

	00000
BÖLÜM 1 : C++'a Giriş.....	1
BÖLÜM 2 : Sınıflara Giriş.....	31
BÖLÜM 3 : Sınıfları Daha Yakından İnceleyelim.....	65
BÖLÜM 4 : Diziler, İşaretçiler ve Başvurular.....	87
BÖLÜM 5 : Fonksiyonların Aşırı Yüklemesi.....	117
BÖLÜM 6 : Operatörleri Aşırı Yüklemek.....	143
BÖLÜM 7 : Miras.....	169
BÖLÜM 8 : C++'ın I/O Sistemine Giriş.....	197
BÖLÜM 9 : İleri Düzey C++ I/O'su.....	225
BÖLÜM 10 : Sanal Fonksiyonlar.....	253
BÖLÜM 11 : Şablonlar ve Hata Denetimi.....	271
BÖLÜM 12 : Çalışma Anı Tip Tanıma ve Tip Dönüşürme Operatörleri.....	297
BÖLÜM 13 : Namespace'ler, Dönüşüm Fonksiyonları ve Değişik Konular.....	319
BÖLÜM 14 : Standart Şablon Kütüphanesi.....	347
Ek A : C ve C++ Arasındaki Diğer Farklar.....	389
Ek B : Cevaplar.....	391

BÖLÜM 1

C++'a Giriş

Nesneye Yönelik Programlama Nedir?

C++'ın İki Sürümü

C++'ın I/O Konsolu

C++'da Açıklamalar

Sınıflara Giriş

C ve C++ Arasındaki Bazı Farklılıklar

Fonksiyonların Aşırı Yüklenmesine Giriş

C++ Anahtar Kelimeleri

C++ dili, C dilinin geliştirilmesiyle oluşturulmuştur. Bu dil C'nin kapsamakta olduklarına ek olarak nesneye dayalı programlamayı da (kısaca OOP) destekliyor. C'yi "daha iyi" hale getiren pek çok yeni özelliği vardı. Birkaç ayrıntıyı göz ardı edersek C++, C'nin iyileştirilmiş halidir diyebiliriz. Bu dil, C hakkında tüm bildiklerinizle tamamen uyumludur, fakat C++'ın gelişmiş özelliklerini anlamak için yine de önemli bir zaman ve emek yatırımı yapmanız gerekecek. Olsun, onun size kazandıracıkları ile kıyaslayacak olursak bu çaba oldukça ömensiz kalır.

Bu bölümde amacımız, sizi bu dilin en önemli özellikleri ile tanıştırmak. Bildiğiniz gibi, bilgisayar dilleri boşlukta sallanan parçalardan oluşmaz, tam tersine onları oluşturan parçalar bir aradadırlar ve işbirliği içerisinde çalışarak dili meydana getirirler. Bu sıkı ilişkiye özellikle C++'da çok sık telaffuz edeceğiz. Hatta C++'dan bahsederken bir konuyu diğer bir konudan yaşıtarak anlatmak mümkün değildir. C++'ın özellikleri birbirine çok sıkı bir şekilde bağlıdır. Bu nedenle C++'ın birkaç özelliğine kısaca değinerek girişimizi yapmayı uygun gördük. Bu girişî okuduktan sonra ilerideki konularda vereceğimiz örnekleri anlamamız kolaylaşacak. Burada dejindiğimiz konuları daha sonra tek tek ayrıntılarıyla inceleyeceğiz.

C++, nesneye dayalı programlamayı desteklemek üzere oluşturuldu, gelin biz de bu konuya işe başlayalım. Siz de göreceksiniz, bu dilin pek çok özelliği bir şekilde OOP ile ilgili dir. Hatta OOP teorisi, C++'ın özüdür de denilebilir. Fakat şunu anlamamız gerekiyor: C++'ı, nesneye dayalı olan ve *olmayan* programları yazmak için kullanabiliriz. Onu nasıl kullanacağımız tamamen bize bağlıdır.

Bu yazının hazırlanışı sırasında C++'ın standartlaştırılma işlemleri de tamamlanıyordu. Bu nedenle ilk bölümde birkaç yıldır kullanılan C++ ile yeni Standart C++ arasındaki bazı önemli farkları anlatmak istedim. Bu kitapta Standart C++'ı anlatacağız. Bu nedenle özellikle de eski bir derleyici kullanıyorsanız kitabın içeriği daha da önem kazanacak.

Burada C++'ın çeşitli özelliklerini tanıtabaçız, C ve C++ programlama stilleri arasındaki bazı farklılıklarını tartışacağız. Yeni özelliklerle C++'da program hazırlanması daha esnek hale getirilmiştir. Bu özelliklerden bazıları nesnel programlama ile ilgili değildir, bu nedenle C++ ile hazırlanan programların çoğunda onlarla karşılaşacağız. Biz de bunu düşünerek kitapta onlara daha önce yer verdik.

Fakat C++'ın genel yapısı üzerine önce biraz konuşalım. Öncelikle şunu söyleyelim, genelde C++ programları fiziksel açıdan C programlarına benzerler. C programları, `main()`'in içerisindeki yordam çalıştırılarak başlar. C++, komut satırı argümanları için tipki C gibi `argc` ve `argv`'yi kullanır. C++'ın kendi nesneye dayalı kütüphanesi vardır, fakat yine de C'nin standart kütüphanesindeki tüm fonksiyonları destekler ve onunla aynı kontrol yapılarını kullanır. C tarafından tanımlanmış tüm hazır veri tiplerini de içerir.

Bu kitapta, C dilini bildiğinizi kabul edeceğiz. Diğer bir deyişle, bu kitapla C++'ı öğrenmeden önce, C'yi kullanabiliyor olmalısınız.

NOT Bu kitapta C++ derleyicinizi kullanarak programları nasıl derleyip çalıştıracağınızı bildiğinizi varsayıyoruz. Eğer bilmiyorsanız derleyicinizin kullanım kılavuzuna bir göz atın. Derleyiciler arasında büyük farklılıklar vardır ve bu nedenle de derleme işleminin nasıl yapılacağını her bir derleyici için ayrı ayrı anlatmamız imkansızdır. Programcılık en iyi yapılarak öğrenilir, bu yüzden kitaptaki örnekleri Internet'ten indirmeli, onları verdığımız sırayla derlemeli ve çalıştırmalısınız.

1.1. Nesneye Yönelik Programlama Nedir?

Nesneye-dayalı programlama, güçlü bir programlama yöntemidir. Başlangıcından bugüne programlamaya çeşitli metodolojiler hakim olmuştur. Programmanın evrim sürecindeki her kritik noktada yeni yaklaşımlar yaratılarak programcının gittikçe karmaşıklaşan programların üstesinden gelmesine yardımcı olunmuştur. İlk programlar, bilgisayarın ön panelindeki düğmeler açıp kapatılarak yapılmıştı. Tabii bu yaklaşım sadece oldukça küçük programlar için geçerliydi. Sonra assembly dili icat edildi ve bu sayede daha uzun programlar yazılabilir. Ardından 1950'lerde ilk üst-düzey dil (FORTRAN) yaratıldı.

Üst-düzey dil kullanımıyla programcılar binlerce satır uzunluğunda programlar yazabiliyorlardı. Ama o tarihteki yaklaşım sınırlı, yetersiz bir yaklaşımıydı. Nispeten kısa programlar için oldukça uygun bir yaklaşım olmasına karşın, uzun programlar yazmaya kalkışıldığında okunması (ve işlenmesi) oldukça güç bir "spaghetti kod" ortaya çıkıyordu. 1960'larda *yapısal-programmanın* icadıyla spaghetti kodlardan kurtulmak mümkün oldu. Yapısal diller Algol ve Pascal idi. Genel olarak C de bir yapısal dildir ve dolayısıyla sizin kullandığınız programlama tarzı da yapısal programlama kabul edilebilir. Yapısal programlama şunlara dayanmaktadır: İyi-tanımlanmış kontrol yapıları, kod blokları, GOTO kullanılmaması (ya da minimum sayıda kullanılması), yineleme yeteneğine sahip ve yerel değişken kullanımına izin veren bağımsız alt-yordamlar. Yapısal programlama esas bakımından, programın, kendini oluşturan bileşenlerine indirgenmesinden ibarettir. Yapısal programlama sayesinde, ortalama bir programcı, 50,000 satır kadar uzunluklarda programlar yazabilir ve kontrol edebilir.

Tüm OOP dillerinin sahip olduğu üç nitelik vardır: Depolama (encapsulation), çok-bağımlilik (polymorphism) ve miras (inheritance). Bu kavramları hep birlikte inceleyelim.

Encapsulation (Depolama)

Depolama, kodu ve kodun işlemiği verileri bir araya getiren ve onları dış etkilerden, yanlış kullanımlardan koruyan mekanizmadır. Nesneye dayalı dillerde kod ve veri, bir "kara kutu" oluşturacak şekilde bir araya getirilir. Kod ve verinin bu şekilde birbirine bağlanmasıyla da *nesne* meydana getirilmiş olur. Diğer bir deyişle nesne depolama yaptığımız yerdir.

Nesnenin içindeki kod, veri veya her ikisi bu nesneye *private* (özel) veya *public* (genel) olabilir. Private kod ve veriler, sadece o nesneye ait bileşenler tarafından bilinabilir ve erişilebilir. Programın nesneye ait olmayan bileşenleri bu private kod ve verilere erişemez. Kod ve verilerimiz public ise, belirli bir nesnenin içinde tanımlansalar da programın diğer par-

çaları tarafından erişilebilirler. Nesnelere ait public elemanları private elemanlara denetlenebilir şekilde erişebilmek amacıyla kullanırız.

Her ne sebeple olursa olsun, nesne, kullanıcı tarafından tanımlanan tipte bir değişkendir. Kodu ve verileri bağlayan nesneyi değişken olarak düşünmemiz size garip görünebilir. Fakat nesnel programlamada durum kesinlikle bu şekildedir. Nesne tipi tanımladığımızda aslında yeni bir veri tipi oluşturuyoruzdur. Bu veri tipi de bileşik bir değişkendir.

Polimorfizm (Çok-Bİçimlilik)

Polimorfizm genel anlamda (Yunanca'da "çok biçimli" demektir) bir adın birbiriyle ilişkili, fakat teknik açıdan farklı iki veya daha fazla amaç için kullanılabilmesi yeteneğidir. OOP kavramı içinde ise polimorfizm, bir adın genel bir iş (action) sınıfını belirlemesine izin verir. Genel bir iş sınıfı içerisinde, yapılacak iş, veri tipi tarafından belirlenir. Örneğin, polimorfizmi pek fazla desteklemeyen C'de, mutlak değer bulma işi üç ayrı fonksiyon adı gerektirir: **abs()**, **labs()** ve **fabs()**. Bu fonksiyonlar sırayla bir integer'in (tamsayının), bir long integer'in (işaretli 32 bitlik tamsayı) ve bir reel sayının mutlak değerini hesaplar ve döndürür. Fakat polimorfizmi destekleyen C++'da bu fonksiyonlar, **abs()** gibi tek bir adla adlandırılabilirler. Bu konuyu bölümün sonlarında tekrar inceleyeceğiz. Fonksiyonu çağrılmak için kullanılan veri tipi,其实teki hangi fonksiyonun çalıştırılacağını belirler. Göreceğiniz gibi, C++'da bir fonksiyon adını birkaç farklı amaç için kullanmak mümkündür. Buna *fonksiyonların aşın yüklenmesi (function overloading)* adı verilir.

Daha genel olarak düşünürsek, polimorfizm, genel bir arabirimin, birbiriyle ilişkili bir takım işler için kullanılması anlamına gelen "bir arabirime birden çok metot" fikri tarafından karakterize edilmiştir. Polimorfizmin avantajı, bir arabirimin *genel bir iş sınıfını* belirlemek için kullanılmasına imkan vererek karmaşıklığı azaltmasıdır. Duruma uygun *isin* seçilmesi derleyicinin görevidir. Siz programcı olarak kendiniz bu seçimi elle yapmak zorunda değilsiniz. Sadece genel arabirimini hatırlamanız ve kullanmanız yeterlidir. Yukarıda verdigimiz örnekte mutlak değer fonksiyonu için bir yerine üç ad kullanılması, bir sayının mutlak değerini bulan genel işi gerçekleştirdiğinden daha karmaşık hale getirir.

Polimorfizm operatörlerde de uygulanabilir. Gerçekte her programlama dili, aritmetik operatörlerle alakalı olduğundan bir parça polimorfizm içerir. Örneğin, C'de + operatörü integer, long integer ve reel tiplerdeki sayıları ve karakterleri toplamak için kullanılır. Bu durumların her birinde derleyici otomatik olarak ne tür bir aritmetik işlem yapacağını bilir. C++'da bunu tanımladığınız diğer veri tiplerine de uygulayabilirsiniz. Polimorfizmin bu çeşidine *operatörlerin aşın yüklenmesi (operator overloading)* adı verilir.

Polimorfizm hakkında hatırlamanız gereken önemli nokta, birbiriyle bağlantılı işlere standart arabirimler oluşturmanıza imkan vererek daha karmaşık durumların üstesinden gelmenizi sağladığıdır.

Inheritance (Miras)

Miras, bir nesnenin, diğer bir nesnenin özelliklerini alabilmesidir. Daha derine inceek olursak, bir nesne, kendisine ait özellikleri de ekleyebileceği bir özellikler kümesini mirasla alabilir. Miras önemlidir, çünkü nesnelerin *hiyerarşik sınıflandırma* (*hierarchical classification*) kavramını desteklemesini sağlar. Pek çok bilgiyi bu hiyerarşik sınıflandırmayı kullanarak kontrol edebiliriz.

Örneğin bir evi düşünelim. Ev, **bina** genel sınıfının bir parçasıdır. **Bina**, daha genel bir sınıf olan **yapının** bir parçasıdır. **Yapı** ise **insan yapımı** diye adlandırdığımız daha genel bir sınıfa aittir. Bu çocuk sınıflar, ebeveynlerine ait tüm özellikleri mirasla alırlar ve onları kendi karakteristiklerine eklerler. Bu sıralı sınıflandırma kullanılmamasaydı, her nesnenin kendisiyle ilgili tüm karakteristikleri açık bir şekilde tanımlaması gereklidir. Fakat miras sayesinde nesnenin hangi genel sınıfa (veya sınıflara) ait olduğunu ve onu diğerlerinden ayıran özellikleri belirleyerek nesneyi tanımlamak mümkündür. Daha sonra da göreceğimiz gibi miras, OOP için çok önemli rol oynayan bir özelliktir.

Örnekler

- Depolama OOP için yeni sayılmas. Depolama özelliği, bir dereceye kadar da olsa C dilinde de mevcuttur. Örneğin bir kütüphane fonksiyonunu kullandığımızda aslında içerisindekileri değiştiremeyeceğimiz ve etkileyemeyeceğimiz (kötü niyetli hareketler hariç) kara kutu şeklinde bir rutini kullanıyoruz. **fopen()** fonksiyonunu düşünelim. Bu fonksiyonu dosya açmak için kullandığımızda çeşitli dahili değişkenler oluşturulur ve bu değişkenler ile ilgili hazırlıklar yapılır. Sizin programınız söz konusu olduğu sürece, bu değişkenler gizli ve erişilemez bir durumdadır. Fakat C++, depolama için daha emniyetli bir yöntem sunmaktadır.
- Gerçek dünyada polimorfizmin örnekleri ile oldukça sık karşılaşabiliriz. Örneğin arabamızın vitesini düşünelim. Vites, ister otomatik olsun ister normal olsun aynı prensiple çalışır. Arabirim (vites) aynıdır ve gerçekte ne tip bir vites mekanizmasının kullanıldığı önemli değildir.
- Özelliklerin mirası ve genel olarak sınıflandırma kavramı bilginin düzenlenmesi açısından önemlidir. Örneğin kereviz **sebze** sınıfının bir üyesidir. **Sebzeler** de bitki sınıfının bir parçasıdır. Bitkiler canlı organizmalardır ve bu böyle devam eder. Hiyerarşik sınıflandırma olmasaydı bilgi sistemleri kurmak mümkün olmazdı.

Alıştırma

- Sınıflandırma ve polimorfizmin günlük hayatımızda ne şekilde rol oynadığını düşünün.

1.2. C++'ın İki Sürümü

Bu yazı hazırlanırken C++, bir değişimin tam ortasındaydı. Kitabın girişinde de söyledik; C++, birkaç yıldır standartlaştırılmaya çalışılıyor. Armaç, programcıların gelecek yüzyıldaki ihtiyaçlarına uygun, kararlı, standart ve zengin özellikli bir dil oluşturmaktı. Sonuç olarak C++'ın gerçekten de iki sürümü var. Bunların ilki Bjarne Stroustrup'un orijinal tasarıma dayanan geleneksel sürümdür. Bu, son on yıldır programcılar tarafından kullanılan sürümdür. İkincisi ise Stroustrup ve ANSI/ISO standartlar komitesi tarafından oluşturulmuş olan yeni Standart C++'dır. C++'ın bu iki sürümü temelde birbirine çok benzemektedir, fakat Standart C++, geleneksel C++'da bulunmayan çeşitli ek özelliklere sahiptir. Bu nedenle Standart C++, geleneksel C++'ın geliştirilmiş şeklidir denebilir.

Bu kitapta, Standart C++'ı öğreniyoruz. Standart sürüm, C++'ın ANSI/ISO standartlar komitesi tarafından tanımlanmış ve tüm modern C++ derleyicileri tarafından kullanılabilen sürümür. Kitabın içerisindeki kodlar, çağdaş kod stilini yansıtır ve bu stili Standart C++ tarafından tavsiye edilen şekilde uygular. Bu sayede öğrenciklerimizi bugün olduğu gibi yarın da kullanabiliriz. Kısaca özetlemek gerekirse, Standart C++, gelecektir. Standart C++, C++'ın daha önceki sürümlerindeki tüm özellikleri de içerdiginden, öğrencilerimizin tamamını tüm C++ ortamlarında kullanabiliriz.

Fakat derleyicimiz eskiyse, programların hepsini kabul etmeyecek. Standartlaştırma süreci içerisinde ANSI/ISO komitesi dile birkaç yeni özellik eklemiştir ve bu özellikler tanımlıkça derleyiciler de geliştiriciler tarafından tamamlanmıştır. Elbette, dile yeni bir özellik eklenmesiyle bu özelliğin ticari derleyicilerde geçerli olması arasında her zaman bir gecikme vardır. Yıllar geçtikçe C++'a yeni özellikler eklendiğinden eski derleyiciler, bundan bir veya birkaçını desteklemeyecektir. Bu önemli bir konu, çünkü yeni eklenen bu özellikler, yazacağımız her programı etkileyecektir, hatta en basit olanlarını bile. Bu özellikleri desteklemeyen eski bir derleyici kullanıyorsanız, endişelenmeyin. Biraz sonra bu sorunu ortadan kaldıracak bir yöntemden bahsedeceğiz.

İki yeni özellikimiz var: Yeni tarz başlıklar (header) ve **namespace** deyimi. İşe bu iki sürümü biraz inceleyerek başlayalım ve C++'a şimdilik el sürmeyeelim. İlk sürüm bize C++ programlarının son zamanlara kadar ne şekilde yazıldığını yansıtıyor. (Bu sürüm eski kod stilini kullanıyor.)

```
/*
  Geleneksel stilde bir C++ programı
*/
#include <iostream.h>
int main()
{
    /* program kodu */
    return 0;
}
```

C++, C üzerine kuruldu, bu nedenle programın iskeleti size pek yabancı gelmeyecek, fakat **#include** deyimine özellikle dikkat edelim. Bu deyimle C++'in I/O sistemini des-

teklemesini sağlayan **iostream.h** dosyasını dahil ediyoruz. (C için **stdio.h** neyse, C++ için de bu dosya odur.) Bu da program iskeletinin modern stili kullanan ikinci sürümü:

```
/*
 * Yeni başlık stillerini ve namespace'ı kullanan modern bir C++ programı.
 */
#include <iostream>
using namespace std;

int main()
{
    /* program kodu */
    return 0;
}
```

İlk açıklamadan sonra gelen ilk iki satırı dikkat edelim, işte burası değişikliğin meydana geldiği yer. İlk yenilik **#include** deyiminde **iostream** adından sonra **.h** olmaması ve ikinci yenilik de bir sonraki satırda **namespace** denilmesi. Bu yenilikleri daha sonra ayrıntılarıyla ele alacağız, fakat yine de kısa bir giriş yapmadan geçmeyeceğim.

