucu fonk. (argumansiz)
         { }                                //kurucu fonk. (iki argumanli)
      Distance(int ft, float in) : feet(ft), inches(in)
         { }
      void getdist()                        //uzakligi kullanicidan al
         {
         cout << "\n   Enter feet: "; cin >> feet;
         cout << "   Enter inches: "; cin >> inches;
         }
      void showdist()                       //uzakligi goster
         { cout << feet << "\'-" << inches << '\"'; }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   char ch;
   Distance dist;                           //bir Distance nesnesi tanimla
   fstream file;                            //giris/cikis dosyasi tanimla
                                            //dosyayi ekleme icin ac
   file.open("DIST.DAT", ios::binary | ios::app |
                         ios::out | ios::in );
   do                                       //kullanidan gelen veriler dosyaya
      {
      cout << "\nDistance";
      dist.getdist();                       //bir uzaklik al
                                            //dosyaya yaz
      file.write( (char*)&dist, sizeof(dist) );
      cout << "Enter another distance (y/n)? ";
      cin >> ch;
      }
   while(ch=='y');                          //'n ' ile cik

   file.seekg(0);                           //dosyanin basina geri don
                                            //ilk uzakligi oku
   file.read( (char*)&dist, sizeof(dist) );
   int count = 0;
   while( !file.eof() )                     //EOF ile cik
      {
      cout << "\nDistance" << ++count << ":";      //uzakligi goster
      dist.showdist();
      file.read( (char*)&dist, sizeof(dist) );     //bir baska
      }                                            //uzaklik daha oku
   cout << endl;
   return 0;
   }
```

2.

```
// ex12_2.cpp
// COPY komutunu taklit eder
#include <fstream>                      //dosya fonksiyonlari icin
#include <iostream>
using namespace std;
#include <process.h>                    //exit() icin

int main(int argc, char*argv[] )
```



```
{
if( argc != 3 )
   { cerr << "\nFormat: ocopy srcfile destfile "; exit(-1); }
char ch;
ifstream infile;                        //okunacak karakter
infile.open( argv[1] );                 //girdi icin dosya tanimla
if( !infile )                           //dosyayi ac
   { cerr << "\nCan't open " << argv[1]; exit(-1); }  //hatalari kontrol et

ofstream outfile;
outfile.open( argv[2] );                //cikti icin dosya tanimla
if( !outfile )                          //dosyayi ac
   { cerr << "\nCan't open " << argv[2]; exit(-1); }  //hatalari kontrol et
while( infile )
   {                                    // EOF'a kadar
   infile.get(ch);
   outfile.put(ch);                     //bir karakter oku
   }                                    //karakteri yaz
return 0;
}
```

3.

```
// ex12_3.cpp
// dosyanin boyutunu gosterir
#include <fstream>
#include <iostream>                     //dosya fonksiyonlari icin
using namespace std;
#include <process.h>
                                        //exit() icin

int main(int argc, char* argv[] )
{
   if( argc != 2 )
      { cerr << "\nFormat: filename\n"; exit(-1); }
   ifstream infile;                     //girdi icin dosya ac
   infile.open( argv[1] );              //dosyayi ac
   if( !infile )                        //hatalari kontrol et
      { cerr << "\nCan't open " << argv[1]; exit(-1); }
   infile.seekg(0, ios::end);           //dosyanin sonuna git
                                        //byte sayisini bildir
   cout << "Size of " << argv[1] << " is " << infile.tellg();
   cout << endl;
   return 0;
   }
```

Bölüm 13

Soruların Yanıtları

1. a, b, c, d
2. `#include` direktifini
3. `.CPP` dosyasını derlemek için derleyici ve ortaya çıkan `.OBJ` dosyalarını bağlamak için bir bağlayıcı
4. a, b
5. sınıf kütüphanesi

6. doğru
7. c, d
8. doğru
9. yanlış
10. a, c, d
11. bağlama ile (bağlanarak)
12. yanlış
13. d
14. kapsam
15. nesne
16. B dosyası, bildirilmelidir.
17. doğru
18. b
19. yanlış
20. d
21. b
22. **namespace**
23. b, d

Bölüm 14

Soruların Yanıtları

1. b ve c
2. **class**
3. yanlış; derleme sırasında farklı fonksiyonlar oluşturulur
4.

```
template<class T>
T times2(T arg)
   {
   return arg*2;
   }
```

5. b
6. doğru
7. örnekleme
8. c
9. sabit veri tipi, herhangi bir veri tipi
10. verileri, saklayan
11. c
12. **try**, **catch** ve **throw**
13. **throw BoundsError();**
14. yanlış; bir deneme bloğunun parçası olmaları lazım
15. d
16.