C++'ın Yeni Başlıklarını

C'den cdindiğimiz tecrübelerden de biliyoruz ki, bir kütüphane fonksiyonunu programımızda kullandığysak, onun başlık dosyasını programımıza dahil etmemiz gereklidir. Bu da **#include** deyimi ile yapılır. Örneğin C'de I/O fonksiyonları için gerekli **stdio.h** başlık dosyasını şu şekilde programa dahil edilmesi gereklidir:

```
#include <stdio.h>
```

Burada **stdio.h** I/O fonksiyonları tarafından kullanılan dosyanın adıdır ve yukarıdaki deyim ile dosyanın programa dahil edilmesi sağlanır. Burada anlaşılması gereken ince nokta, **#include** deyiminin programa *bir dosya dahil ettiğidir*.

C++, ilk oluşturulduğu bir iki yıl içerisinde C ile aynı başlık stilini kullandı. Hatta Standart C++, sizin oluşturduğunuz başlık dosyaları için ve eski sürümlere uyumluluk sağlanması açısından hala C stili başlıklarını destekler. Fakat Standart C++, kendi kütüphanesi tarafından kullanılan yeni bir başlık stili getirmiştir. Bu yeni tarz başlıklar, dosya adlarını *değil*, dosyalara derleyici tarafından bağlanabilmesi mümkün standart tanımlayıcıları belirler. Yeni C++ başlık stilleri, C++ kütüphanesi için gerekli, uygun prototiplerin ve tanımlarının deklare edilmiş olduğunu gösteren soyut kavramlardır.

Yeni başlık stilleri, dosya adları değildir, bu nedenle bu stillerde **.h** uzantısı yoktur. Küçük tür büyük tür işaretleri arasında bulunan başlık adından meydana gelir. Standart C++ tarafından desteklenen yeni başlık stillerini inceleyelim.

```
<iostream>
<fstream>
<vector>
<string>
```

Bu yeni stilde de başlıklar, **#include** deyimi ile programa dahil edilir. Aradaki tek fark, yeni stilde dosya adlarının gösterilmesinin gerekmemesidir.

C++, C'nin fonksiyon kütüphanesinin tamamını kapsamaktadır, bu yüzden C kütüphanesinin başlık dosyalarını da destekler ve **stdio.h** ve **ctype.h** gibi başlık dosyaları hala geçerlidir. Fakat, bunların yanı sıra yeni başlık stillerimizi de kullanabiliriz. C++'da, C'nin standart başlıklarındaki dosya adlarına bir **c** öneki eklenmiştir ve **.h** düşmüştür. Örneğin **math.h** için C++'in yeni tarz başlığı **<cmath>** şeklindedir. **string.h** ise **<cstring>** şeklärini alır. C kütüphane fonksiyonları için C stilinde başlık dosyalarını kullanabiliyoruz, fakat bu yaklaşım Standart C++ tarafından pek hoş karşılanmaz, yanı tavsiye edilmez. Bu sebeple, burada tüm **#include** deyimlerinde yeni başlıkları kullanacağız. Eğer derleyiciniz C fonksiyon kütüphanesi için yeni başlık stilini desteklemiyorsa eski stilde başlıkları kullanın.

Bu başlıklar C++'a çok kısa bir süre önce eklendi, bu nedenle karşılaşığınız pek çok derleyici onları desteklemeyecek. Bu derleyicilerde C stili başlıklarını kullanmanız gerekecek. Örneğin I/O başlığı şu şekilde dahil edeceksiniz.

```
#include <iostream.h>
```

Bu şekilde **iostream.h** dosyası programa dahil edilmiş olur. Genelde eski stil başlıklar, onlara karşılık gelen yeni başlıklarla aynı ada sahiptirler, tek fark onlarda **.h** uzantısının olmasıdır.

Daha önce de söyledğimiz gibi, tüm C++ derleyicileri eski başlıkları desteklerler. Fakat bu eski başlıklar sonuçta eskidirler ve bunları yeni programlarda kullanmanız tavsiye etmiyoruz. Bu nedenle eski başlıkları kitapta kullanmadık.

HATIRLATMA Bu eski başlıklarla mevcut C++ kodlarında karşılaşabileceğiniz, fakat bu başlıklar artık pek fazla kullanılmamaktadır.

Namespace'ler

Programımıza yeni bir başlık eklediğimizde bu başlığın içeriği **std namespace**'de saklanır. **Namespace'ler** tanımlama için ayrılmış alanlardır. Namespace'i kullanmamızın amacı ad çakışmalarını engellemek için tanımlayıcı adlarını yerleştirmektir. Kütüphane fonksiyonlarının ve diğer elemanların adları eski stilde (tipki C'de olduğu gibi) global namespace'lerin içine yerleştirilirdi. Fakat yeni başlıkların içeriği **std namespace**'in içerisine yerleştirilir. Daha sonra namespace konusunu ayrıntılarıyla inceleyeceğiz. Bu konuda pek endişelenmenize gerek yok, çünkü **std namespace**'i (örn. **std**'yi global namespace'lere yerleştirmek için)

```
using namespace std;
```

deyimini kullanabilirsiniz. Bu deyim derlendikten sonra eski veya yeni başlıklarla çalışmış olmanızın bir önemi kalmaz.

Eski Derleyicilerle Çalışmak

Daha önce konuştuğumuz gibi, hem namespace'ler hem de yeni stildeki başlıklar C++ diline yeni eklenmiştir. Yeni derleyicilerin tamamının bu özellikleri desteklediğini kabul etsek de daha eski derleyiciler bunları desteklemeyebilir. Eski derleyicilerden birine sahipseniz verdigimiz örnek programları derlemeye çalıştığımızda ilk iki satırda bir veya birkaç hata çıkacaktır. Eğer böyle bir durumla karşılaşırsanız, bunun kolay bir çözüm yolu var: eski tipte başlık kullanın ve **namespace** deyiminini silin. Yani, sadece

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

deyimini

```
#include <iostream.h>
```

deyimi ile değiştirmeniz gerekiyor. Bu değişikliği yaparak modern bir programı eski stile dönüştürmüs oluruz. Eski başlıklarda içerik global namespace'lere yerleştirildiğinden ayrıca bir **namespace** deyiminine gerek yoktur. Dikkat edilmesi gereken diğer bir nokta daha var: Önümüzdeki birkaç yıl içerisinde pek çok C++ programında eski başlıkların kullanıldığını ve **namespace** deyiminin kullanılmadığını göreceksiniz. C++ derleyiciniz bu programları gayet güzel derleyecektir. Fakat yeni programlar için yeni stili kullanmalısınız, çünkü bu, Standart C++'ın tavsiye ettiği program stilidir. Eski stil uzun bir süre daha desteklenecektir, fakat yine de teknik olarak kabul edilemezler.

Alıştırma

1. Yukarıda verdigimiz yeni program iskeletini derlemeyi deneyin. Bu program hiçbir şey yapmıyor, fakat onu derleyerek derleyicinizin modern C++'ın yazım kurallarını destekleyip desteklemediğini öğrenebilirsınız. Eğer yeni stil başlıkları ve **namespace** deyiminin derleyiciniz kabul etmezse, yukarıda anlattığımız gibi modern başlıkları eski stil başlıklarla değiştirin. Unutmayın, eğer derleyiciniz yeni kod stilini kabul etmezse bu değişikliği kitaptaki her program için yapmanız gereklidir.

1.3. C++'IN I/O Konsolu

C++, C'nin gelişmiş şeklidir, bu yüzden C'deki tüm elemanlar, C++'da da bulunmaktadır. Her C programı aynı zamanda bir C++ programıdır da denebilir. (Aslında bu kurala uymayan bazı küçük durumlar vardır, bu durumları daha sonra ele alacağız. Sonuç olarak, tamamen C programı gibi görünen C++ programları yazmak mümkündür. Bu düşüncede her ne kadar bir yanlışlık yoksa da bu durum C++'ın avantajlarından yararlanmayacağınız anlamına gelmez. C++'dan maksimum seviyede faydalananmak için C++ stiline programlar yazmalısınız. Bu da C++'a özgü kod yazma stilini ve Özellikleri kullanmak anlamına gelmektedir.

C++ programcılar tarafından en çok kullanılan özellik belki de I/O konsoludur. Siz hala `printf()` ve `scanf()` gibi fonksiyonları kullanırken C++, bu tip I/O işlemleri için daha iyi ve yeni bir yol önermektedir. C++'da I/O, I/O fonksiyonları yerine I/O operatörleri kullanılarak gerçekleştirilir. Çıkış operatörü, `<<` ve giriş operatörü, `>>` şeklindedir. Bunlar C'de sırasıyla sağa ve sola kaydırma operatörleridir. C++'da bu orijinal anımlarını (sağ ve sol Shift) muhafaza eden bu işaretler giriş ve çıkış işlemlerini gerçekleştirmeye rolünü üstlenirler. Bu C++ deyimi üzerinde biraz düşünelim :

```
cout << "Bu katar ekranas basılacak.\n";
```

Bu deyim, katarın ekranda gösterilmesini sağlar. `cout`, C programı çalıştırıldığında otomatik olarak konsola bağlanan önceden tanımlanmış bir akımdır. C'nin `stdout`'una benzemektedir. C'de olduğu gibi, C++ konsol I/O'su da tekrar yönlendirilebilir, fakat tartışmamızın sonuna kadar konsolu kullandığımızı varsayacağız.

`<<` çıkış operatörünü kullanarak C++'ın temel tiplerinden herhangi birini çıkışa göndermek mümkündür. Örneğin bu deyim 100.99 değerini verir :

```
cout << 100.99;
```

Konsola çıkış göndermek için `<<` operatörünün genel kullanımı şu şekilde yapılır :

```
cout << ifade;
```

Burada `ifade`, bir çıkış ifadesi veya geçerli herhangi bir C++ ifadesi olabilir. Klavyeden giriş yapılması için ise `>>` giriş operatörünü kullanıyoruz. Aşağıdaki örnekte `num` değişkenine bir tam sayı değeri koymuyoruz:

```
int num;
cin >> num;
```

`num`'dan önce `&` işaretini kullanmadığımıza dikkat edin. Biliyoruz ki, C'nin `scanf()` fonksiyonunu değer girişi için kullandığımızda, değişkenlerin adreslerini fonksiyona göndermemiz gereklidir, böylece kullanıcı tarafından girilen değerleri alabiliriz. Bu durum, C++'ın giriş operatörü için böyle değildir. (Bunun nedeni C++'ı daha yakından tanıdığımızda açıklığı kavuşturacaktır.) Klavyeden değer okumak için `>>` operatörünü şu şekilde kullanıyoruz :

```
cin >> değişken;
```

NOT `<<` ve `>>` operatörlerine asıl görevleri dışında ek olarak verdığımız bu görevler *operatörün arşiv yükleyen mesinin ömektedir*

C++'ın I/O operatörlerini kullanmak için programınıza `<iostream>` başlığını eklememiz gereklidir. Bunun, C++'ın standart başlıklarından biri olduğunu ve C++ derleyiciniz tarafından destekleneceğini daha önce belirtmiştim.

Örnekler

- Bu program, ekrana bir katar, iki tamsayı ve bir double reel sayı göndermektedir:

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

int main()
{
    int i, j;
    double d;

    i = 10;
    j = 20;
    d = 99.101;
    cout << "İşte size bazı değerler: ";
    cout << i;
    cout << ' ';
    cout << j;
    cout << ' ';
    cout << d;

    return 0;
}
```

Program şu sonucu verecektir:

```
İşte size bazı değerler: 10 20 99.101
```

HATIRLATMA Eğer derleyiciniz eskiye yeni stil başlıktan, bu ve diğer programlarda kullanılan namespace deyimlerini kabul etmeyebilir. Eğer durum böyleyse bir önceki konuda anlatıldığı gibi eski stil kod kullanın.

- Tek bir I/O ifadesinde birden fazla çıkış vermemiz mümkün değildir. Örneğin, Örnek 1'de verilen programdaki I/O deyimlerini burada tek satırda veriyoruz:

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

int main()
{
    int i, j;
    double d;

    i = 10;
    j = 20;
    d = 99.101;

    cout << "İşte size bazı değerler: ";
    cout << i << ' ' << j << ' ' << d;

    return 0;
}
```

Şu satırda,

```
cout << i << ' ' << j << ' ' << d;
```

birden fazla çıkış gönderme işlemini tek bir ifadede yapıyoruz. Tek bir deyimde istedigimiz kadar çıkış gönderebiliriz. Eğer bu kafanızı karıştırdıysa, << çıkış operatörünün diğer C++ operatörleri gibi davranışını ve uzun ifadelerde yer alabileceğini unutmamanız yeterli olacaktır.

Gerekliyorsa maddeler arasında boşluk bırakmayı unutmayın. Eğer boşluk bırakmazsanız gönderdiğiniz çıkışlar ekranda bitişik gözükecektir.

- Bu program, kullanıcıdan bir tamsayı değeri girmesini istemektedir:

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

int main()
{
    int i;

    cout << "Bir değer girin: ";
    cin >> i;
    cout << "İşte size bazı değerler: " << i << "\n";

    return 0;
}
```

Programı çalıştırduğumda ekranda şu sonuçları göreceğiz :

```
Bir değer girin: 100
İşte size bazı değerler: 100
```

Burada kullanıcı tarafından girilen değer i'nin içine konmuştur.

- Aşağıdaki program kullanıcının bir tamsayı, bir reel sayı ve bir katar girmesini istemektedir. Sonra bu üç veriyi okumak için bir giriş deyimi kullanmaktadır.

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

int main()
{
    int i;
    float f;
    char s[80];

    cout << "Bir tamsayı, bir reel sayı, bir de katar girin: ";
    cin >> i >> f >> s;
    cout << "Verileriniz: ";
    cout << i << ' ' << f << ' ' << s;

    return 0;
}
```

Bu örnekte olduğu gibi tek bir deyimde istediğimiz sayıda giriş yapabiliriz. Tík C'deki gibi veriler çeşitli şekillerde (boşluklar, sekmeler ve alt satır geçme şeklinde) birbirinden ayrılmalıdır.

Katarlar okunurken okuma ilk boşlukta durdurulur. Örneğin yukarıdaki programa aşağıdaki verileri girerseniz:

```
10 100.12 Bu bir deneme
```

program şunu sonucu verecektir :

```
10 100.12 Bu
```

Katar tamamlanmamıştır, çünkü katarın okunması **Bu** kelimesinden sonraki boşlukta kesilmiştir. Katarın kalan kısmı tampon bellekte bırakılarak bir sonraki okuma için bekler. (Bu, **scanf()**'ı **%s** formатıyla kullanılarak yapılan katar girişi işlemine benzer bir durumdur.)

- Normalde **>>** operatörünü gördüğünüzde, tüm giriş, satır halinde tampon belleğe konur. Bunun anlamı, siz ENTER tuşuna basana kadar hiçbir bilginin sizin C++ programınıza geçmeyeceğidir. (C'de **scanf()** fonksiyonu, satır halinde belleğe alınır, bu yüzden bu giriş şekli sizin için yeni olmamalıdır.) Girişin satır halinde belleğe konulmasının ne sağladığını anlamak için şu programı deneyin:

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

int main()
{
    char ch;

    cout << "Tuşlara basın, çıkış için x.\n";
    do {
        cout << ": ";
        cin >> ch;
    } while (ch != 'x');

    return 0;
}
```

Bu programı denedığınızda bastığınız her tuştan sonra ona karşılık gelen karakterin programa gönderilmesi için ENTER'a basmanız gerekecek.

Alıştırmalar

- Çalışan bir kişinin toplam mesai saatinin ve maaşının girildiği bir program yazın. Sonra çalışanın net maaşını ekranda gösterin. (Saat sayısını ve maaşı kullanıcı ekrandan gırsın.)
- Fitiince çeviren bir program yazın. Kullanıcıya kaç fit olduğunu sorun ve bunun inç cinsinden eşdeğerini ekranda gösterin. Programınız, bu işleme kullanıcı fit için 0 girene kadar devam etsin.

3. Size bir C programı veriyoruz. Onu C++ stili I/O deyimlerini kullanacak şekilde bir daha yazın.

```
/* Bu C programını C++ stiline dönüştürün.  
Bu program en küçük ortak paydayı hesaplar.  
*/  
#include <stdio.h>  
  
int main(void)  
{  
    int a, b, d, min;  
  
    printf("İki sayı girin: ");  
    scanf("%d%d", &a, &b);  
    min = a > b ? b : a;  
    for(d = 2; d<min; d++)  
        if((a%d)==0) && ((b%d)==0)) break;  
    if(d==min) {  
        printf("Ortak payda yok \n");  
        return 0;  
    }  
    printf("En küçük ortak payda %d\n", d);  
    return 0;  
}
```

1.4. C++'da Açıklamalar

C++'da programımıza iki şekilde açıklama ekleyebiliyoruz. Bunlardan ilki Standart C'dekine benzer açıklama mekanizmasının kullanılmasıdır. Yani açıklamanızı `/*` ile başlatıp `*/` ile bitirebilirsiniz. Tıpkı C'de olduğu gibi, C++'da da bu tür açıklamalar başka bir açıklamaının içinde bulunamazlar.

C++'da programımıza açıklama ekleyebileceğiniz ikinci yol ise *tek satır* açıklamalarını kullanmaktadır. Tek satır açıklaması `//` ile başlar ve satırın sonunda biter. Tek satır açıklamaları satırın fiziksel sonu haricinde (yani satır başı ve satır ekleme operatörlerinin kombinasyonu) açıklama bitisi operatörü kullanmaz.

Genelde C++ programcılar, uzun açıklamalar için C yöntemini ve daha kısa olanlar için ise tek satırlık açıklamaları kullanırlar.

Örnekler

1. Hem C, hem de C++ stili açıklamaları içeren bir program:

```
/*  
 * Bu C tipinde bir açıklama.  
 *  
 * Bu program bir tamsayının  
 * tek mi çift mi olduğuna karar verir.  
  
#include <iostream>  
using namespace std;
```

```

int main()
{
    int num; // bu C++'in tek satırlık bir açıklamasıdır

    // sayıyı oku
    cout << "Test edilecek sayıyı girin: ";
    cin >> num;

    // tek mi çift mi bak
    if((num%2)==0) cout << "Bu bir çift sayıdır \n";
    else cout << "Bu bir tek sayıdır \n";

    return 0;
}

```

2. Çok satırlı açıklamalar bir diğerinin içinde bulunamazlarsa da, tek satırlık bir açıklama çok satırlı bir açıklamanın içine koymak mümkündür. Örneğin, aşağıdaki tamamen geçerlidir:

```

/* Bu çok satırlı bir açıklamadır.
   İçinde // tek satırlık bir açıklama vardır.
   Bu da çok satırlı açıklamanın sonudur.
*/

```

Tek satırlı açıklamaların çok satırı olanların içinde bulunabilmesi, kodun sonradan incelenmesi için satırların “açıklanmasını” kolay hale getirecektir.

Aliştırmalar

1. Bu açıklamanın (C++ stilinde tek satırlık komutun içinde bulunan C benzeri açıklama) geçerli olup olmadığına karar verin:

```
// Bu, açıklama yapmak için /* garip bir yoldur */
```

2. Konu 1.3'deki alıştırmalara açıklamalar ekleyin.

1.5. Sınıflara Giriş

C++'in en önemli özelliği belki de *sınıflardır*. Sınıf, nesneleri oluşturmak için kullanılan mekanizmadır. Pek çok C++ özelliğinin kalbidir. Sınıflar konusunuince detaylarıyla ele alacağız, fakat çok önemli bir konu olduğu için burada kısaca bir giriş yapılmasını gereklidir.

Sınıf, **class** kelimesi kullanılarak deklare edilir. **class**'ların bildirimini yapılarındaki benzer bir şekilde yapılır. Bu bildirimin genel formatı şu şekildedir:

```

class sınıf-adı {
    // private (özel) fonksiyonlar ve değişkenler
public:
    // public (genel) fonksiyonlar ve değişkenler
} nesne-listesi;

```

Bir sınıf bildiriminde *nesne-listesini* isterseniz verirsiniz. Tipki yapılar için olduğu gibi, sınıfa ait nesneleri de gerekirse daha sonra deklare edebilirsiniz. Teknik açıdan *sınıf-adi*'nın yazımı da isteğinize bağlıdır, fakat pratikte her zaman gereklidir. Bunun nedeni *sınıf-adi*'nın sınıflara ait nesneleri deklare etmekte kullanılan yeni tipte bir ad haline gelmesidir.

Sınıf bildiriminin içinde deklare edilmiş olan fonksiyonlar ve değişkenler sınıfın *üyeleri* olarak bilinir. Normalde sınıfın içerisinde deklare edilmiş tüm fonksiyonlar ve değişkenler sınıfa özeldir (*private*). Onların sınıfa özel olmaları sadece sınıfın diğer üyeleri tarafından erişilebilmeleri anlamına gelir. *Public* sınıf üyelerini deklare etmek için ise *public* anahtar kelimesi ve onu takip eden iki nokta üst üste operatörü kullanılır. *public* belirleyicisinden sonra deklare edilmiş tüm fonksiyonlar ve değişkenler, hem sınıfın diğer üyeleri tarafından hem de o sınıfın dahil edildiği program kısımlarından erişilebilirler. Basit bir sınıf bildiriği:

```
class myclass {
    // myclass'a özel (private)
    int a;
public:
    void set_a(int num);
    int get_a();
};
```

Bu sınıfın *a* adında bir *private* (özel) değişkeni ve *set_a()* ve *get_a()* olmak üzere iki tane *public* (genel) fonksiyonu vardır. Fonksiyonların, sınıf içerisinde, kendi prototip formları kullanılarak deklare edildiğine dikkat edin. Bir sınıfa ait olarak deklare edilen fonksiyonlara *üye fonksiyonları* denilir.

a *private* (özel) olduğundan *myclass*'ın dışındaki herhangi bir koddan ona erişilemez. Fakat *set_a()* ve *get_a()*, *myclass*'ın elemanı olduklarından *a*'ya erişebilirler. Üstelik *get_a()* ve *set_a()*, *myclass*'ın *public* üyeleri olarak deklare edilmişlerdir ve programın *myclass*'ı içeren herhangi bir kısmından kendilerine erişilebilir.

get_a() ve *set_a()* fonksiyonları *myclass* tarafından deklare edilmişlerdir, fakat henüz tanımlanmamışlardır. Üye fonksiyonları tanımlamak için, sınıfın tip adımı fonksiyon adıyla bağlamanız gereklidir. Bu işlemi, sınıf adı ve iki nokta üst üste işaretlerinin ardından fonksiyonun adını yazarak yapabilirsiniz⁷. Yan yana yazılmış olan iki tane iki nokta üst üste işaretine *kapsam çözümleme operatörü* adı verilir. *set_a()* ve *get_a()* üye fonksiyonlarının nasıl tanımlandığını inceleyelim:

```
void myclass::set_a(int num)
{
    a = num;
}

int myclass::get_a()
{
    return a;
}
```

Hem `set_a()`'nın hem de `get_a()`'nın `myclass`'ın private değişkeni `a`'ya erişim hakkı olduğuna dikkat edelim. Çünkü `set_a()` ve `get_a()` `myclass`'ın üyeleriidir, onun private (özel) verilerine doğrudan erişebilirler.

Sınıfa ait fonksiyonların genel formatı şu şekildedir :

```
döndürme-tipi sınıf-adi::fonksiyon-adi(parametre-listesi)
{
    // Fonksiyonun yordamı
}
```

Burada `sınıf-adi` fonksiyonların ait olduğu sınıfın adıdır.

`myclass`'ın bildiriminde `myclass` tipinin herhangi bir nesnesi tanımlanmadı, sadece nesne gerçekten deklare edildiğinde oluşturulacak nesnenin tipi tanımlandı. Nesne oluşturmak için sınıf adını bir tip belirleyicisi olarak kullanırız. Örneğin, şu satırda `myclass` tipinde iki nesne deklare ediyoruz:

```
myclass ob1, ob2; // bunlar myclass tipinde nesnelerdir
```

HATIRLATMA Sınıf bildirimleri yeni tipleri tanımlayan soyut kavramlardır. Bu tipte bir nesnenin ne şekilde olacağına karar verirler. Nesne bildirimleri, nesnenin tipinde fiziksel bir büyülük oluştururlar. Yani, nesne bellekte yer kaplar, fakat bir tip tanımlaması, koplamaz.

Sınıfa ait bir nesne oluşturduğumuzda programımızda, tipki yapıya ait üyelere eriştiğimiz gibi, nokta (dot, periyot) operatörünü kullanarak bu nesnelere de erişebiliriz. Nesne bildiriminin üstte yazılı olduğunu varsayırsak, aşağıdaki deyimde `ob1` ve `ob2` nesneleri için `set_a()` fonksiyonunu çağrıyoruz:

```
ob1.set_a(10); // ob1'in a'sını 10 yapar
ob2.set_a(99); // ob2'nin a'sını 99 yapar
```

Açıklamalarda da belirttiğimiz gibi bu deyimler `a`'nın `ob1` kopyasına 10 ve `ob2` kopyasına da 99 koyar. Her nesne, sınıf içerisinde deklare edilmiş tüm verilerin kendisine ait kopyasını içerir. Bu da, `ob1`'in `a`'sı `ob2`'ye bağlanan `a`'dan ayrı ve farklıdır anlamına gelir.

HATIRLATMA Sınıfı ait tüm nesneler, sınıf içerisinde deklare edilen değişkenlerin birer kopyasına sahiptir.

Örnekler

- İlk örneğimiz olan bu programda bahsettiğimiz `myclass` sınıfını veriyoruz. Burada `a`'nın değeri `ob1` ve `ob2` için veriliyor ve sonra her nesne için `a`'nın değeri gösteriliyor:

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

class myclass {
    // myclass'a private
```

```

    int a;
public:
    void set_a(int num);
    int get_a();
};

void myclass::set_a(int num)
{
    a = num;
}

int myclass::get_a()
{
    return a;
}

int main()
{
    myclass ob1, ob2;
    ob1.set_a(10);
    ob2.set_a(99);
    cout << ob1.get_a() << "\n";
    cout << ob2.get_a() << "\n";
    return 0;
}

```

Sizin de tahmin edeceğiniz gibi, bu program ekranda 10 ve 99 değerlerini gösterir.

- Yukarıdaki örnekte **myclass** içinde **a** **private**'dır (özel). Bu, ona doğrudan sadece **myclass**'in üye fonksiyonlarının erişebileceği anlamına gelir. (Bu, **get_a()** genel fonksiyonuna gerek duyulmasının sebeplerinden biridir.) Sınıfın **private** üyelerine, programın sınıfı ait olmayan bir kısmından erişmeyi denerseniz, derleme sırasında hata çıkacaktır. Örneğin **myclass**'in yukarıdaki örnekte görüldüğü gibi tanımlanmış olduğunu kabul edersek aşağıdaki **main()** fonksiyonu hataya neden olacaktır:

```

// Bu program hatalıdır.
#include <iostream>
using namespace std;

int main()
{
    myclass ob1, ob2;

    ob1.a = 10; // HATA! Üye olmayan fonksiyonlar
    ob2.a = 99; // private üyeye erişemez.

    cout << ob1.get_a() << "\n";
    cout << ob2.get_a() << "\n";

    return 0;
}

```

- Public fonksiyon olabildiği gibi public değişken de olabilir. Örneğin **a**'yı, **myclass**'in public bölümünde declare etmişsek, **a**'ya programın herhangi bir yerinden erişebiliriz:

```

#include <iostream>
using namespace std;

```

```

class myclass {
public:
    // şimdi a public (genel)
    int a;
    // ve set_a() veya get_a()'ya gerek yok
};

int main()
{
    myclass obj, obj2;

    // burada a'ya doğrudan erişiliyor
    obj.a = 10;
    obj2.a = 99;

    cout << obj.a << "\n";
    cout << obj2.a << "\n";

    return 0;
}

```

Bu örnekte a myclass'a ait public bir üye olarak deklare edildiğinden, kendisine doğrudan main()'den erişilebilir. Nokta operatörünün a'ya erişmek için nasıl kullanıldığına dikkat edelim. Genelde fonksiyonu kendi sınıfının dışında bir yerden çağırırken veya bu şekilde bir değişkene erişirken, hangi nesneye ait üyeye gönderme yaptığınızı tam olarak belirtmek için nesnenin adı, nokta operatörü ve üye adının verilmesi gereklidir.

- Nesneleri kullanmanın bize neler sağladığım görmek için daha pratik bir örneğe gelim. Bu programda, karakterleri saklamak için kullanılacak stack (yığın) adında bir sınıf oluşturuyoruz:

```

#include <iostream>
using namespace std;

#define SIZE 10

// Karakterler için bir stack sınıfı deklare eder
class stack {
    char stck[SIZE]; // stack için yer açar
    int tos; // index of top of stack
public:
    void init(); // stack için gerekli hazırlıkları yapar
    void push(char ch); // stack'a karakter iter
    char pop(); // stack'dan karakter çeker
};

// stack için gerekli hazırlıkları yapar
void stack::init()
{
    tos = 0;
}

// Bir karakter iter
void stack::push(char ch)
{
    if(tos==SIZE) {

```

```
cout << "Yığın dolu ";
return;
}
stack[tos] = ch;
tos++;
}

// Bir karakter çeker
char stack::pop()
{
    if(tos==0) {
        cout << "Yığın boş";
        return 0; // Yığın boşsa sıfır döndürür
    }

    tos--;
    return stack[tos];
}

int main()
{
    stack s1, s2; // iki yığın oluşturur
    int i;

    // Yığınları hazır hale getirir
    s1.init();
    s2.init();
    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "s1'i çek: " << s1.pop() << "\n";
    for(i=0; i<3; i++) cout << "s2'i çek: " << s2.pop() << "\n";

    return 0;
}
```

Bu program, ekrana aşağıdaki çıkışı gönderir :

```
s1'i çek: c
s1'i çek: b
s1'i çek: a
s2'i çek: z
s1'i çek: y
s1'i çek: x
```

Şimdi bu programa daha yakından göz gezdirelim. **stack** sınıfı iki private (özel) değişken içeriyor: **stck** ve **tos**. **stck** dizisi gerçekte **stack**'a itilen karakterleri saklar ve **tos** ise **stack**'ın indeksidir. Public (genel) **stack** fonksiyonları, sırasıyla **stack** için gerekli hazırlıkları yapan, ona bir değer iten ve ondan bir değer çeken **init()**, **push()** ve **pop()** fonksiyonlarıdır.

`main()`'in içerisinde `s1` ve `s2` adlarında iki yiğin oluşturuyoruz ve her yiğine üç karakter itiyoruz. Her stack nesnesi diğerinden farklıdır. Yani, `s1`'e itilen karakterler *hiçbir şekilde* `s2`'e itilen karakterleri etkilemezler. Her nesne kendi `stack` ve `top` yollarını içerir. Bu kavram nesneleri anlamamız için önemlidir. Sınıfa ait tüm nesneler fonksiyonları paylaşıyorsa da her nesne *kendisine ait verileri* oluşturur ve saklar.

Aliştırmalar

1. Eğer henüz yapmadıysanız, bu konuda verilen örnek programları derleyin ve çalıştırın.
2. `card` adında kütüphane kartı verilerini saklayan bir sınıf oluşturun. Bu sınıf kitap adı, yazar ve elde bulunan kitap adeti verilerini saklasın. Kitap adı ve yazarı katar olarak, eldeki adedi de tamsayı olarak saklayın. Kitap bilgilerini saklamak için `store()` adında ve bilgileri göstermek için ise `show()` adında birer public (genel) fonksiyon kullanın. Sınıfı göstermek için kısa bir `main()` fonksiyonu kullanın.
3. Tamsayıların saklanması için `queue` (kuyruk) adında bir sınıf oluşturun. Kuyruğun büyüklüğünü 100 tamsayı uzunlığında yapın. Bu işlemi gösteren kısa bir `main()` fonksiyonu oluşturun.

1.6. C ve C++ Arasındaki Bazı Farklılıklar

Her ne kadar C++, C'nin gelişmiş şekli de olsa aralarında bazı küçük farklar bulunmaktadır ve bu farklardan birkaçının öğrenilmesi gerekmektedir. Şimdi isterseniz biraz bu farklılıklar üzerinde duralım.

C'de fonksiyonun parametresi yoksa, prototipin fonksiyon parametre listesinde `void` kelimesi bulunur. Örneğin C'de `f1()` adlı bir fonksiyonun parametresi yoksa (ve geriye bir `char` gönderiyorsa) bu fonksiyonun prototipi şu şekilde olur:

```
char f1(void);
```

Fakat C++'da `void`'un yazılması isteğimize bağlıdır. Sonuç olarak C++'da `f1()`'in prototipi genelde şu şekildedir:

```
char f1();
```

C++, parametre listesi boş da olsa onu belirler ve bu yönyle C'den farklıdır. Eğer üstteki prototip bir C programında bulunsaydı fonksiyona parametreler *hakkında hiçbir şey söylememiş* olacaktı. C++'da ise bu, fonksiyonun *parametresi olmadığı* anlamına gelir. Önceki örneklerde boş parametre listelerini deklare etmek için `void` kullanılmamasının nedeni budur. (Boş parametre listelerinin deklare edilmesi için `void`'un kullanımı doğru değildir, bu sadece bir fazlalıktır. C++ programclarının çoğu, verimliliği dini bir kural gibi kabul ettiklerinden hemen hiçbir zaman `void`'un bu şekilde kullanıldığını görmeyeceksiniz.) Unutmayın C++'da bu iki bildirim birbirine eşittir.

```
char f1();
char f1(void);
```

C ve C++ arasındaki ince farkların bir diğeri de şudur: Bir C++ programında tüm fonksiyonların prototipi bulunmak zorundadır. Unutmayın C'de prototiplerin kullanımı önerilir, fakat bunların kullanımı teknik olarak isteğinize bağlıdır. C++'da ise prototipler gereklidirler. Diğer konunun örneklerinde de olduğu gibi sınıfı ait fonksiyonun prototipi aynı zamanda genel bir prototip olarak da rol oynar ve bu prototipten başka prototipe gerek kalmaz.

C ve C++ arasındaki üçüncü fark da şudur: Eğer fonksiyon C++'da, geriye değer gönderecek şekilde deklare edilmişse, bu fonksiyon, bu değeri göndermek zorundadır. Yani, eğer fonksiyon **void** haricinde bir şey geri döndürüyorsa bu fonksiyonun içindeki herhangi bir **return** deyimi, bir değer içermelidir. C'de ise **void** içermeyen fonksiyonların aslında geriye değer döndürmesi şart değildir. Eğer fonksiyon herhangi bir değer döndürmüyorsa saçma bir değer "döndürülür".

C'de fonksiyonun döndüreceği sonucun tipini tam olarak belirlemezseniz bu tipin tamsayı olduğu varsayıılır. C++, "tamsayı kabulü" kuralını bozmaktadır. Bu yüzden tüm fonksiyonların return tipini tam olarak deklare etmeniz gereklidir.

C++ programlarında sıkça karşılaşacağınız diğer bir fark da, yerel değişkenlerin deklare edilebileceği yerle ilgilidir. C'de yerel değişkenler sadece blokların başında herhangi bir "iş" deyiminden önce deklare edilebilir. C++'da yerel değişkenler her yerde deklare edilebilirler. Bu yöntemin avantajlarından biri, yerel değişkenlerin ilk kullandıkları yere yakın yerde tanımlanabilmesi ve böylelikle istenmeyen yan etkilerden kurtulunmasıdır.

Son değişiklik ise C++'ın **bool** veri tipini tanımıştır. Bu veri tipi, Boolean (örn., doğru/yanlış) değerleri saklamak için kullanılır. C++ aynı zamanda **true** (doğru), **false** (yanlış) anahtar kelimelerini de tanır. **bool** tipi sadece bu iki değeri alabilir. C++'da karşılaştırma ve mantık işlemlerinin sonucu **bool** tipinde bir değerdir ve tüm koşullu deyimler bir **bool** değeri almaktadır. İlk başta bu durum çok büyük bir değişiklikmiş gibi görünebilir, fakat durum böyle değildir. Hatta bu,其实te gözle bile görünmeyecek kadar küçük bir faktır, çünkü bildiğiniz gibi C'de **true** sıfır olmayan herhangi bir değerdir ve **false** 0'dır. Bu C++'da da geçerlidir, bir Boolean ifade kullanıldığında sıfır olmayan herhangi bir değer otomatik olarak **true**'ya ve herhangi bir 0 değeri ise otomatik olarak **false**'a dönüştürülür. Bunun tersi de doğrudur: Bir **bool** değeri bir tamsayı ifadede kullanıldığında **true** 1'e, **false** da 0'a dönüştürülür. **bool**'un eklenmesi daha fazla tip denetimi yapmamıza izin verir, Boolean ve tamsayı tiplerini birbirinden ayırmamızı sağlar. Tabii ki kullanım isteğimize bağlıdır; fakat **bool**'un oldukça kullanışlı olduğunu hatırlatmakta fayda var.

Örnekler

1. Bir C programında komut satırı argümanı yoksa **main()**'i aşağıda gösterilen şekilde deklare etmek oldukça yaygın bir kullanım şeklidir:

```
int main(void)
```

Fakat C++'da void'un kullanımı bir fazlalıktır ve gereksizdir.

2. Bu kısa C++ programı derlenemez, sum() fonksiyonunun prototipi yoktur.

```
// Bu program derlenmeyecek.
#include <iostream>
using namespace std;

int main()
{
    int a, b, c;
    cout << "İki sayı girin: ";
    cin >> a >> b;
    c = sum(a, b);
    cout << "Toplam: " << c;
    return 0;
}

// Bu fonksiyonun bir prototipi bulunmalı.
sum(int a, int b)
{
    return a+b;
}
```

3. Yerel değişkenlerin blok içerisinde herhangi bir yerde deklare edilebileceğinin gösterildiği örnek:

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

int main()
{
    int i; // bloğun başında deklare edilmiş yerel değişken
    cout << "Sayı girin: ";
    cin >> i;

    // faktöriyeli hesap et
    int j, fact=1; // iş deyiminden sonra deklare edilmiş değişkenler
    for(j=i; j>=1; j--) fact = fact * j;
    cout << "Faktöriyel " << fact;

    return 0;
}
```

j ve fact'in ilk kullanıldığı noktaya yakın yerde deklare edilmesi bu kısa örnekte pek önemli değildir, fakat büyük fonksiyonlarda değişkenlerin ilk kullanıldığı yere yakın bir noktada deklare edilebilmesi kodunuzun anlaşılabilirliğini artırır ve istenmeyen yan etkileri önler.

4. Aşağıdaki programda outcome adında bir Boolean değişken oluşturulmaktadır ve bu değişkene false değeri atanmaktadır. Sonra bu değişken bir if deyiminde kullanılmaktadır.

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

int main()
{
    bool outcome;
    outcome = false;

    if(outcome) cout << "true";
    else cout << "false";

    return 0;
}
```

Sizin de tahmin ettiğiniz gibi program ekrana false değerini gösterir.