```
class X
   {
   public:
      int xnumber;
      char xname[MAX];
      X(int xd, char* xs)
         {
         xnumber = xd;
         strcpy(xname, xs);
         }
   };
```

17. yanlış
18. a ve d
19. d
20. doğru
21. bağımsız, bağımlı
22. a
23. yanlış
24. ek bilgi

Alıştırmaların Çözümleri

1.

```
// ex14_1.cpp
// dizinin ortalamasini bulan fonksiyon icin sablon kullanir
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
template <class atype>                    //fonksiyon sablonu
atype avg(atype* array, int size)
   {
   atype total = 0;
   for(int j=0; j<size; j++)              //dizinin ortalamasini bul
      total += array[j];
   return (atype)total/size;
   }
////////////////////////////////////////////////////////////////
int intArray[] =        { 1, 3, 5, 9, 11, 13 };
long longArray[] =      { 1, 3, 5, 9, 11, 13 };
double doubleArray[] =  { 1.0, 3.0, 5.0, 9.0, 11.0, 13.0 };
char charArray[] =      { 1, 3, 5, 9, 11, 13 };

int main()
   {
   cout << "\navg(intArray)=" << avg(intArray, 6);
   cout << "\navg(longArray)=" << avg(longArray, 6);
   cout << "\navg(doubleArray)=" << avg(doubleArray, 6);
   cout << "\navg(charArray)=" << (int)avg(charArray, 6) <<endl;
   return 0;
   }
```

2.

```
// ex14_2.cpp
// kuyruk sinifini sablon olarak uygular
#include <iostream>
using namespace std;
const int MAX = 3;
```

```
////////////////////////////////////////////////////////////
template <class Type>
class Queue
   {
   private:
      Type qu[MAX];                    //herhangi tipte bir dizi
      int head;                        //kuyrugun basinin indeksi (ogeler buradan cikart)
      int tail;                        //kuyrugun sonunun indeksi (ogeleri buradan ekle)
   public:
      Queue()                          //kurucu fonk.
         { head=-1; tail=-1; }
      void put(Type var)               //kuyrugun sonuna ekle
         {
         qu[++tail] = var;
         if(tail >= MAX-1)             //dizinin sonunu gectiyse, basa don
         tail = -1;
         }
      Type get()                       //kuyrugun basindan cikart
         {
         Type temp = qu[++head];       //ogeyi sakla
         if(head >= MAX-1)             //dizinin sonunu gectiyse, basa don
            head = -1;
         return temp;                  //ogeyi dondur
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   Queue<float> q1;                    //q1, Queue<float> sinifinin bir nesnesi

   q1.put(1111.1F);                    //3'u yerlestir
   q1.put(2222.2F);
   q1.put(3333.3F);
   cout << "1: " << q1.get() << endl;  //2'yi al
   cout << "2: " << q1.get() << endl;
   q1.put(4444.4F);                    //2'yi yerlestir
   q1.put(5555.5F);
   cout << "3: " << q1.get() << endl;  //1'i al
   q1.put(6666.6F);                    //1'i yerlestir
   cout << "4: " << q1.get() << endl;  //3'u al
   cout << "5: " << q1.get() << endl;
   cout << "6: " << q1.get() << endl;

   Queue<long> q2;                     //q2, Queue<long> sinifinin bir nesnesi

   q2.put(123123123L);                 //3 long yerlestir, 3 long al
   q2.put(234234234L);
   q2.put(345345345L);
   cout << "1: " << q2.get() <<endl;
   cout << "2: " << q2.get() <<endl;
   cout << "3: " << q2.get() <<endl;
   return 0;
   }
```

3.

```
// ex14_3.cpp
// kuyruk sinifini sablon olarak uygular
// kuyruktaki hatalari kontrol altina almak icin kural dısı durumlari
```



```
// kullanır
#include <iostream>
using namespace std;
const int MAX = 3;
////////////////////////////////////////////////////////////
template <class Type>
class Queue
   {
   private:
      Type qu[MAX];
      int head;                        //herhangi bir tipte bir dizi
      int tail;                        //kuyrugun basının indeksi (eski ogeyi buradan cikart)
      int count;                       //kuyrugun sonunun indeksi (yeni ogeyi buradan ekle)
   public:                             //kuyruktaki oge sayısı
      class full { };                  //kural dısı durum sınıfları
      class empty { };
//--------------------------------------------------------------
      Queue()                          //kurucu fonk.
         { head=-1; tail=-1; count=0; }