Örnekler

- Aşağıdaki program C++ programı olarak derlenmeyecektir. Neden?

```
// Bu program bir hata içermektedir.
#include <iostream>
using namespace std;

int main()
{
    f();
    return 0;
}

void f()
{
    cout << "bu çalışmaz ";
}
```

- Bir C++ programında çeşitli noktalarda yerel değişkenleri deklare etmeyi deneyin. Aynı şeyi bir C programında deneyin ve hangi bildirimlerin hataya neden olduğuna dikkat edin.

1.7. Fonksiyonların Aşırı Yüklenmesine Giriş

Sınıflardan sonra C++'ın belki de en önemli ve en yaygın özelliği *fonksiyonların aşırı yüklenmesidir* (*function overloading*). Fonksiyonları aşırı yükleyerek sadece C++'a bir çeşit polimorfizm kazandırmış olmuyoruz, aynı zamanda C++ programlama ortamını dinamik olarak genişletebilmenin temellerini de oluşturuyoruz. Aşırı yükleme çok önemli bir konu, bu nedenle burada kısa bir giriş yapalım.

C++'da, iki veya daha çok fonksiyon, argümanlarının tipleri veya argüman sayısı ya da her ikisi farklı olduğu sürece aynı adı paylaşabilirler. İki veya daha fazla fonksiyon aynı adı paylaşıyorsa onlar *aşırı yüklenmiştir* denir. Aşırı yüklenmiş fonksiyonlar, birbiriyile alakalı işlemlerin aynı adla çağrımasına izin vererek programın karmaşıklığını azaltabilir.

Bir fonksiyonu aşırı yüklemek oldukça kolaydır : fonksiyonun gereklili olan tüm versiyonlarını deklare edin ve tanımlayın. Derleyici, otomatik olarak fonksiyonu çağrıırken kullanılan argümanların sayısına ve tipine göre fonksiyonun doğru versiyonunu seçecektir.

NOT C++'da operatörleri aşırı yüklemek de mümkündür. Fakat operatörlerin aşırı yüklenmesini tam olarak enlemeden önce C++'ı daha iyi kavramamız gerekiyor.

Örnekler

1. Fonksiyonların aşırı yüklenmesinin temel nedenlerinden biri, bir arabirimde birden fazla metod felsefesini oluşturan tasarım polimorfizmini elde etmektir. Biliyoruz ki C'de, sadece işledikleri veri tipleri farklı, fakat birbiriyle bağlantılı birkaç fonksiyon kullanmak pek yaygındır. Bu durumun klasik örneği C standart kütüphanesindedir. Daha önce belirttiğimiz gibi, kütüphanede **abs()**, **labs()** ve **fabs()** fonksiyonları bulunmaktadır. Bunlar sırasıyla bir tamsayının mutlak değerini, long tamsayının mutlak değerini ve bir reel sayının mutlak değerini vermektedir.

Fakat üç farklı veri tipinin olması nedeniyle üç ayrı adın kullanılması, durumu olduğundan daha karmaşık hale getiriyor. Her üç durumda da mutlak değer döndürülyor; sadece verinin tipi değişiyor. C++'da bu durumu, örnekte de görüldüğü gibi, üç veri tipi için bir ada aşırı yükleme yaparak düzleştirebiliriz.

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

// abs()'a üç şekilde aşırı yükleyin
int abs(int n);
long abs(long n);
double abs(double n);

int main()
{
    cout << "-10'un mutlak değeri: " << abs(-10) << "\n\n";
    cout << "-10L'nin mutlak değeri: " << abs(-10L) << "\n\n";
    cout << "-10.01'nin mutlak değeri: " << abs(-10.01) << "\n\n";

    return 0;
}

// abs() for ints
int abs(int n)
{
    cout << "Tamsayılar için abs()\n";
    return n<0 ? -n : n;
}

// long tamsayılar için abs()
long abs(long n)
{
    cout << "long sayılar için abs()\n";
    return n<0 ? -n : n;
}
```

```
// double için abs()
double abs(double n)
{
    cout << "Double sayılar için abs()\n";
    return n<0 ? -n : n;
}
```

Bu programda, her veri tipi için bir tane olmak üzere `abs()` adında üç fonksiyon tanımlıyoruz. `main()`'in içerisinde `abs()` üç ayrı argüman tipi kullanılarak çağrılır. Derleyici, argüman olarak kullanılan verinin tipine göre otomatik olarak fonksiyonun doğru versiyonunu çağırır. Program aşağıdaki sonucu verir:

```
Tamsayılar için abs()
-10'un mutlak değeri: 10

Long sayılar için abs()
-10L'un mutlak değeri: 10

Double sayılar için abs()
-10.01'un mutlak değeri: 10.01
```

Bu örnek çok basittir, fakat yine de fonksiyonların aşırı yüklenmesinin ne kadar değerli olduğunu göstermektedir. Genel bir iş sınıfı tek bir adla tanımlandığından birbirine benzer üç farklı adın (bu durumda `abs()`, `fabs()` ve `labs()`) oluşturduğu karmaşıklık engellenmiş olur. Şimdi sadece tek bir adı, *genel* işi tanımlayan adı hatırlamanız gerekiyor. Fonksiyonun (yani metodun) çağrılması gereken uygun versiyonunu seçmek derleyiciye kalıyor. Böylece karmaşıklık azalıyor ve polimorfizmin kullanılmasıyla üç ad bire düşüyor.

Polimorfizmin kullanılması bu örnekte önelsiz gibi görünse de çok büyük programlarda "bir arabirimde birden çok metod" yaklaşımının oldukça etkili olduğunu görebilirsiniz.

2. Fonksiyonların aşırı yüklenmesi ile ilgili bir başka örnek. Bu örnekte `date()` fonksiyonu veriyi katar veya üç tamsayı şeklinde alabilecek şekilde aşırı yüklenmiştir. Her iki durumda da fonksiyon kendisine gönderilen tarihi göstermektedir.

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

void date(char *date); // katar şeklinde tarih
void date(int ay, int gün, int yıl); // sayı şeklinde tarih

int main()
{
    date("8/23/99");
    date(8, 23, 99);

    return 0;
}

// Katar şeklinde tarih.
void date(char *date)
{
```

```

    cout << "Date: " << date << "\n";
}

// Tamsayı şeklinde tarih.
void date(int ay, int gün, int yılı)
{
    cout << "Tarih: " << ay << "/";
    cout << gün << "/" << yıl << "\n";
}

```

Fonksiyonların aşırı yüklenmesi için verdığımız bu örnek, bir fonksiyon için en doğal arabirimin nasıl sağladığını bize gösteriyor. Tarihin hem katar şeklinde hem de ay, gün ve yılı içeren üç tamsayı şeklinde temsil edilmesiyle çok sık karşılaşıldığından duruma göre en uygun şekli seçmekte serbestsiniz.

3. Şimdiye kadar argümanlarının veri tipleri farklı olan aşırı yüklenmiş fonksiyonlar gördünüz. Fakat, aşağıdaki örnekteki gibi, argümanlarının sayıları farklı fonksiyonlar da aşırı yüklenebilirler:

```

#include <iostream>
using namespace std;
void f1(int a);
void f1(int a, int b);

int main()
{
    f1(10);
    f1(10, 20);
    return 0;
}
void f1(int a)
{
    cout << "f1(int a)\n";
}
void f1(int a, int b)
{
    cout << "f1(int a, int b)\n";
}

```

4. Döndürülen sonucun tipinin farklı olması fonksiyonun aşırı yüklenemesi için yeterli değildir. Eğer iki fonksiyonun, sadece döndürdüğü veri tipleri farklıysa derleyici doğru fonksiyonu her zaman seçemeyebilir. Örneğin aşağıdaki program yanlıştır, çünkü içerisinde belirsizlik vardır:

```

// Bu yanlıştır ve derlenmez.
int f1(int a);
double f1(int a);
.

.

f1(10); // derleyici hangi fonksiyonu çağıracak???

```

Açıklamada da belirttiğimiz gibi, derleyici **f1()**'in hangi versiyonunun çağrılacağını bilemez.

Aliştırmalar

- Argümanının karesini döndüren **sroot()** adında bir fonksiyon oluşturun. **sroot()**'u üç şekilde aşırı yükleyin, bunlardan biri bir tamsayının, diğeri bir long tamsayının, diğeri de bir **double** sayının karesini alsin. (Kare alma hesabını gerçekleştirmek için standart kütüphanedeki **sqrt()** fonksiyonunu kullanabilirsiniz.)
- C++ standart kütüphanesi aşağıdaki üç fonksiyonu içerir. Bu fonksiyonlar s tarafından işaret edilen katarın içindeki nümerik değeri döndürürler. **atof()** bir **double** döndürür, **atoi()** bir tamsayı döndürür ve **atol()** bir long döndürür. Bu fonksiyonları aşırı yüklemek neden mümkün değildir?

```
double atof(const char *s);
int atoi(const char *s);
long atol(const char *s);
```

- Fonksiyonu çağrıırken kullanılan iki argümanın en küçüğünü döndüren **min()** adında bir fonksiyon oluşturun. **min()**'i, karakterleri, tamsayıları ve **double**'ları argüman olarak kabul edebilecek şekilde aşırı yükleyin.
- Argümani ile belirlenen sayıda saniye süresince bilgisayarı durdurulan **sleep()** adında bir fonksiyon oluşturun. **sleep()**'ı bir tamsayı ile ya da tamsayının katar temsili ile çağrılabilecek şekilde aşırı yükleyin. Örneğin, **sleep()**'e yapılan bu çağrımların ikisi de bilgisayarı 10 saniye için durduracaktır:

```
sleep(10);
sleep("10");
```

Fonksiyonlarımızı onları kısa bir program içerisinde çalıştırarak gösterin. (Bilgisayarı durdurmak için bir gecikme döngüsü kullanmakta serbestsiniz.)

1.8. C++ Anahtar Kelimeleri

C++, C tarafından tanımlan tüm anahtar kelimeleri desteklemektedir ve bunlara 30 tane de yeni kelime eklemektedir. C++ tarafından tanımlan tüm anahtar kelimeler Tablo 1.1'de gösterilmektedir. C++'ın eski sürümleri **overload** anahtar kelimesini tamir, fakat bu kelime artık kullanılmamaktadır.

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	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			

Tablo 1.1. C++ anahtar kelimeleri

Pekiştirme Testi

Bu noktada aşağıdaki alıştırmaları yapabiliyor ve soruları cevaplayabiliyor olmalısınız.

1. Polimorfizm, depolama ve mirası kısaca tanımlayın.
2. Bir C++ programında nasıl açıklama yapıldır?
3. C++ stili I/O'ları kullanarak, klavyeden iki tam sayı girilen, sonra birinci sayıyı ikincinin üssü olarak alıp sonucu hesaplayan ve sonucu ekranda gösteren bir program yazın. (Örneğin kullanıcı 2 ve 4 girerse sonuç 16 olur.)
4. Bir katarı tersine çeviren `rev_str()` adında bir fonksiyon oluşturun. `rev_str()`'i bir veya iki katar dizisi ile çağrılabilecek şekilde aşırı yükleyin. Tek bir katarla çağrıldığında bu katar yerine tersine çevrilmiş katar koyun. İki katarla çağrıldığında tersine çevrilmiş katar ikinci argümana koyun. Örneğin:

```
char s1[80], s2[80];
strcpy(s1, "merhaba");
rev_str(s1, s2); // ters çevrilmiş katar s2'ye konulur, s1'e dokunulmaz.
rev_str(s1); // tersine çevrilmiş katar s1'in içine döndürülür.
```

5. Aşağıda verilen yeni stil C++ programının eski stile nasıl dönüştürülebileceğini gösterin.

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

int f(int a);

int main()
{
    cout << f(10);
    return 0;
}

int f(int a)
{
    return a * 3.1416;
}
```

6. `bool` veri tipi nedir?

www.Gerogokku.com

BÖLÜM 2

Sınıflara Giriş

Constructor ve Destructor Fonksiyonları

Parametre Alan Constructor'lar

Mirasa Giriş

Nesne İşaretçileri

Sınıf, Yapı ve Bileşim Birbiriyle Bağlıantılıdır.

In-Line Fonksiyonlar

Otomatik In-Line İşlemi

Bu bölümde sınıfları ve nesneleri tanıyalım. Burada çeşitli açılardan C++'la ilgisi olan pek çok önemli konuyu ele alacağız, bu yüzden dikkatle okumanızı tavsiye ediyoruz.

Gözden Geçirme Testi

Bu bölümme başlamadan önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

1. C++ stilinde I/O kullanarak, kullanıcıdan bir katar girmesini isteyen ve sonra bu katarın uzunluğunu gösteren bir program yazın.
2. Ad ve adres bilgilerini saklayan bir sınıf oluşturun. Bu bilgileri, sınıfın private üyesi olan karakter katarları halinde saklayın. Ad ve adresin saklanması ve ekranda gösterilmesi için iki public fonksiyon oluşturun. (Bu fonksiyonlara **sakla()** ve **goster()** adlarını verin.)
3. **rotate()** adında aşırı yüklenmiş bir fonksiyon oluşturun. Bu fonksiyon argümanındaki bitleri sola döndürsün ve sonra elde ettiği sonucu versin. Bu fonksiyonu argüman olarak **int** tipinde ve **long** tipinde değişkenler alabilecek şekilde aşırı yükleyin. Döndürme işlemi, shift (kaydırma) işlemine benzemektedir. Aralarındaki tek fark, shift işleminde en sondaki bitin yok olmaması, bunun yerine sayının diğer ucuna gitmesidir.
4. Aşağıdaki programda nasıl bir hata var?

```
#include <iostream>
using namespace std;
class myclass {
    int i;
public:
    :
    :
};

int main()
{
    myclass ob;
    ob.i = 10;
    :
    :
}
```

2.1. Constructor ve Destructor Fonksiyonları

Program geliştirmekle uzun süredir uğraşıyorsanız, programınızın bazı elemanları için hazırlık yapılmasının gerektiğini bilirsiniz. Nesnelerle çalışıyorsanız bu durumla daha da sık karşılaşıyorsunuzdur. Hatta gerçek problemler düşünülürse, her nesne bu tür bir hazırlığa gereksinim duyar. C++'da bu gereksinim, sınıf bildirimini içerisinde *constructor fonksiyonlarının*

konulmasıyla karşılanır. Bir sınıfın yapıcısı (constructor), bu sınıfa ait nesnelerin her oluşturulusunda çağrılr. Böylelikle bir nesne için gerçekleştirilmesi gereken tüm hazırlıklar constructor fonksiyonu tarafından otomatik olarak yapılabilir.

Constructor fonksiyonu ait olduğu sınıf ile aynı addadır ve bu fonksiyonun geri dönürecegi bir veri tipi yoktur. Constructor fonksiyonu olan örnek sınıfı aşağıda inceleyelim:

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

class myclass {
    int a;
public:
    myclass(); // constructor
    void show();
};

myclass::myclass()
{
    cout << "constructor'da\n";
    a = 10;
}

void myclass::show()
{
    cout << a;
}

int main()
{
    myclass ob;
    ob.show();
    return 0;
}
```

Bu basit örnekte **a** değeri **myclass()** constructor'ı (yapıcısı) tarafından hazır hale getiriliyor. Constructor, **ob** nesnesi oluşturulurken çağrılr. Nesne, kendisine ait bildirim deyimi işletildiğinde oluşturulur. C++'da , değişkenin bildirim deyiminin bir "iş deyimi" olması oldukça önemli. C'de program geliştirirken, bildirim deyimleri basitçe değişkenlerin oluşturulduğu yer olarak düşünülebilir. Oysa C++'da, nesne constructor'a sahip olabileceğiinden değişkenin bildirim deyimi dikkate değer sayıda işin yapılmasını sağlayabilir.

myclass()'ı nasıl tanımladığımıza dikkat edelim. Daha önce de söylediğim gibi, bu fonksiyonun **return** (döndürme) tipi yoktur. C++'ın resmi yazılım kurallarına göre constructor'ların **return** tipi olması doğru değildir.

Global nesneler için nesne constructor'u, bir kere, program ilk çalışmaya başladığında çağrılr. Yerel nesneler için ise constructor, bildirim deyiminin her işletilişinde çağrılr. Constructor'ın tersi *destructor*'dır. Bu fonksiyon, nesne yok edilirken çağrılr. Nesnelerle çalışıyorsak onların yok edilmeleri esnasında bazı işlerin yapılması gereklidir. Örneğin, oluşturulduğunda bellekte yer kaplayan bir nesne, yok edilirken bu belleği serbest bırakmak isteyecektir. Destructor da sınıfla aynı addadır, fakat başında ~ işaretini vardır. Aşağıdaki sınıfın destructor fonksiyonunu inceleyelim:

```

#include <iostream>
using namespace std;

class myclass {
    int a;
public:
    myclass(); // constructor
    ~myclass(); // destructor
    void show();
};

myclass::myclass()
{
    cout << "constructor'da\n";
    a = 10;
}
myclass::~myclass()
{
    cout << "Yokediliyor...\n";
}

void myclass::show()
{
    cout << a << "\n";
}

int main()
{
    myclass ob;

    ob.show();

    return 0;
}

```

Sınıfa ait destructor, nesne yok edilirken çağrıılır. Yerel nesneler bulunduğu alandan çıkışında yok edilirlerken **global nesneler program sona erdiğinde yok edilirler.** Bir constructor'ın veya destructor'ın adresinin alınması imkansızdır.

NOT

Teknik açıdan düşünecek olursak constructor veya destructor her tür işlemi yapabilir. Bu fonksiyonların kodunda, tanımlanıldıkları sınıfta ilgili herhangi bir şeyin hazır hale getirilmesi veya sıfırlanması zorunlu değildir. Örneğin, aşağıdaki örneklerdeki constructor, pi'yi 100 ayrı yere hesaplayabilirdi. Fakat nesnenin hazırlığıyla ve düzgün bir şekilde yok edilmesiyle doğrudan bağlantılı olmayan işleri constructor veya destructor'a yapmak programcılığının zayıf olduğunu gösterir, bundan kaçınmalıyız.

Örnekler

1. Bölüm 1'de oluşturulan **stack** sınıfını hatırlayın. Bu sınıf, yiğinin index değişkeninin ayarlanması için bir hazırlık fonksiyonu gerektiriyordu. Bu işlem, kesinlikle constructor fonksiyonunun yapması gereken bir işlemidir. Burada **stack** sınıfı biraz daha geliştirilmiştir. Bu sınıf, kendisine ait bir nesne oluşturulduğunda otomatik olarak gerekli hazırlığı yapan bir constructor fonksiyonu kullanmaktadır:

BÖLÜM 2: Sınıflara Giriş

```

#include <iostream>
using namespace std;

#define SIZE 10

// Karakterler için stack sınıfı deklare ediliyor.
class stack {
    char stck[SIZE]; // bu, stack'ı saklamak içindir
    int tos; // stack'ın Üst indeksi
public:
    stack(); // constructor
    void push(char ch); // stack'a karakter iter
    char pop(); // stack'dan karakter çeker
};

// stack'ı hazır hale getirir.
stack::stack()
{
    cout << "Yığın oluşturuluyor \n";
    tos = 0;
}

// Bir karakter iter.
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Yığın dolu\n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Bir karakter çeker.
char stack::pop()
{
    if(tos==0) {
        cout << "Yığın boş\n";
        return 0; // Yığın boşsa sıfır döndürür
    }
    tos--;
    return stck[tos];
}

int main()
{
    // Otomatik olarak hazırlanan iki yığın oluşturur.
    stack s1, s2;
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "s1'i çek: " << s1.pop() << "\n";
    for(i=0; i<3; i++) cout << "s2'i çek: " << s2.pop() << "\n";

    return 0;
}

```

Dikkat edersek hazırlık görevi şimdilik, program tarafından ayrıca çağrılmazı gereken ayrı bir fonksiyon yerine, constructor fonksiyonu tarafından otomatik olarak gerçekleştiriliyor. Bu, önemli bir noktadır. Hazırlık işlemleri, nesne oluşturulurken otomatik olarak gerçekleştirildiğinden, herhangi bir hata meydana gelmesiyle gerekli hazırlığın gerçekleştirilmemesi gibi bir durum söz konusu olmaz. Nesneleri kullanarak programımızın karmaşıklığını yine azaltmış olduk. Sizin, programcı olarak, hazırlık konusunda endişelenmenize gerek yok; nesne var duruma getirildiğinde bu işlemler otomatik olarak gerçekleştiriliyor.

2. Bu örnek, constructor ve destructor fonksiyonlarına neden ihtiyaç duyabileceğimi gösteriyor. Burada **strtype** adında, bir katar ve bu katarın uzunluğunu içeren basit bir katar sınıfı oluşturuyoruz. **strtype** sınıfında bir nesne oluşturulduğunda, bellekte katar ve onun uzunluğu için yer ayrıılır. Katarın uzunluğuna ilk değer olarak sıfır atanıyor. **strtype** sınıfındaki nesne yok edildiğinde bellekteki bu yer serbest bırakılır.

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

#define SIZE 255

class strtype {
    char *p;
    int len;
public:
    strtype(); // constructor
    ~strtype(); //destructor
    void set(char *ptr);
    void show();
};

// Katar nesnesinin hazırlığı.
strtype::strtype()
{
    p = (char *) malloc(SIZE);
    if(!p) {
        cout << "Bellekte yer ayırma hatası \n";
        exit(1);
    }
    *p = '\0';
    len = 0;
}

// Katar nesnesi yok edilirken bellek serbest bırakılıyor.
strtype::~strtype()
{
    cout << "p, serbest bırakılıyor \n";
    free(p);
}

void strtype::set(char *ptr)
{
    if(strlen(p) >= SIZE) {
```

BÖLÜM 2: Sınıflara Giriş

```

        cout << "Hatar fazla büyük\n";
        return;
    }

    strcpy(p, ptr);
    len = strlen(p);
}

void strtype::show()
{
    cout << p << " - uzunluğu: " << len;
    cout << "\n";
}

int main()
{
    strtype s1, s2;
    s1.set("Bu bir deneme.");
    s2.set("C++'ı seviyorum.");
    s1.show();
    s2.show();
    return 0;
}

```

Bu programda, bellekte yer ayrılması ve bu yerin boşaltılması için **malloc()** ve **free()** fonksiyonlarını kullanıyoruz. Bu, tamamen geçerli bir yoldur, fakat C++, belleğe dinamik olarak erişmek için başka bir yol daha sunuyor, daha ileri konularda bu yoldan da bahsedeceğiz.

NOT

Bu programda, kullanılan C kütüphane fonksiyonları için yeni stilde başlıklar kullanılıyor. Bölüm 1'de bahsettiğimiz gibi derleyiciniz bu başlıklar desteklemiyorsa onların yerine standard C başlık dosyalannı koyun. Bu, C kütüphane fonksiyonlarının kullandığı diğer programlar için de geçerli.

3. Burada nesneye ait constructor ve destructor fonksiyonları çok ilginç bir şekilde kullanılıyor. Bu programda, **timer** tipinde bir nesnenin oluşturulması ve yok edilmesi arasındaki süreyi ölçmek için **timer** sınıfından bir nesne kullanıyoruz. Nesneye ait destructor çağrıldığında, geçen zaman gösterilir. Burada olduğu gibi, bir nesneyi, programın süresini veya bir fonksiyonun bir blok içerisinde ne kadar zaman harcayacağını öğrenmek için kullanabiliriz. Tek yapmamız gereken, zaman ölçme aralığının bitmesini istediğimiz yerde nesnenin yok olmasını sağlamaktır.

```

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

class timer {
    clock_t start;
public:
    timer(); // constructor
    ~timer(); // destructor
};

timer::timer()
{

```

```

    start = clock();
}

timer::~timer()
{
    clock_t end;

    end = clock();
    cout << "Geçen zaman: " << (end-start) / CLOCKS_PER_SEC << "\n";
}

int main()
{
    timer ob;
    char c;

    // gecikme...
    cout << "Bir tuşa ve sonra ENTER'a basın: ";
    cin >> c;

    return 0;
}

```

Bu programda, `clock()` standart kütüphane fonksiyonunu kullanıyoruz. Bu fonksiyon, program çalışmaya başladıkten sonra geçen saat çevrim sayısını döndürür. Bu değer, `CLOCKS_PER_SEC` ile bölünerek saniyeye çevrilir.

Aliştırmalar

1. Bölüm 1'in Aliştırmalarında oluşturduğunuz `queue` (kuyruk) sınıfını tekrar inceleyin ve bu sınıfın hazırlık fonksiyonu yerine bir constructor kullanın.
2. `stopwatch` adında geçen zamanı tutan kronometre görevi görecek bir sınıf oluşturun. Geçen zamanın ilk değerine sıfır koyan bir constructor fonksiyonu kullanın. `start()` (başla) ve `stop()` (dur) adında bu sınıfa ait iki fonksiyon oluşturun. Bu fonksiyonlar sırasıyla zamanlayıcıyı başlatmak ve durdurmak içindir. `show()` (göster) adında geçen zamanı gösteren başka bir fonksiyon daha oluşturun. Öte yandan `stopwatch` nesnesi yok edilirken destructor fonksiyonunuz, otomatik olarak geçen zamanı göstersin. (Problemi basitleştirmek açısından, zamanı saniye cinsinden gösterin.)
3. Aşağıdaki programda verilen constructor neden hatalıdır?

```

class sample {
    double a, b, c;
public:
    double sample(); // neden hatalı?
};

```

2.2. Parametre Alan Constructor'lar

Bir constructor fonksiyonuna argüman göndermemiz mümkündür. Bunun için constructor fonksiyonunun bildirimine ve tanımına, uygun parametreler eklememiz yeterlidir. O halde nesneleri deklare ederken argümanlarını belirleyeceğiz. Bunu nasıl yapacağımızı görmek için aşağıdaki örneğe bir göz atalım:

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

class myclass {
    int a;
public:
    myclass(int x); // constructor
    void show();
};

myclass::myclass(int x)
{
    cout << "constructor'da\n";
    a = x;
}

void myclass::show()
{
    cout << a << "\n";
}

int main()
{
    myclass ob(4);

    ob.show();

    return 0;
}
```

`myclass`'in parametre aldığı constructor fonksiyonu, `myclass()`'a gönderilen değer `a`'nın hazırlığı için kullanılmaktadır. `ob`'un `main()`'in içerisinde nasıl deklare edildiğine özellikle dikkat edin. `ob`'dan sonra gelen parantezlerin içinde verilen 4 değeri, `a`'ya ilk değer vermek için `myclass()`'ın `x` parametresine gönderilen argümandır.

Gerçekte parametrelî bir constructor'a argüman gönderme işlemi şu notasyonla yapılmalıdır:

```
myclass ob = myclass(4);
```

Fakat, C++ programcılarının çoğu, kısa şekli tercih etmektedir. Bu iki yazım şekli arasında, aslında teknik açıdan küçük bir fark bulunmaktadır. Bu fark, constructor'ların kopyalanmasıyla ilgilidir ve daha sonra ele alınacaktır. Fakat aradaki bu fark konusunda şu anda endişelenmemiz gerekmeyor.

NOT

Constructor fonksiyonlarının tersine, destructor fonksiyonları parametre alamazlar. Bunun sebebi oldukça basittir: Yok edilen bir nesneye argümanlar göndermeyeceğiz bir mekanizma bulunmamaktadır.

Örnekler

1. Constructor'a birden fazla argüman göndermek mümkündür, hatta bu oldukça sık yapılan bir iştır. Burada **myclass()**'a iki argüman gönderiyoruz:

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

class myclass {
    int a, b;
public:
    myclass(int x, int y); // constructor
    void show();
};

myclass::myclass(int x, int y)
{
    cout << "constructor'da\n";
    a = x;
    b = y;
}

void myclass::show()
{
    cout << a << ' ' << b << "\n";
}

int main()
{
    myclass ob(4, 7);

    ob.show();

    return 0;
}
```

Burada 4, x'e ve 7, y'ye gönderilmektedir. Aynı yöntemle, istediğimiz sayıda argüman gönderebiliriz (tabii ki, derleyici tarafından konulan limite kadar).

2. Burada **stack** sınıfını, yiğina bir "isim" göndermek için parametreli constructor kullanacak şekilde değiştirdik. Bu tek karakterlik ad, bir hata çıktığında bahsedilen yiğini belirtmek için kullanılacak.

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

#define SIZE 10

// Karakterler için stack sınıfı deklare ediliyor.
class stack {
    char stck[SIZE]; // bu, yiğini saklamak içindir
    int tos; // yiğinin en üst indeksi
    char who; // yiğini tanımlar
```

```

public:
    stack(char c); // constructor
    void push(char ch); // Yiğine karakter iter
    char pop(); // Yiğinden karakter çeker
};

// Yiğin için gerekli hazırlık yapılıyor.
stack::stack(char c)
{
    tos = 0;
    who = c;
    cout << "Yiğin oluşturuluyor " << who << "\n";
}

// Bir karakter iter.
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Yiğin " << who << " dolu\n";
        return;
    }
    stack[tos] = ch;
    tos++;
}

// Bir karakter çeker.
char stack::pop()
{
    if(tos==0) {
        cout << "Yiğin " << who << " boş\n";
        return 0; // Yiğin boşsa sıfır döndürür
    }
    tos--;
    return stack[tos];
}

int main()
{
    // Otomatik olarak hazırlanan iki yiğin (stack) oluşturuluyor.
    stack s1('A'), s2('B');
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    // Bu, bazı hata mesajlarına neden olacaktır.

    for(i=0; i<5; i++) cout << "s1'i çekin: " << s1.pop() << "\n";
    for(i=0; i<5; i++) cout << "s2'i çekin: " << s2.pop() << "\n";

    return 0;
}

```

Bu örnekte olduğu gibi nesnelere "isim" verilmesi, özellikle programın sonradan incelenmesi gerekiğinde ve bir nesnede hata meydana geldiğinde faydalı olur.

3. Parametreli constructor fonksiyonu kullanan **strtype** sınıfını (önceki oluşturmuşuk) oluşturmanın diğer bir yolu:

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

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype();
    void show();
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr);
    p = (char *) malloc(len+1);
    if(!p) {
        cout << "Bellekte yer ayırma hatası \n";
        exit(1);
    }
    strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "p, serbest bırakılıyor\n";
    free(p);
}
void strtype::show()
{
    cout << p << " -uzunluk: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("Bu, bir deneme."), s2("C++'ı seviyorum.");
    s1.show();
    s2.show();
    return 0;
}
```

strtype'in bu versiyonunda constructor fonksiyonu kullanılarak katara bir başlangıç değeri verilmektedir.

4. Daha önceki örneklerde hep sabitler gönderdik, fakat nesnelerin constructor'larına değişkenler de dahil olmak üzere geçerli herhangi bir ifade gönderebiliriz. Örneğin, bu program nesnenin oluşturulması için kullanıcıdan bilgi almaktadır:

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

class myclass {
```

```

int i, j;
public:
    myclass(int a, int b);
    void show();
};

myclass::myclass(int a, int b)
{
    i = a;
    j = b;
}
void myclass::show()
{
    cout << i << ' ' << j << "\n";
}
int main()
{
    int x, y;

    cout << "İki tamsayı girin: ";
    cin >> x >> y;

    // ob'ın oluşturulması için değişkenler kullanılıyor.
    myclass ob(x, y);

    ob.show();

    return 0;
}

```

Bu program, nesneler hakkında önemli bir noktayı daha ortaya koymaktadır : Nesnelerin özellikleri, oluşturuldukları anki duruma uygun bir şekilde belirlenebilir. C++ hakkında daha fazla şey öğrendikçe nesne oluşturmaının ne kadar işimize yaradığını göreceğiz.

Aliştırmalar

- stack sınıfını yiğin için bellekte dinamik olarak yer ayırbilmesini sağlayacak şekilde değiştirin. Yiğinin büyüklüğü, constructor fonksiyonunun parametresi ile belirlensin. (Bu belleği bir destructor fonksiyonu ile serbest bırakmayı unutmayın.)
- t_and_d** adında bir sınıf oluşturun. Bu sınıf oluşturulurken o anki sistem saatini ve tarihi kendi constructor'ına parametre olarak göndersin. Zamanı ve tarihi ekran da gösteren bu sınıfa ait bir de fonksiyon oluşturun. (İpucu : Zamanı ve tarihi öğrenmek ve göstermek için standard kütüphanede bulunan time ve date fonksiyonlarını kullanın.)
- Constructor fonksiyonuna, her biri bir kutunun kenar uzunluklarını temsil eden üç **double** değer gönderen **box** adında bir sınıf oluşturun. Bu **box** sınıfı, kutunun hacmini hesaplasın ve sonucu bir **double** değişkeninde saklasın. Bu sınıfa ait **vol()** adında bir fonksiyon oluşturun. Bu fonksiyon, her **box** nesnesinin hacmini göstersin.

2.3. Mirasa Giriş

Miras konusunu Bölüm 7'de detaylarıyla ele alacağız, şimdilik konuya küçük bir giriş yapalım. C++ için miras, bir sınıfın bir diğer sınıfın özelliklerini mirasla alabildiği mekanizmadır. Miras, sınıflar arasında en genelden en özele doğru bir hiyerarşinin oluşmasını sağlar.

Konuya başlamadan önce, sık kullanacağımız iki terimi tanımlayalım. Bir sınıf, diğer bir sınıfın miras aracılığıyla oluşturulurken, ebeveyn sınıfı *temel sınıf*, mirasla oluşturulan sınıf ise *alt sınıf* diyoruz. Genelde miras süreci, temel sınıfın tanımlanmasıyla başlar. Temel sınıf, tüm alt sınıfların sahip olacağı genel özellikleri tanımlar. Esasen temel sınıf, birtakım özelliklerin en genel tanımını temsil etmektedir. Alt sınıf, bu genel özellikleri mirasla alır ve onlara sadece kendisine özgü yeni özellikler ekler.

Bir sınıfın bir diğer sınıfın mirasla oluşmasının ne anlamına geldiğini anlamak istiyoruz, bunun için kısa ve basit bir örnekde işe başlayalım. Bu örnek basit olduğu halde mirasın anahtar özelliklerinden pek çoğunu bize gösterecek. İlk olarak temel sınıfın deklarasyonunu inceleyelim:

```
// Temel sınıf tanımlanıyor.
class B {
    int i;
public:
    void set_i(int n);
    int get_i();
};
```

Bu temel sınıf kullanılarak bir alt sınıf oluşturuluyor:

```
// Alt sınıf tanımlanıyor.
class D : public B {
    int j;
public:
    void set_j(int n);
    int mul();
};
```

Bu deklarasyonu daha yakından inceleyelim. **D** sınıfından sonra **public** anahtar kelimesi, ondan sonra da iki nokta üst üste ve **B** sınıfı adı geliyor, buna dikkat edelim. Bu satırda, derleyiciye **D** sınıfının, **B** sınıfına ait tüm bileşenleri mirasla alacağını söylüyoruz. **public** anahtar kelimesi, derleyiciye, temel sınıfın tüm **public** elemanlarının alt sınıfın da **public** elemanları olacağını söylemektedir. Fakat temel sınıfın tüm **private** elemanları kendisine **private** (özel) kılır ve alt sınıf onlara doğrudan erişemez:

B ve **D** sınıflarını kullanan programın tamamı:

```
// Mirasa basit bir örnek.
#include <iostream>
using namespace std;

// Temel sınıfın tanımlanması.
class B {
    int i;
```

```
public:
    void set_i(int n);
    int get_i();
};

// Alt sınıfın tanımlanması.
class D : public B {
    int j;
public:
    void set_j(int n);
    int mult();
};

// Temel sınıf için i değerinin verilmesi.
void B::set_i(int n)
{
    i = n;
}

// Temel sınıf için i değerinin dondurülmesi.
int B::get_i()
{
    return i;
}

// Alt sınıf için j değerinin verilmesi.
void D::set_j(int n)
{
    j = n;
}

// Temel sınıfın i'si çarpı alt sınıfın j'si değerinin döndürülmesi.
int D::mul()
{
    // Alt sınıf, temel sınıfın public fonksiyonlarını çağırabilir.
    return j * get_i();
}

int main()
{
    D ob;

    ob.set_i(10); // Temel sınıfta i'ye değer veriliyor
    ob.set_j(4); // Alt sınıfta j'ye değer veriliyor

    cout << ob.mul(); // ekranda 40 gösteriliyor

    return 0;
}
```

mul() fonksiyonunu nasıl tanımladığımıza bir bakalım. Bu fonksiyonda, **B** temel sınıfına ait olan, **D**'ye ise ait olmayan **get_i()** fonksiyonunu herhangi bir nesneye bağlamadan çağrıduğumuza dikkatinizi çekmek istiyorum. Bu, mümkündür, çünkü **B**'nin public üyeleri **D**'nin de public üyeleri olmuştur. Fakat, **mul()**'un doğrudan **i**'ye erişmek yerine **get_i()**'yi çağrımak zorunda olmasının nedeni şudur: Temel sınıfa private olarak ait olan üyeler (bu durumda **i**), bu sınıfa özel üyeler olarak kalmaktadır ve herhangi bir alt sınıf tarafından bu üyelerle karşılaşamaz. Bir sınıfın private üyelerine alt sınıflar tarafından erişile-

memesi, depolama özelliğinin sürdürülmesi için gereklidir. Eğer bir sınıf'a ait private üye-lər, bu sınıfın mirasla alınması ile public yapılabileseydi, depolama (encapsulation) özelliği ortadan kalkardı.

Temel sınıfın mirasla alınması, genelde şu notasyonla yapılır:

```
class alt-sınıf-ismi:erişim-türü alt-sınıf-ismi {
    ...
};
```

Burada *erişim-türü* yerine üç anahtar kelimededen birini yazacağımız **public**, **private** veya **protected**. Şimdi mirasla sınıf oluştururken **public**'ı kullanalım. Bu erişim türlerini daha sonra detaylı olarak ele alacağız.