      void put(Type var)               //kuyrugun sonuna ekle
         {
         if(count >= MAX)
            throw full();              //kuyruk zaten doluysa,
         qu[++tail] = var;             //kural disi bir durum firlat
         ++count;                      //ogeyi sakla
         if(tail >= MAX-1)
            tail = -1;                 //dizinin sonunu gecmisse, basa don
         }
//--------------------------------------------------------------
      Type get()                       //kuyrugun basindan oge cikart
         {
         if(count <= 0)                //eger kuyruk bossa,
            throw empty();             //kural disi bir durum firlat
         Type temp = qu[++head];       //ogeyi al
         --count;
         if(head >= MAX-1)             //eger dizinin sonunu çecmisse, basa don
            head = -1;
         return temp;                  //ogeyi dondur
         }
   };
////////////////////////////////////////////////////////////
int main()
   {
   Queue<float> q1;                    //q1, Queue<float> sinifinin bir nesnesi
   float data;                         //kulanicidan alinan veri ogesi
   char choice = 'p';                  //'x','p' veya 'g'

   do                                  //do dongusu (cikmak icin 'x' gir)
      {                                //deneme blogu
      try
         {
         cout << "\nEnter 'x' to exit, 'p' for put, 'g' for get: ";
         cin >> choice;
         if( choice == 'p' )
            {
            cout << "Enter data value: ";
            cin >> data;
            q1.put(data);
```

```
        }
      if( choice == 'g' )
         cout << "Data=" << q1.get() << endl;
      }     //deneme blogunun sonu
   catch(Queue<float>::full)
      {
      cout << "Error: queue is full." << endl;
      }
   catch(Queue<float>::empty)
      {
      cout << "Error: queue is empty." << endl;
      }
   } while(choice != 'x');
return 0;
}     //main()'in sonu
```

Bölüm 15

Soruların Yanıtları

1. a, b, d
2. vektör, liste, çift uçlu kuyruk
3. küme, eşleme
4. a
5. doğru
6. c
7. yanlış
8. iteratör
9. bir fonksiyon nesnesi
10. c
11. yanlış; o sadece değerini döndürür
12. 3, 11
13. çift
14. b, c
15. işaret eder
16. yanlış
17. iki yönlü
18. `*iter++`
19. d
20. c
21. doğru
22. iteratör
23. bu argüman, basılan değerleri birbirinden ayırmak için kullanılan bir string'dir.
24. b
25. elemanların nasıl sıralanacağını
26. doğru
27. çift (veya bağlantılar-associations)
28. yanlış
29. a, d
30. kurucu fonksiyonunda

Alıştırmaların Çözümleri

1.

```
// ex15_1.cpp
// dizide saklanan float tipi, sort() ile siralanir
#include <iostream>
#include <algorithm>
using namespace std;

int main()
   {
   int j = 0, k;
   char ch;
   float fpn, farr[100];

   do {
      cout << "Enter a floating point number: ";
      cin >> fpn;
      farr[j++] = fpn;
      cout << "Enter another ('y' or 'n'): ";
      cin >> ch;
      } while(ch == 'y');
   sort(farr, farr+j);
   for(k=0; k<j; k++)
      cout << farr[k] << ",";
   cout << endl;
   return 0;
   }
```

2.

```
// ex15_2.cpp
// string nesneleri, push_back() ve [] ile kullanilan vektor
#include <iostream>
#include <string>
#pragma warning (disable:4786)      //sadece Microsoft derleyici icin
#include <vector>
#include <algorithm>
using namespace std;

int main()
   {
   vector<string> vectStrings;
   string word;
   char ch;

   do {
      cout << "Enter a word: ";
      cin >> word;
      vectStrings.push_back(word);
      cout << "Enter another ('y' or 'n'): ";
      cin >> ch;
      } while(ch == 'y');
   sort( vectStrings.begin(), vectStrings.end() );
   for(int k=0; k<vectStrings.size(); k++)
      cout << vectStrings[k] << endl;
```

```
   return 0;
   }
```

3.

```
// ex15_3.cpp
// ev yapimi reverse() algoritmasi, listeyi tersine cevirir
#include <iostream>
#include <list>
using namespace std;

int main()
   {
   int j;

   list<int> theList;
   list<int>::iterator iter1;
   list<int>::iterator iter2;

   for(j=2; j<16; j+=2)            //listeyi 2, 4, 6,... ile doldur
      theList.push_back(j);
   cout << "Before reversal: ";    //listeyi goster
   for(iter1=theList.begin(); iter1 != theList.end(); iter1++)
      cout << *iter1 << " ";

   iter1 = theList.begin();        //ilk elemana ayarla
   iter2 = theList.end();          //sonuncudan bir sonrakine ayarla
   --iter2;                        //sonuncuya gec

   while(iter1 != iter2)
      {
      swap(*iter1, *iter2);        //ondeki ve arkadakini takas et
      ++iter1;                     //ondeki artir
      if(iter1==iter2)             //eleman sayisi cift ise,
         break;
      --iter2;                     //arkayi bir azalt
      }
   cout << "\nAfter reversal: ";   //listeyi goster
   for(iter1=theList.begin(); iter1 != theList.end(); iter1++)
      cout << *iter1 << " ";
   cout << endl;
   return 0;
   }
```

4.