Örnek

1. Aşağıdaki programda, meyvelerin belirli karakteristiklerini içeren **fruit** (meyve) altında bir temel sınıf tanımlıyoruz. Bu sınıfın **apple** (elma) ve **orange** (portakal) adında iki alt sınıf türetiyoruz. Bu sınıflar, **fruit** sınıfının özelliklerine ek olarak, elma ve portakala özgü bazı bilgileri de eklemektedir.

```
// Sınıf mirasına ilişkin bir örnek.
#include <iostream>
#include <cstring>
using namespace std;
enum yn {hayır, evet};
enum color {kırmızı, sarı, yeşil, turuncu};

void out(enum yn x);
char *c[] = { "kırmızı", "sarı", "yeşil", "turuncu" };

// Genel fruit sınıfı.
class fruit {
    // Temel sınıf içindeki her şey public
public:
    enum yn sutekli;
    enum yn mevsimlik;
    enum yn agac;
    enum yn tropik;
    enum color clr;
    char name[40];
};

// Apple sınıfı türetiliyor.
class Apple : public fruit {
    enum yn pisirmekicin;
    enum yn suyuicin;
    enum yn yemekicin;
public:
    void seta(char *n, enum color c, enum yn ck, enum yn crch,
        enum yn el);
    void show();
};
```

```
orange sınıfı türetiliyor.  
class Orange : public fruit {  
    enum yn meyvasuyu;  
    enum yn recel;  
    enum yn yemek;  
public:  
    void seto(char *n, enum color c, enum yn j, enum yn sr,  
    enum yn e);  
    void show();  
};  
  
void Apple::seta(char *n, enum color c, enum yn ck, enum yn crhy, enum yn e)  
{  
    strcpy(name, n);  
    surekli = hayir;  
    mevsimlik = evet;  
  
    agac = evet;  
    tropik = hayir;  
    clr = c;  
    pisirmekicin = ck;  
    ezmekicin= crhy;  
    yemekicin = e;  
}  
  
void Orange::seto(char *n, enum color c, enum yn j, enum yn sr, enum yn e)  
{  
    strcpy(name, n);  
    surekli = hayir;  
    mevsimlik = evet;  
    agac = evet;  
    tropik = evet;  
    clr = c;  
    meyvasuyu = j;  
    recel = sr;  
    yemek = e;  
}  
  
void Apple::show()  
{  
    cout << name << " elma: " << "\n";  
    cout << "Surekli: "; out(surekli);  
    cout << "Mevsimlik: "; out(mevsimlik);  
    cout << "Ağaç: "; out(agac);  
    cout << "Tropik: "; out(tropik);  
    cout << "Color: " << c[clr] << "\n";  
    cout << "Pişirilir mi?: "; out(pisirmekicin);  
    cout << "Ezilip suyu çıkarılır mı?: "; out(ezmekicin);  
    cout << "Yenir mi?: "; out(yemekicin);  
    cout << "\n";  
}  
  
void Orange::show()  
{  
    cout << name << " portakal: " << "\n";  
    cout << "Surekli: "; out(surekli);  
    cout << "Mevsimlik: "; out(mevsimlik);  
    cout << "Ağaç: "; out(agac);  
    cout << "Tropik: "; out(tropik);  
    cout << "Renk: " << c[clr] << "\n";  
    cout << "Meyva suyu yapılır mı?: "; out(meyvasuyu);  
}
```

```

cout << "Reçel: "; out(reçel);
cout << "Yenilir mi?: "; out(yemek);
cout << "\n";
}

void out(enum yn x)
{
    if(x==no) cout << "hayır\n";
    else cout << "evet\n";
}

int main()
{
    Apple a1, a2;
    Orange o1, o2;

    a1.seta("Red Delicious", kirmizi, hayir, evet, evet);
    a2.seta("Jonathan", kirmizi, evet, hayir, evet);

    o1.seto("Navel", turuncu, hayir, hayir, evet);
    o2.seto("Valéncia", turuncu, evet, evet, hayir);

    a1.show();
    a2.show();
    o1.show();
    o2.show();
    return 0;
}

```

Gördük ki, **fruit** temel sınıfı tüm meyve türleri için geçerli çeşitli karakteristikleri içermektedir. (Tabii bu örneğin kitabı konabilecek kadar kısa olması için **fruit** sınıfını oldukça basit tuttuk.) Örneğin, tüm meyveler ya sürekli bitkilerde ya da mevsimslik bitkilerde yetişir. Meyvelerin tümü ya ağaçlarda ya da asma veya çah gibi diğer bitkilerde yetişir. Tüm meyvelerin bir rengi ve bir adı vardır. Bu temel sınıfın özellikleri **Apple** ve **Orange** sınıfları tarafından mirasla alınmıştır. Bu sınıfların her biri ek olarak kendi meyve türlerine özgü bilgiler içermektedir.

Örnekte, mirasın temel kullanım nedenini görebiliyoruz. Burada *tüm* meyvelerle ilişkili genel özellikleri tanımlayan bir temel sınıf oluşturuyoruz ve *her* meyve türüne özgü özellikleri ise alt sınıflara bırakıyoruz.

Program mirasla ilgili önemli bir başka noktası daha gösteriyor. Temel sınıf, bir alt sınıf tarafından tek başına "sahiplenilemez". Temel sınıfın özellikleri istenilen sayıda sınıf tarafından mirasla alınabilir.

Aliştırma

- Size aşağıdaki temel sınıf verilmektedir,

```

class area_cl {
public:
    double height;
    double width;
};

```

Temel sınıfı `area_cl` olan `rectangle` (dikdörtgen) ve `isosceles` (ikizkenar üçgen) adında iki alt sınıf oluşturun. Bu sınıflardan her birinin dikdörtgenin veya ikizkenar üçgenin alanını uygun bir şekilde veren `area()` (alan) adında bir fonksiyonu olsun. `height` (yükseklik) ve `width` (genişlik) değişkenlerine ilk değerlerini atamak için parametreli constructor kullanın.

2.4. Nesne İşaretçileri

Şimdiye kadar nesnelere ait üyelerine nokta işaretini kullanarak erişiyorduk. Bu yöntem, tek nesneye cağızıysak doğrudur. Fakat nesnelere erişirken bu nesneye atanmış bir işaretçiyi de kullanabiliriz. Erişimi işaretçi ile gerçekleştiriyorsak, nokta işaretini yerine ok işaretini (`->`) kullanacağız. (Ok işaretini yapılarla işaretçi atarken yaptığımız şekilde kullanacağız.)

Nesne işaretçilerini herhangi bir değişkeni nasıl deklare ediyorsak o şekilde deklare edeceğiz. Bunun için sınıf adını belirledikten sonra değişken adının başına bir yıldız işaretini (asterisk) koyacağız. Nesnenin adresini elde etmek için ise adın başına & işaretini koyacağız, tipki herhangi bir tipteki değişkenin adresini alırken yaptığımız gibi.

Diğer tiplerin işaretçileri gibi, nesne işaretçileri de arttırdıkları zaman bu tipteki diğer bir nesneyi işaret edeceklerdir.

Örnek

1. Nesne işaretçisi kullanan basit bir örnek:

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

class myclass {
    int a;
public:
    myclass(int x); // constructor
    int get();
};

myclass::myclass(int x)
{
    a = x;
}

int myclass::get()
{
    return a;
}

int main()
{
    myclass ob(120); // nesne oluşturuluyor
    myclass *p; // Nesne için işaretçi oluşturuluyor

    p = &ob; // ob'nin adresini p'ye koy
```

```

cout << "Nesneyi kullanan değer: " << ob.get();
cout << "\n";
cout << "İşaretçiyi kullanan değer: " << p->get();
return 0;
}

```

Örnekteki

```
myclass *p;
```

deklarasyonu **myclass** sınıfına ait bir nesneye işaretçi oluşturuyor. Nesne işaretçisi oluşturulduğunda bir nesne *oluşturulmaz*, sadece mevcut nesne için bir işaretçi oluşturulur, bu durumu kavramamız çok önemli. **ob**'un adresi aşağıdaki bildiri ile **p**'ye konuluyor:

```
p = &ob;
```

Programda, son olarak, nesne üyelerine bir işaretçi üzerinden nasıl eriştiğini görüyoruz.

Nesne işaretçileri konusuna Bölüm 4'te, C++ hakkında daha fazla şey öğrendiğimizde tekrar geri döneceğiz ve nesne işaretçilerinin kendilerine özgü çeşitli özelliklerini öğreneceğiz.

2.5. Sınıf, Yapı Ve Bileşim Birbiriyle Bağlılıdır

Sınıfların notasyonunun yapılarının notasyonuna benzедiğini gördük. Sınıflar ile yapıların gerçekte aynı kapasiteye sahip olduğunu öğrenince şaşırabilirsiniz. C++'da, sınıf tanımları gibi yapı tanımları da, constructor ve destructor fonksiyonlarının yanı sıra tüm üye fonksiyonlarını içerecek şekilde genişletilmiştir. Hatta, bir yapı ile bir sınıf arasındaki tek fark, sınıflara ait üyelerin varsayılan olarak private (özel), yapılara ait üyelerin ise yine varsayılan olarak public (genel) olmasıdır. Yapıların genişletilmiş yazılım şeklini aşağıda görüyoruz:

```

struct tip-ismi {
    // public fonksiyonlar ve veri üyeleri
private:
    // private fonksiyonlar ve veri üyeleri
} nesne-listesi;

```

Hatta, C++'ın resmi yazım kurallarına göre gerek **struct**, gerekse **class**, yeni sınıf tipleri oluşturur. Yeni bir anahtar kelimeyle karşılaştık: **private**. Bu kelime derleyiciye, kendisinden sonra gelen üyelerin bu sınıfa private (özel) olduğunu söylemektedir.

Yüzeysel olarak baktığımızda, sınıfların ve yapıların filen eş kapasitede olması bir fazlalık gibi görünebilir. C++'la yeni tanışanlar bu durumun nedenini merak ediyor olabilirler. Hatta, **class** anahtar kelimesinin gereksiz olduğu da pek sık karşılaşılan bir tartışma konusudur.

Bunun bir "kuvvetli" bir de "zayıf" sebebi vardır. "Kuvvetli" olan sebep (ya da zorlayıcı olan sebep) C'ye uyumluluğun sağlanmasıdır. C++ programlarında, C yapıları geçerlidir. C'de yapılara ait tüm üyeler varsayılan olarak public'tır ve bu durum C++'da da sürdürmek istenmiştir. Daha da ötesi, **class** yazım şekli olarak **struct**'dan ayrı bir varlık olduğundan, sınıf tanımı, C'nin yapı tanımına uymayacak şekilde genişlemekte serbesttir. Bu iki farklı şeyler olduğu sürece, C++'ın geleceği uyumluluk sorunlarıyla kısıtlanmamış olmaktadır.

Birbirine benzeyen sınıf ve yapı kavramlarının her ikisini de kullanmanın "zayıf" nedeni ise C++'da üye fonksiyonlarının içерilmesi için yapı tanımını genişletmenin hiçbir dezavantajı bulunmamasıdır.

Her ne kadar yapılar sınıflarla aynı kapasiteye sahip olsalar da, programcının çoğu C'ye olan uyumluluğu bozmamak için yapıların kullanımını sınırlamışlardır ve onları üye fonksiyonları içermek için kullanmamaktadırlar. Programcının çoğu, hem veri, hem de kod içeren nesneleri tanımlarken **class** anahtar kelimesini kullanırlar. Fakat bu sadece stille alakalı bir meseledir ve sizin tercihinize kalmıştır. (İlerideki konularda **struct**, sadece, fonksiyonu olmayan nesneler için kullanılacaktır.)

Eğer sınıflarla yapılar arasındaki bağlantıyı ilgi çekici bulduysanız C++'in bir diğer özelliği de ilginizi çekecek demektir : bileşimler ve sınıflar da birbiriyle bağlantılıdır. C++'da bileşimler, hem fonksiyon hem de veri içerebilen sınıf tiplerini tanımlar. Bileşimler bu açıdan yapılara benzemektedirler, varsayılan olarak tüm üyeleri, **private** belirleyicisi kullanılıana kadar public'tır. Bir bileşimde tüm veri üyeleri aynı bellek yerini paylaşır (tipki C'de olduğu gibi). Bileşimler, constructor ve destructor fonksiyonlarını içerebilirler. C bileşimleri neyse ki, C++ bileşimleri ile uyumludurlar.

Olaya yüzeysel olarak baktığımızda sınıfların ve yapıların ikisinin birden kullanılıyor olması fazlalıkmiş gibi görünse de, durum bileşimler için böyle değildir. Nesneye dayalı bir dilde depolama özelliğinin korunması önemlidir. Böylelikle bileşimin kod ve veriyi bağlayabilme kabiliyeti, içindeki tüm verilerin aynı yeri paylaştığı sınıf tipleri oluşturmamıza imkan verir. Bunu, sınıfları kullanarak yapamayız.

Bileşimler için C++'da çeşitli sınırlamalar bulunmaktadır. Başka sınıflardan miras aracılığıyla oluşturulamazlar ve başka sınıfların temel sınıfı olamazlar. Bileşimlerin **static** üyeleri olamaz ve constructor veya destructor'ı olan herhangi bir nesne de içeremezler. Bileşimin kendisinin constructor ve destructor'ı olabilir. Son olarak ta, bileşimler gerçek fonksiyonlara sahip olamazlar. (Gerçek fonksiyonlar kitapta daha sonra ele alınacaktır.)

C++'da *anonim bileşim* (*anonymous union*) adı verilen özel bir bileşim tipi vardır. Anonim bileşimlerin tip adları yoktur ve bu bileşim türü için hiçbir değişken deklare edilmez. Aksine, anonim bileşim, derleyiciye kendine ait olan üyelerin aynı bellek yerini paylaşacaklarını söyler. Fakat, diğer açılardan, bu üyeler, normal değişkenler gibi rol oynarlar ve onlara da o şekilde davranışılır. Yani, üyelere, doğrudan, nokta işaretini kullanılmaksızın erişilir. Aşağıdaki programı inceleyerek bunu kavramaya çalışalım:

```
union { // bir anonim bileşim
```

```
int i;
char ch[4];
}
i = 10; // i'ye ve ch'ye doğrudan erişiliyor
ch[0] = 'X';
```

Burada **i** ve **ch**'ya doğrudan erişiyoruz, çünkü bu değişkenler herhangi bir nesnenin parçası değildirler. Belkete aynı yeri paylaşırlar.

Anonim bileşimi kullanarak derleyiciye iki veya daha fazla sayıda değişkenin bellekte aynı yeri paylaşmasını istediğimizi kolayca söyleyebiliriz. Bu özel nitelik haricinde, anonim bileşimlerin üyeleri, tamamen diğer değişkenler gibi davranışlarırlar.

Anonim bileşimlerin bu ek özellikleri yanında, normal bir bileşime konulan tüm sınırlamalar geçerlidir. Global bir anonim bileşimi, **static** olarak deklare etmeliyiz. Anonim bileşimler **private** üyelere sahip olamazlar ve onlara ait üyelerin adları aynı alandaki diğer belirleyicilerle çakışmamalıdır.

Örnekler

1. Sınıf oluşturmak için **struct** kullanan kısa bir program:

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

// sınıf tipi tanımlamak için struct kullanılıyor
struct st_type {
    st_type(double b, char *n);
    void show();
private:
    double balance;
    char name[40];
};
st_type::st_type(double b, char *n)
{
    balance = b;
    strcpy(name, n);
}
void st_type::show()
{
    cout << "İsim: " << name;
    cout << ": $" << balance;
    if(balance<0.0) cout << "****";
    cout << "\n";
}

int main()
{
    st_type acc1(100.12, "Johnson");
    st_type acc2(-12.34, "Hedricks");

    acc1.show();
    acc2.show();
    return 0;
}
```

Daha önce belirttiğimiz gibi, yapıya ait üyeleri varsayılan olarak public'tır. Private üyeleri deklare edilmesi için **private** anahtar kelimesini kullanmalıyız.

C ve C++ yapıları arasındaki bir diğer farka daha dikkatinizi çekmek istiyorum. C++'da yapı etiket-ismi, nesnelerin deklare edilmesinde kullanılabilecek bir tip adı haline gelmektedir. C'de etiket-isminin bir tip olması için **struct** anahtar kelimesinin kendisinden önce gelmesi gerekmektedir.

Aynı programın sınıf kullanılarak yazılmış hali:

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

class cl_type {
    double balance;
    char name[40];
public:
    cl_type(double b, char *n);
    void show();
};

cl_type::cl_type(double b, char *n)
{
    balance = b;
    strcpy(name, n);
}

void cl_type::show()
{
    cout << "İsim: " << name;
    cout << ": $" << balance;
    if(balance<0.0) cout << "****";
    cout << "\n";
}

int main()
{
    cl_type acc1{100.12, "Johnson"};
    cl_type acc2{-12.34, "Hedricks"};
    acc1.show();
    acc2.show();

    return 0;
}
```

2. **double** bir değer içerisindeki ikilik bit düzenini byte byte göstermek için bileşim kullanan bir örnek.

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

union bits {
    bits(double n);
    void show_bits();
    double d;
    unsigned char c[sizeof(double)];
};
```

```

bits::bits(double n)
{
    d = n;
}

void bits::show_bits()
{
    int i, j;

    for(j = sizeof(double)-1; j>=0; j--) {
        cout << "Bayt cinsinden bit düzeni " << j << ": ";
        for(i = 128; i; i >>= 1)
            if(i & c[j]) cout << "1";
            else cout << "0";
        cout << "\n";
    }
}

int main()
{
    bits ob(1991.829);
    ob.show_bits();
    return 0;
}

```

Programın çıkışı şu şekildedir:

```

Bayt cinsinden bit düzeni 7: 01000000
Bayt cinsinden bit düzeni 6: 10011111
Bayt cinsinden bit düzeni 5: 00011111
Bayt cinsinden bit düzeni 4: 01010000
Bayt cinsinden bit düzeni 3: 11100101
Bayt cinsinden bit düzeni 2: 01100000
Bayt cinsinden bit düzeni 1: 01000001
Bayt cinsinden bit düzeni 0: 10001001

```

3. Gerek yapıların gerekse bileşimlerin, constructor ve destructor fonksiyonları olabilir. Aşağıdaki örnekte, **strtype** sınıfının yapı olarak nasıl kullanıldığını görüyoruz. Bu sınıfın, bir constructor fonksiyonu bir de destructor fonksiyonu bulunmaktadır.

```

#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

struct strtype {
    strtype(char *ptr);
    ~strtype();
    void show();
private:
    char *p;
    int len;
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr);
    p = (char *) malloc(len+1);
    if(!p) {

```

```

        cout << "Müllekte yer ayırma hatası\n";
        exit(1);
    }
    strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "p, serbest bırakılıyor\n";
    free(p);
}

void strtype::show()
{
    cout << p << " - uzunluk: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("Bu, bir deneme."), s2("C++'yi seviyorum.");
    s1.show();
    s2.show();

    return 0;
}

```

4. Bu programda, bir **double** sayı içerisindeki byte'ları teker teker göstermek için anonim bileşim kullanıyoruz. (**double** tipinde değişkenlerin 8 byte uzunluğunda olduğunu farz ediyoruz.)

```

// Bu programda anonim bileşim kullanılıyor.
#include <iostream>
using namespace std;

int main()
{
    union {
        unsigned char bytes[8];
        double value;
    };
    int i;
    value = 859345.324;

    // double içerisindeki baytlar gösteriliyor
    for(i=0; i<8; i++)
        cout << (int) bytes[i] << " ";
}

return 0;
}

```

Burada, **value** ve **bytes**'a normal değişkenler gibi erişildiğini görüyoruz. Onlara sanki bir bileşime ait değillermiş gibi davranıyoruz. Bu değişkenler bir anonim bileşimin parçası olarak declare edilmişlerdir, fakat adları, aynı noktada declare edilen diğer yerel değişkenlerle aynı alan seviyesindedir. Anonim bileşimlerin üyelerinin aynı alandaki diğer değişkenlerle aynı adı alamamalarının nedeni budur.

Aliştırmalar

1. Konu 2.1'de verilen `stack` sınıfını sınıf yerine yapı kullanarak yeniden yazın.
2. Bir tamsayının düşük ve yüksek anlamlı byte'larının yerlerini değiştirmek için `union` sınıfını kullanın (tamsayıları 16 bit farz edin, eğer bilgisayarınız 32 bitlik tam sayılar kullanıyorsa, bir `short int`'in byte'larının yerini değiştirin.)
3. Bir anonim bileşimin ne olduğunu ve normal bir bileşimden ne farkı olduğunu açıklayın.

2.6. In-Line Fonksiyonlar

Sınıflar konusuna kısa bir ara verelim ve yine alakalı bir başka konuya geçelim. C++'da gerçekçe çağrılmayan, fakat her çağrı noktası satır içerisinde genişleyen fonksiyonlar tanımlamak mümkün değildir. Bu fonksiyonlar, C benzeri parametreli makrolarla aynı prensiple çalışırlar. *In-line fonksiyonların* avantajı, çağrıma ve değer döndürme mekanizmalarıyla hiçbir alakaları olmamasıdır. Bu nedenle in-line fonksiyonları, normal fonksiyonlardan daha hızlı çalıştırılabilirler. (Hatalayalım, fonksiyonu çağrıran ve değer döndürmesini sağlayan makine kodutlarının fonksiyonun her çağrılığında işletilmesi belli bir zaman alır. Eğer bu fonksiyonların bir de parametreleri varsa, bu iş daha da uzun sürer.)

In-line fonksiyonların dezavantajı ise bu fonksiyonların çok büyük olduğu ve çok sık çağrıldığı durumlarda programımızın büyümesine neden olmasıdır. Bu nedenle genelde kısa fonksiyonları, in-line fonksiyonlar olarak deklare ediyoruz.

Bir in-line fonksiyonunu deklare etmek için, fonksiyon tanımının başına bir `inline` belirleyicisi koyarız. Aşağıdaki programda, bir in-line fonksiyonun nasıl deklare edildiğini göreceğiz:

```
// Bir in-line fonksiyon örneği
#include <iostream>
using namespace std;

inline int even(int x)
{
    return !(x%2);
}

int main()
{
    if(even(10)) cout << "10 çift bir sayıdır \n";
    if(even(11)) cout << "11 çift bir sayıdır \n";

    return 0;
}
```

Örnekte, argümanı çiftse `true` gönderen `even()` fonksiyonunu, in-line olarak deklare ettiğimiz anlamları şudur:

```
if(even(10)) cout << "10 çift bir sayıdır\n";
```

satırı fonksiyonel olarak aşağıdaki satırda eşittir:

```
if((!10&2)) cout << "10 çift bir sayıdır \n";
```

Bu örnek, **inline**'ın kullanımının önemli bir diğer özelliğine daha işaret etmektedir: **inline** fonksiyonlarını, ilk çağrılarından önce tanımlamalıyız. Eğer tanımlamazsa derleyici, onun satır içinde genişletilmesi gerektiğini bilemez. **even()**'ın **main()**'den önce tanımlanmasının nedeni de budur.

Parametreli makrolar yerine **inline**'ı kullanmamızın iki avantajı vardır. Bunlardan ilki, kısa fonksiyonları satır içinde genişletmek için bize daha yapısal bir yol sağlar. Örneğin, parametreli makro oluştururken, makronun her durumda düzgün bir şekilde satır içinde genişletilmesi için ekstra parantezlerin sıkça kullanılmasının gerektiğini kolayca unutabiliriz. In-line fonksiyonlarını kullanarak, bu tür problemleri önleyebiliriz.

İkinci avantaj ise şudur: Bir in-line fonksiyonu, bir makro genişletilmesine göre çok daha düzgün bir şekilde derleyici tarafından optimize edilebilir. C++ programcılar,其实te hiçbir olayda parametreli makroları kullanmazlar, bunun yerine **inline**'a güvenirler ve onu kullanırlar.

inline belirleyicisinin derleyici için bir *istem* olduğunu, bir komut olmadığını anlamamız gerekiyor. Eğer çeşitli sebeplerden dolayı, derleyici bu isteği yerine getiremezse, fonksiyon normal bir fonksiyon olarak derlenir ve **inline** isteği göz ardı edilir.

Derleyicinize bağlı olarak in-line fonksiyonlarına çeşitli sınırlamalar konulabilir. Örneğin bazı derleyiciler, fonksiyonun **statik** bir değişkeni, bir döngü bildirisini, bir **switch**'i veya bir **goto**'su varsa, ya da fonksiyon tekrar ediliyorsa, o fonksiyona in-line işlemi yapmayaçaktır. Derleyicinizin kullanım kılavuzunu inceleyin ve in-line fonksiyonlarına konulan siyi etkileyebilecek özel sınırlamalar olup olmadığına bakın.

HATIRLATMA Eğer herhangi bir in-line sınırlamasına uyulmazsa derleyici normal bir fonksiyon üretmekte serbesttir.

Örnekler

1. Sınıflara ait fonksiyonlar da dahil olmak üzere herhangi bir tipte fonksiyona in-line işlemi yapılabilir. Örnekte, **divisible()** üye fonksiyonu hızlı çalışması için in-line hale getiriyoruz. (Bu fonksiyon, ilk argümanı ikincisi tarafından tam bölünebiliyorsa **true** döndürür.)

```
// Üye bir fonksiyonun in-line hale getirilmesi.
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    samp(int a, int b);
    int divisible(); // kendi tanımı içerisinde in-line edilmiş
};
```

```

samp::samp(int a, int b)
{
    i = a;
    j = b;
}

/* Eğer i, j tarafından tam bölünebiliyorsa 1 döndürür.
   Bu üye fonksiyon, satır içinde genişletilmiştir.
*/
inline int samp::divisible()
{
    return !(i*j);
}

int main()
{
    samp ob1(10, 2), ob2(10, 3);
    // bu doğru
    if(ob1.divisible()) cout << "10, 2 ile bölünebilir\n";
    // bu yanlış
    if(ob2.divisible()) cout << "10, 3 ile bölünebilir\n";

    return 0;
}

```

2. Aşırı yüklenmiş bir fonksiyon da in-line haline getirilebilir. Örneğin, bu programda `min()` fonksiyonunu üç şekilde aşırı yükliyoruz. Her üçünde de fonksiyonu `inline` şeklinde declare ediyoruz.

```

#include <iostream>
using namespace std;

// min() üç şekilde aşırı yüklenmektedir.

// tamsayılar
inline int min(int a, int b)
{
    return a<b ? a : b;
}

// long'lar
inline long min(long a, long b)
{
    return a<b ? a : b;
}

// double'lar
inline double min(double a, double b)
{
    return a<b ? a : b;
}

int main()
{
    cout << min(-10, 10) << "\n";
    cout << min(-10.01, 100.002) << "\n";
    cout << min(-10L, 12L) << "\n";

    return 0;
}

```

Aliştırmalar

1. Bölüm 1'de **abs()** fonksiyonunu tamsayıların, long tamsayıların ve **double**'ların mutlak değerini bulacak şekilde aşağı yüklediniz. Bu programı, fonksiyonlar satır içinde genişletecek şekilde değiştirin.
2. Aşağıdaki fonksiyon derleyici tarafından neden in-line olarak tanınmayabilir?

```
void f1()
{
    int i;
    for(i=0; i<10; i++) cout << i;
}
```

2.7. Otomatik In-Line İşlemi

Eğer üye fonksiyonumuzun tanımı yeterince kısaysa, bu tanımı sınıf deklarasyonunun içine koyabiliriz. Bu şekilde hareket edilmesi, eğer mümkünse fonksiyonun otomatik olarak in-line fonksiyon haline gelmesine neden olacaktır. Fonksiyon, sınıf deklarasyonunun içinde tanımlanmışsa **inline** anahtar kelimesi artık gerekli değildir. (Fakat, kullanılmasında bir sakınca yoktur.) Örneğin, önceki konuda adı geçen **divisible()** fonksiyonunu, aşağıdaki gibi, otomatik olarak in-line hale getirebiliriz:

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

class samp {
    int i, j;
public:
    samp(int a, int b);

    /* divisible() burada tanımlanmaktadır ve otomatik olarak
     * in-line haline getirilir. */
    int divisible() { return !(i%j); }
};

samp::samp(int a, int b)
{
    i = a;
    j = b;
}

int main()
{
    samp ob1(10, 2), ob2(10, 3);
    // bu doğru
    if(ob1.divisible()) cout << "10, 2 ile bölünebilir\n";
    // bu yanlış
    if(ob2.divisible()) cout << "10, 3 ile bölünebilir\n";
    return 0;
}
```

Burada **divisible()**'a bağlı olan kod, **samp** sınıfının deklarasyonunun içinde bulunmaktadır. **divisible()**'a ait başka bir tanımlamaya gerek yoktur veya buna izin verilmez. **divisi-**

ble()'ın **samp**'in içinde tanımlamamız, bu fonksiyonun otomatik olarak bir in-line fonksiyon sayılmasına neden olmaktadır.

Fonksiyonları, sınıf deklarasyonlarının içinde tanımladıysak, bu fonksiyonların bir in-line fonksiyon olması mümkün değildir (çünkü bir sınırlamaya uyulmamış olur) ve otomatik olarak normal bir fonksiyona dönüştürülürler.

divisible()'ın **samp**'in içinde nasıl tanımlandığına dikkat edelim, özellikle de fonksiyonun gövdesini dikkatle inceleyelim. Fonksiyonun tamamı tek bir satırda bulunmaktadır. C++ programlarında bir fonksiyon, sınıf deklarasyonunda deklare edilmişse, bu tür tek satırlık bir kullanımla oldukça sık karşılaşılır. Bu kullanım şekli, deklarasyonun daha yoğun olmasına imkan verir. Fakat, **samp** sınıfı şu şekilde de yazılabilir:

```
class samp {
    int i, j;
public:
    samp(int a, int b);
    /* divisible() burada tanımlanmaktadır ve otomatik olarak
     * in-line haline getirilmektedir. */
    int divisible()
    {
        return !(i%j);
    }
};
```

Programın bu şeklinde **divisible()** fonksiyonunda standart satırbaşı stili kullanılmaktadır. Olaya derleyicinin tarafından bakarsak, yoğun stille standart stil arasında bir fark yoktur. Fakat, genelde yoğun stil, C++ programlarında, kısa fonksiyonlar, sınıf tanımlaması içinde tanımlandığında kullanılır.

"Normal" in-line fonksiyonlarına konulan kısıtlamalar, sınıf deklarasyonu içinde bulunan otomatik in-line fonksiyonları için de geçerlidir.

Örnekler

1. Sınıf içinde tanımlanan in-line fonksiyonlarının belki de en yaygın kullanımını constructor ve destructor fonksiyonlarının tanımlanmasıdır. Örneğin, **samp** sınıfını şu şekilde çok daha etkili bir şekilde tanımlayabiliriz:

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

class samp {
    int i, j;
public:
    // in-line constructor
    samp(int a, int b) { i = a; j = b; }
    int divisible() { return !(i%j); }
};
```

samp sınıfının içerisindeki **samp()** fonksiyonunun tanımlanması yeterlidir ve **samp()**'in başka bir yerde ayrıca tanımlanmasına gerek yoktur.

2. Bazen, otomatik in-line özelliğinin pek değerli olmadığı durumlarda da kısa fonksiyonlar, sınıf deklarasyonuna konulacaktır. Şu sınıf deklarasyonunu inceleyelim:

```
class myclass {
    int i;
public:
    myclass(int n) { i = n; }
    void show() { cout << i; }
};
```

Burada `show()` fonksiyonu otomatik olarak bir in-line fonksiyonuna dönüştürüller. Fakat, şunu bilmemiz gerekiyor: I/O işlemleri (genelde) CPU/bellek işlemlerine göre öyle yavaştır ki, fonksiyonun çağrıması için harcanan zamanın artık harcanmaması bazen fark edilmez bile. Hatta, C++ programlarında sınıf içerisinde deklare edilmiş bu tip küçük fonksiyonlarla sıkça karşılaşmamız hala mümkündür, çünkü bunun bir zararı yoktur.

Aliştırmalar

1. Konu 2.1, Örnek 1'deki `stack` sınıfını, otomatik in-line fonksiyonlarını uygun yerlerde kullanarak değiştirin.
2. Konu 2.2, Örnek 3'deki `strtype` sınıfını otomatik in-line fonksiyonlarını kullanarak değiştirin.

Pekiştirme Testi

Bu noktada, aşağıdaki alıştırmaları yapabiliyor ve soruları cevaplayabiliyor olmalısınız.

1. Constructor ve destructor nedir? Bu fonksiyonlar ne zaman çalıştırılırlar?
2. Ekranda çizgi çizen `line` (çizgi) adında bir sınıf oluşturun. Çizginin uzunluğunu `len` adında private bir tamsayı değişkende saklayın. `line`'in constructor fonksiyonunun çizgi uzunluğu için bir parametresi olsun. Constructor fonksiyonu uzunluğu saklasın ve çizgiyi çizsin. Eğer sisteminiz grafik işlemlerini desteklemiyorsa, çizgiyi * işaretini kullanarak oluşturun. İsterseniz, `line` için çizgiyi silen bir destructor hazırlayın.
3. Aşağıdaki program ekranda ne gösterir?

```
#include <iostream>
using namespace std;
int main()
{
    int i = 10;
    long l = 10000000;
    double d = -0.0009;
    cout << i << ' ' << l << ' ' << d;
    cout << "\n";
    return 0;
}
```

4. Konu 2.3, Alistırma 1'deki `area_cl`'den türeyen bir alt sınıf oluşturun. Bu sınıf `cylinder` adını verin. Bu sınıf bir silindirin yüzeyinin alanını versin. İpucu: Silindirin yüzey alanı $2 * \pi * R^2 + \pi * D * \text{yükseklik}$.
5. In-line fonksiyon nedir? Avantajları ve dezavantajları nelerdir?
6. Aşağıdaki programı tüm üye fonksiyonlar otomatik olarak in-line edilecek şekilde değiştirin:

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

class myclass {
    int i, j;
public:
    myclass(int x, int y);
    void show();
};

myclass::myclass(int x, int y)
{
    i = x;
    j = y;
}

void myclass::show()
{
    cout << i << " " << j << "\n";
}

int main()
{
    myclass count(2, 3);

    count.show();

    return 0;
}
```

7. Sınıf ile yapı arasındaki fark nedir?
8. Aşağıdaki program parçası geçerli midir?

```
union {
    float f;
    unsigned int bits;
};
```

Bütünleştirme Testi

Burada bir önceki bölümde öğrendiklerinizle bu bölümde öğrendiklerinizi ne kadar iyi bir araya getirebildiğiniz kontrol edilmektedir.

1. `prompt` adında bir sınıf oluşturun. Constructor fonksiyonuna, ekrandan bir tam-sayı girilmesini istediğiniz bir katar gönderin. Constructor, bu katarı gösterirsin ve ekrandan girilen sayıyı alsin. Bu değeri `count` adında bir private değişken-

- nin içerisinde saklayın. **prompt** tipinde bir nesne yok edildiğinde, kullanıcının girdiği sayı kadar bip sesi çıkarın.
2. Bölüm 1'de fitiince çeviren bir program oluşturmustunuz. Şimdi, aynı şeyi yapan bir sınıf oluşturun. Bu sınıf, fit cinsindeki değeri ve bu değerin inç cinsinden eşdeğerini saklasın. Sınıfin constructor'ına fit cinsindeki değeri gönderin ve constructor inç cinsindeki bu değeri göstersin.
 3. **dice** adında tamsayı tipinde private bir değişken içeren bir sınıf oluşturun. 1 ile 6 arasında sayılar üretmek için **roll()** adında, rasgele standart sayı üretten **rand()** fonksiyonunu kullanan bir fonksiyon oluşturun. Sonra **roll()**'un bu değeri göstermesini sağlayın.

www.cergokku.com

BÖLÜM 3

Sınıfları Daha Yakından inceleyelim

Nesne Atamak

Fonksiyonlara Nesne Aktarmak

Fonksiyonlardan Nesne Döndürmek

Arkadaş Fonksiyonlara Giriş

Bu bölümde sınıfları incelemeye devam edeceğiz. Nesne atamaları, fonksiyonlara nesne aktarılması ve fonksiyonlardan nesne döndürülmesi konularında daha fazla bilgi sahibi olacağız. Ayrıca, yeni ve önemli bir fonksiyon tipini daha öğreneceğiz.

Gözden Geçirme Testi

Daha ileri konulara geçmeden önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

- Aşağıda verilen sınıfların yapıcı ve yıkıcı fonksiyonlarının adları nelerdir?

```
class Widget {
    int x, y;
public:
    // ... yapıcı ve yıkıcı fonksiyonları yazın
};
```

- Yapıcı ve yıkıcı fonksiyonlar ne zaman çağrırlırlar?
- Aşağıda verilen temel sınıfın, **Mars** adlı bir türetilmiş sınıfından nasıl miras alınabileceğini gösterin.

```
class planet {
    int moons;
    double dist_from_sun;
    double diameter;
    double mass;
public:
    // ...
};
```

- Bir fonksiyonun satır içinde genişletilmesinin iki nedeni vardır. Bunlar nelerdir?
- In-line fonksiyonlara ilişkin iki kısıtlamayı sayınız.
- Aşağıda verilen sınıfa bakarak, **a**'ya 100 ve **c**'ye X değerlerini geçiren **ob** adlı bir nesnenin belirtiminin nasıl yapılacağını gösteriniz.

```
class sample {
    int a;
    char c;
public:
    sample(int x, char ch) { a = x; c = ch; }
    // ...
};
```

3.1. Nesne Atamak

Bir nesne aynı tipten başka bir nesneye atanabilir. Normalde, nesne diğer nesneye atandığında tüm veri üyelerinin bit bit kopyası alınmaktadır. Örnek olarak **o1** adındaki nesne **o2** adında diğer bir nesneye atandığında **o1**'e ait verilerin tümü **o2**'nin eşdeğer üyelerinin içine kopyalanır. Bunu aşağıdaki programda inceleyelim:

```
// Nesne atamalarına bir örnek.
#include <iostream>
using namespace std;

class myclass {
    int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
    myclass o1, o2;

    o1.set(10, 4);

    // o1'i o2'ye atı
    o2 = o1;
    o1.show();
    o2.show();

    return 0;
}
```

Burada **o1** nesnesi, kendisine ait olan **a** ve **b** üye değişkenlerine sırasıyla 10 ve 4 değerlerini koyar. Daha sonra, **o1**, **o2**'ye atanır. Bu, **o1.a**'nın **o2.a**'ya atanmasını ve **o1.b**'nin de **o2.b**'ye atanmasına neden olur. Programı çalıştırdığınızda, ekranda aşağıdaki sonucu göreceksiniz.

```
10 4
10 4
```

Bir nesneyi diğerine atadığımızda bu nesnelerin içerişindeki verilerin eşitlendiğini aklimızda tutun. İki nesne halen birbirinden ayrı durumdadır. Örneğin atama işleminden sonra **o1.a**'ya değer atamak için **o1.set()**'in çağrılması, **o2** ve ona ait **a** değeri üzerinde bir etki yapmaz.

Örnekler

1. Sadece aynı tipteki nesneler arasında atama işlemleri yapılabilir. Eğer nesneler aynı tipte değilse bu bir derleme hatasına neden olur. Üstelik tiplerin fiziksel olarak aynı olmaları da yeterli değildir, tip adları da aynı olmak zorundadır. Örneğin şu program doğru değildir:

```
// Bu programda bir hata var.
#include <iostream>
using namespace std;

class myclass {
    int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << "\n"; }
};
```

```

/* Bu sınıf myclass'a benzemektedir, fakat farklı bir sınıf adı
kullanmaktadır ve bu yüzden derleyiciye farklı bir tip olarak görünür.
*/
class yourclass {
    int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
    myclass o1;
    yourclass o2;
    o1.set(10, 4);
    o2 = o1; // HATA, nesneler aynı tipte değil
    o1.show();
    o2.show();
    return 0;
}

```

Bu durumda, **myclass** ve **yourclass** fiziksel olarak aynı olmasına rağmen tip adları farklı olduğundan, derleyici onlara farklı tipteymişler gibi davranışır.

2. Bir atama işlemi gerçekleştiğinde, bir nesneye ait tüm veriler diğer nesneye atanır. Bu, tipki diziler gibi bileşik verileri de içine alır. Örnek olarak, aşağıdaki **stack** örneğinde gerçekte sadece **s1**'e karakter gönderilir. Fakat, atama nedeniyle **s2**'nin **stck** dizisi de **a**, **b** ve **c** karakterlerini içerecektir.