```
// ex15_4.cpp
// isaretci ile saklanan person nesnelerini otomatik olarak siralayan bir
// coklu kume
#include <iostream>
#include <set>
#pragma warning (disable:4786)
#include <string>
using namespace std;

class person
   {
```



```
   private:
      string lastName;
      string firstName;
      long phoneNumber;
   public:
      person():
            lastName("blank "), firstName("blank "), phoneNumber(0L)
         { }                                  //varsayilan kurucu fonk.

      person(string lana, string fina, long pho) :
                  lastName(lana), firstName(fina), phoneNumber(pho)
         { }                                  //3-argumanli kurucu fonk.
      friend bool operator<(const person&, const Person&);

      void display()const
         {                                    //kisinin verilerini ekranda goster
         cout << endl << lastName << ",\t" << firstName
              <<"\t\tPhone: " << phoneNumber;
         }
      long get_phone()const                   //telefon numarasini dondur
         { return phoneNumber; }
   };    //class person'in sonu
//-----------------------------------------------------------
// person sinifi icin asiri yuklenen <
bool operator<(const person& p1, const person& p2)
   {
   if(p1.lastName == p2.lastName)
      return (p1.firstName < p2.firstName) ? true : false;
   return (p1.lastName < p2.lastName) ? true : false;
   }
//-----------------------------------------------------------
// isaretcileri kullanarak kisileri karsilastirmak icin fonksiyon nesnesi
class comparePersons
   {
   public:
   bool operator() (const person* ptrP1,
                    const person* ptrP2) const
      { return *ptrP1 < *ptrP2; }
   };
////////////////////////////////////////////////////////////////
int main()
   {                          //kisileri gosteren isaretci coklu kumesi
   multiset<person*, comparePersons> setPtrsPers;
   multiset<person*, comparePersons>::iterator iter;
                                 //kisileri olustur
   person* ptrP1 = new person("KuangThu", "Bruce", 4157300);
   person* ptrP2 = new person("McDonald", "Stacey", 3327563);
   person* ptrP3 = new person("Deauville", "William", 8435150);
   person* ptrP4 = new person("Wellington", "John", 9207404);
   person* ptrP5 = new person("Bartoski", "Peter", 6946473);
   person* ptrP6 = new person("McDonald", "Amanda", 8435150);
   person* ptrP7 = new person("Fredericks", "Roger", 7049982);
   person* ptrP8 = new person("McDonald", "Stacey", 7764987);

   setPtrsPers.insert(ptrP1);    //kisileri coklu kumenin icine yerlestir
   setPtrsPers.insert(ptrP2);
   setPtrsPers.insert(ptrP3);
   setPtrsPers.insert(ptrP4);
   setPtrsPers.insert(ptrP5);
```

```
    setPtrsPers.insert(ptrP6);
    setPtrsPers.insert(ptrP7);
    setPtrsPers.insert(ptrP8);
                                //coklu kumeyi goster
    cout << "\n\nSet sorted when created:";
    for(iter=setPtrsPers.begin(); iter != setPtrsPers.end(); iter++ )
       (**iter).display();

    iter = setPtrsPers.begin();   //kisilerin tumunu goster
    while( iter != setPtrsPers.end() )
       {
       delete *iter;        //kisiyi yok et
       setPtrsPers.erase(iter++); //isaretci ortadan kaldir
       }
    cout << endl;
    return 0;
    }     //main()'in sonu
```

Bölüm 16

Soruların Yanıtları

1. yanlış
2. c, d
3. görev
4. doğru
5. sütuna
6. a, c
7. genelleştirme, bağlantı (association), birlik (aggregation)
8. a, d
9. yanlış
10. a
11. doğru
12. yanlış
13. a, b, c, d
14. insanlar, programla (veya sistemle), diğer sistemler
15. b
16. b, c
17. yanlış
18. a, d
19. b, d
20. yanlış
21. c, d
22. nesneden
23. doğru
24. a, d

Dizin

TÜRKİYE BİLGİSAYARI BU KİTAPLARLA ÖĞRENİYOR!

Bilgiye Giden Yol Herkese Açıktır!

ISBN 975-297-128-8

ALFA

ALFA Basım Yayım Dağıtım Ltd.
Ticarethane Sk. No:41/1
34410 Cağaloğlu-İstanbul
Tel : +90 (212) 511 53 03
+90 (212) 513 87 51
Fax : +90 (212) 519 33 00
e-mail:info@alfakitap.com
www.alfakitap.com