```

#include <iostream>
using namespace std;

#define SIZE 10
// Karakterler için bir stack sınıfı bildiriminde bulunuluyor.
class stack {
    char stck[SIZE]; // Yığını tutar.
    int tos; // stack'ın üst indis'i
public:
    stack(); // constructor
    void push(char ch); // Yığına karakter itiliyor
    char pop(); // Yığından karakter çekiliyor
};

// Yığın başlatılıyor.
stack::stack()
{
    cout << "Yığın oluşturuluyor \n";
    tos = 0;
}

// Bir karakter iter.
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Yığın dolu\n";
        return;
    }
    stck[tos] = ch;
}

```

```

    tos++;
}

// Bir karakter çeker.
char stack::pop()
{
    if(tos==0) {
        cout << "Yığın boş\n";
        return 0; // Yığın boşsa sıfır döndürür
    }
    tos--;
    return stck[tos];
}

int main()
{
    // otomatik başlatılan iki yığın oluşturulur.
    stack s1, s2;
    int i;

    s1.push('a');
    s1.push('b');
    s1.push('c');

    // clone s1
    s2 = s1; // şimdi s1 ve s2 aynı

    for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
    for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

    return 0;
}

```

3. Bir nesneyi diğerine atarken dikkatli olmalısınız. Örnekte kısa `main()` fonksiyonundan sonra Bölüm 2'de geliştirilen `strtype` sınıfı verilmektedir. Program içerisinde hata bulmaya çalışın.

```

// Bu programda bir hata var.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype();
    void show();
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr);
    p = (char *) malloc(len+1);
    if(!p) {
        cout << "Belleğe yerleştirme hatası\n";
        exit(1);
    }
}

```

```

    strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "p serbest bırakılıyor \n";
    free(p);
}

void strtype::show()
{
    cout << p << " - uzunluk: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("Bu bir deneme."), s2("Ben C++'ı seviyorum.");
    s1.show();
    s2.show();
    // s1, s2'ye atanıyor -- bu, bir hataya neden oluyor
    s2 = s1;
    s1.show();
    s2.show();
    return 0;
}

```

Bu programdaki hata oldukça gizli. **s1** ve **s2** oluşturulduğunda her ikisi de katarlarını saklamak için bellekte yer kaplar. Her nesne için ayrılmış olan alanları gösteren işaretçi **p**'nin içerisinde saklanır. **strtype** sınıfına ait bir nesne yok edildiğinde bu bellek serbest bırakılır. Fakat, **s1**, **s2**'ye atandığında, **s2**'nin **p**'si şimdi, **s1**'in **p**'si ile aynı yeri göstermektedir. Böylelikle bu nesneler yok edildiğinde **s1**'in **p**'si tarafından işaret edilen bellek bölgesi *iki kere* serbest bırakılmış olur ve **s2**'nin **p**'si tarafından önceden işaret edilen bellek yeri ise *hicbir şekilde* serbest bırakılmaz.

Bu konuya başlarken şunu da belirtelim, gerçek bir program içerisinde meydana gelen bu tip problemler dinamik bellek ayırma sistemini bozacaktır ve hatta büyük olasılıkla programın kilitlenmesine neden olacaktır. Bir önceki örnekteki gibi, bir nesne diğerine atanırken daha sonra ihtiyacınız olabilecek bilgileri yok etmediğinizden emin olmalısınız.

Alıştırmalar

1. Bu programdaki hata nedir?

```

// Bu programda bir hata var.
#include <iostream>
using namespace std;

class cl1 {
    int i, j;
public:
    cl1(int a, int b) { i = a; j = b; }

```

```

    // ...
}

class cl2 {
    int i, j;
public:
    cl2(int a, int b) { i = a; j = b; }
    // ...
};

int main()
{
    cl1 x(10, 20);
    cl2 y(0, 0);
    x = y;
    // ...
}

```

2. Bölüm 2, Konu 2.1, Alıştırma 1 için oluşturduğumuz **queue** (kuyruk) sınıfını kullanarak bir kuyruğun diğerine nasıl atanabileceğini gösterin.
3. Önceki sorudaki **queue** sınıfının kuyruğu saklamak için dinamik olarak bellekte yer ayırması durumunda neden kuyruk bir diğerine atanamaz?

3.2. Fonksiyonlara Nesne Aktarmak

Nesneler, tipki diğer veri tiplerinin aktarılmasında olduğu gibi argümanlar halinde fonksiyonlara aktarılabilir. Fonksiyonun parametresi sınıf tipinde bildirilir ve sonra fonksiyonun çağrılışında bu sınıfa ait bir nesne argüman olarak kullanılır. Fonksiyona diğer veri tiplerinde olduğu gibi varsayılan olarak nesnelerin değerleri aktarılır.

Örnekler

1. Fonksiyonlara nesne aktarılmasıyla ilgili kısa bir örnek:

```

#include <iostream>
using namespace std;

class samp {
    int i;
public:
    samp(int n) { i = n; }
    int get_i() { return i; }
};

// o.i'nin karesi döndürülür.
int sqr_it(samp o)
{
    return o.get_i() * o.get_i();
}

int main()
{
    samp a(10), b(2);
    cout << sqr_it(a) << "\n";
}

```

```

cout << sqr_it(b) << "\n";
return 0;
}

```

Bu program, *i* tamsayısını içeren, **samp** adlı bir sınıf oluşturur. **sqr_it()** fonksiyonunun **samp** tipinde bir argümanı vardır ve bu fonksiyon nesneye ait *i* değerinin karesini döndürür. Bu program, ekranda 4 ve 100 değerlerini gösterecektir.

2. Belirtildiği gibi, C++'da nesneler de dahil olmak üzere tüm parametrelerin aksi söylenenmediği sürece değerleri aktarılır. Bunun anlamı şudur: argümanın bit bit kopyası yapılır ve bu kopya, fonksiyon tarafından kullanılan kopyadır. Sonuç olarak, fonksiyonun içerisinde nesneye yapılan değişiklikler kopyası alınan nesneci etkilemez. Bu durumu aşağıda inceleyelim:

```

/*
Unutmayın, nesnelerin de diğer parametreler gibi değerleri aktarılır. Böylece
fonksiyonların içerisindeki parametrelerde meydana gelen değişiklikler,
çağırma işleminde kullanılan nesneler üzerinde bir etki yapmaz.
*/
#include <iostream>
using namespace std;

class samp {
    int i;
public:
    samp(int n) { i = n; }
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};

/* o.i'ye kendisinin karesi konulur. Fakat sqr_it()'yi çağırma sırasında kullanılan
nesne bu değişiklikten etkilenmez.
*/
void sqr_it(samp o)
{
    o.set_i(o.get_i() * o.get_i());
    cout << " a'nın kopyası şu değerin i değerine sahiptir : " << o.get_i();
    cout << "\n";
}

int main()
{
    samp a(10);
    sqr_it(a); // a'nın değeri aktarılır
    cout << "Fakat, a.i main'in içerisinde değişikliğe uğramaz ";
    cout << a.get_i(); // 10 gösterir
    return 0;
}

```

Bu program şu şekilde bir çıkış verir:

```

a'nın kopyası şu değerin i değerine sahiptir : 100
Fakat, a.i main'in içerisinde değişikliğe uğramaz : 10

```

3. Tıpkı diğer değişken tipleri gibi, nesne adresleri de fonksiyonlara aktarılabilir, bu şekilde de çağrıma işleminde kullanılan argüman, fonksiyon tarafından değiştirile-

bilir. Aşağıdaki program bir önceki örneğin bunu yapacak şekilde değiştirilmesiyle oluşturulmuştur, bu program hatta `sqr_it()`'nin çağrılarında adresi kullanılan nesnenin değerini değiştirmektedir.

```
/*
Şimdi, nesnenin adresi sqr_it()'e aktarılıyor, fonksiyon, çağrıma işleminde
adresi kullanılan argümanın değerini değiştirebilir.
*/
#include <iostream>
using namespace std;

class samp {
    int i;
public:
    samp(int n) { i = n; }
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};

/* o.i'ye kendisinin karesi konuluyor. Bu, çağrıran argümanı etkiliyor.
*/
void sqr_it(samp *o)
{
    o->set_i(o->get_i() * o->get_i());
    cout << "a'nın kopyası şu değerin i değerine sahiptir : " << o->get_i();
    cout << "\n";
}

int main()
{
    samp a(10);
    sqr_it(&a); // a'nın adresi sqr_it()'e gönderiliyor
    cout << "Şimdi main()'in içerisindeki a değişti : ";
    cout << a.get_i(); // 100 gösterir
    return 0;
}
```

Bu program aşağıdaki sonucu verir:

```
a'nın kopyası şu değerin i değerine sahiptir : 100
Şimdi, main'in içerisindeki a değişti : 100
```

- Nesneler fonksiyonlara gönderilirken bu nesnelerin kopyalanının yapılması, yeni nesnelerin oluşturulması anlamına gelir. Nesnenin gönderildiği fonksiyon sonlandığında argümanın kopyası yok edilir. Bu durumda akılımıza iki soru gelmesi gerekiyor: Nesnenin yapıcısı kopyanın yapılışı sırasında çağrılıyor mu? Nesnenin yıkıcısı kopya yok edilirken çağrılıyor mu? Cevap ilk başta sizi biraz şaşırtabilir. Nesnenin kopyası oluşturulurken yapıcı fonksiyonu üçgenlilik. Üzerinde biraz düşünürsek bunun nedenini kolayca anlayabiliriz. Yapıcı fonksiyonu, genelde nesneyi hazır hale getirmek için kullanılır ve bu nedenle zaten mevcut bir nesnenin fonksiyona gönderilecek kopyasının oluşturulması sırasında çağrılmamalıdır. Bu fonksiyonun çağrılırsa nesnenin içeriğini değişecektir. Oysa biz bu nesnenin o anki durumunun fonksiyona gönderilmesini istiyoruz, başlangıçtaki durumunun değil.

Fakat fonksiyon sonlandığında ve kopya yok edildiğinde yıkıcı fonksiyonu çağrınlı. Bunun nedeni, nesnenin yok olması sırasında yapılması gereken bazı işlemlerin gerçekleştirilmesi gerektidir. Örneğin, kopya için serbest bırakılması gereken bazı bellek yerleri ayrılmış olabilir.

Özetlemek gerekirse, nesnenin kopyası oluşturulurken bu kopya fonksiyonun bir argümanı olarak kullanıldığından, yapıcı fonksiyonu çağrılmaz. Fakat kopya yok edilirken (bu genellikle fonksiyon bir değer döndürdüğünde alandan çıkışması şeklinde olur) yıkıcı fonksiyonu çağrınlı.

Aşağıdaki örneği incelersek bunu daha iyi anlayacağız:

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

class samp {
    int i;
public:
    samp(int n) {
        i = n;
        cout << "Oluşturuluyor\n";
    }
    ~samp() { cout << "Yok ediliyor\n"; }
    int get_i() { return i; }
};

// o.i'nin karesi döndürülür.
int sqr_it(samp o)
{
    return o.get_i() * o.get_i();
}

int main()
{
    samp a(10);
    cout << sqr_it(a) << "\n";

    return 0;
}
```

Bu fonksiyon aşağıdaki sonucu verir:

```
Oluşturuluyor
Yok ediliyor
100
Yok ediliyor
```

Sizin de göreceğiniz gibi, yapıcı fonksiyonu sadece bir kere, o da a'nın oluşturulması sırasında çağrırlı. Fakat yıkıcı iki kere çağrırlı. Bu çağrılarından bir tanesi a, sqr_it()'ye gönderildiğinde oluşturulan kopya içindır. Diğer ise a'nın kendisi içindir.

Argümanın kopyası olan nesneye ait yıkıcının fonksiyon bittiğinde çalıştırılması bazı problemlere yol açabilir. Örneğin, argüman olarak kullanılan nesne dinamik olarak bellekte yer kaplıyor ve yok edildiğinde bu belleği serbest bırakıyorsa, bu

nesnenin kopyasına ait yıkıcı çağrılığında aynı bellek bölgesini serbest bırakmaya çalışacaktır. Bu da kopyası oluşturulan nesneye zarar verecektir ve onu kullanılamaz hale getirecektir. (Örnek olarak bu bölümün en başındaki 2. Aşıtırmayı inceleyebilirsiniz.) Bu hataya karşı tedbir alın ve kopya nesneye ait yıkıcı fonksiyonun orijinal nesneyi etkileyeyecek yan etkilere yol açmadığını emin olun.

Tahmin edeceğiniz gibi, kopyaya ait yıkıcının gerçek nesnenin gerek duyabileceği verileri yok etmesi problemi, nesnenin kendisini göndermek yerine adresini göndererek çözülebilirsiniz. Adres gönderildiğinde yeni bir nesne oluşturulmaz ve bunun sonucu olarak da fonksiyon sona erdiğinde yıkıcı çağrılmaz. (Bir sonraki bölümde göreceğimiz gibi, C++ bu konuda çok çeşitli alternatifler sunmaktadır. Fakat bu soruna getirilebilecek en iyi çözümü, *copy constructor* (kopya yapıcısı) adındaki özel yapıcı tipini öğrendikten sonra kullanabileceğiz. Kopya yapıcları kullanarak nesnelere ait kopyaların tam olarak nasıl oluşturulacağını tanımlayabilirsiniz. (Kopya yapıclar konusu Bölüm 5'de ele alınmıştır.)

Aşıtırmalar

1. Konu 3.1, Örnek 2'deki **stack** örneğine, **stack** tipinde nesnelerin gönderildiği **showstack()** adında bir fonksiyon ekleyin. Bu fonksiyon yiğinin içeriğini göstersin.
2. Bildiğiniz gibi, nesneler fonksiyonlara gönderildiğinde bu nesnelerin kopyaları yapılır. Bu fonksiyon sona erdiğinde kopyanın yıkıcı fonksiyonu çağrılar. Bunu aklınızda bulundurarak aşağıdaki örnekteki hatayı bulun.

```
// Bu program bir hata içermektedir.
#include <iostream>
#include <cstdlib>
using namespace std;

class dyna {
    int *p;
public:
    dyna(int i);
    ~dyna() { free(p); cout << "serbest bırakılıyor \n"; }
    int get() { return *p; }
};

dyna::dyna(int i)
{
    p = (int *) malloc(sizeof(int));
    if(!p) {
        cout << "Bellkte yer ayırma hatası\n";
        exit(1);
    }
    *p = i;
}

// *ob.p'nin negatif değeri döndürülüyor
int neg(dyna ob)
{
```

```
    return -ob.get();
}

int main()
{
    dyna o(-10);
    cout << o.get() << "\n";
    cout << neg(o) << "\n";
    dyna o2(20);
    cout << o2.get() << "\n";
    cout << neg(o2) << "\n";
    cout << o.get() << "\n";
    cout << neg(o) << "\n";
    return 0;
}
```

3.3. Fonksiyonlardan Nesne Döndürmek

Tıpkı fonksiyonlara nesne gönderebildiğimiz gibi, fonksiyonlardan nesne de döndürebiliyoruz. Bunun için öncelikle fonksiyonu sınıf tipi döndürür şekilde bildirmeniz gereklidir. İkinci olarak da bu tipte bir nesneyi normal **return** deyimini kullanarak döndürürüz.

Fakat fonksiyonlardan nesne döndürülmesi konusunda anlaşılması gereken önemli bir nokta daha var: Bir nesne fonksiyon tarafından döndürüldüğünde, otomatik olarak, döndürülen değeri saklayan geçici bir nesne oluşturulur. Gerçekte fonksiyon tarafından döndürülen nesne budur. Değer döndürüldükten sonra bu nesne yok edilir. Bu geçici nesnenin yok edilmesi bazı durumlarda Örnek 2'de olduğu gibi beklenmedik yan etkilere neden olabilir.

Örnekler

1. Nesne döndüren fonksiyonlara bir örnek:

```
// Nesne döndürülüyor
#include <iostream>
#include <cstring>
using namespace std;

class samp {
    char s[80];
public:
    void show() { cout << s << "\n"; }
    void set(char *str) { strcpy(s, str); }
};

// samp tipinde bir nesne döndürülüyor
samp sput();
{
    char s[80];
    samp str;
    cout << "Bir áatar girin: ";
    cin >> s;
    str.set(s);
    return str;
}
```

```

int main()
{
    samp ob;
    // döndürülen nesne ob'a atanıyor
    ob = input();
    ob.show();
    return 0;
}

```

Bu örnekte **input()** fonksiyonu, **str** adında yerel bir nesne oluşturur ve sonra da klavyeden bir katar okur. Bu katar **str.s**'in içine kopyalanır ve sonra **str**, fonksiyon tarafından döndürülür. Nesne, döndürüldükten sonra **main()**'in içerisindeki **ob**'a atanır.

2. Fonksiyonlardan nesne döndürürken eğer bu nesnelerin yıkıcı fonksiyonları varsa dikkatli olmalısınız. Çünkü bu değer döndürülür döndürülmez, döndürülen nesne erim alanının dışına çıkar. Mesela diyelim ki, fonksiyon tarafından döndürülen nesnenin dinamik ayrılmış belleğini serbest bırakın bir yıkıcısı var. Bu bellek, döndürülen değerin atıldığı nesne onu hala kullanıyor da olsa serbest bırakılacaktır. Örnek olarak bir önceki programın hatalı hale getirilmiş şeklini inceleyelim:

```

// Nesnenin döndürülmesi ile ortaya çıkan bir hata.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class samp {
    char *s;
public:
    samp() { s = '\0'; }
    ~samp() { if(s) free(s); cout << " s serbest bırakılıyor\n"; }
    void show() { cout << s << "\n"; }
    void set(char *str);
};

// Katar yükleniyor.
void samp::set(char *str)
{
    s = (char *) malloc(strlen(str)+1);
    if(!s) {
        cout << "Bellekte yer ayırma hatası\n";
        exit(1);
    }
    strcpy(s, str);
}

// samp tipinde bir nesne döndürülüyor.
samp input()
{
    char s[80];
    samp str;
    cout << "Bir katar girin: ";
    cin >> s;
    str.set(s);
    return str;
}

```

```

int main()
{
    samp ob;
    // döndürulen nesne ob'a atanmaya çalışılıyor
    ob = input(); // Bu bir hata yol açar!!!!
    ob.show();
    return 0;
}

```

Program size şu ekran çıkışını verecektir:

```

Bir katar girin: Merhaba
s serbest bırakılıyor
s serbest bırakılıyor
Merhaba
s serbest bırakılıyor
Null pointer assignment (Sıfır işaretçi ataması)

```

samp'in yıkıcı fonksiyonunun üç kere çağrıldığını dikkat edin. Bu fonksiyon ilk olarak **input()** geri döndürüldüğünde **str** yerel nesnesinin erim alanından çıkışma siyla çağrılır. **~samp()**'in ikinci çağrıları **input()** tarafından döndürülen geçici nesnenin yok edilmesinde meydana gelir. Unutmayın, nesneler fonksiyondan döndürüldüğünde, otomatik olarak, döndürülen değeri saklayan (**size**) görünmeyecek bir nesne oluşturulur. Bu durumda nesne, fonksiyonun döndürüdüğü değer olan **str**'nin bir kopyasıdır. Sonuç olarak, fonksiyon sona erdikten sonra geçici nesnenin yıkıcıları çalıştırılır. Son olarak da **main()**'in içerisindeki **ob** nesnesinin yıkıcı program sonlandığında çağrılr.

Bu durumda sorun şudur: Yıkıcının ilk çalıştırılışında, **input()** tarafından alınan katar için ayrılan bellek yeri serbest bırakılır. Böylece **samp**'ın yıkıcısına yapılan diğer iki çağrı, zaten serbest bırakılmış olan bir dinamik bellek parçasını serbest bırakmaya çalışmaya kalmaz, aynı zamanda bellekte dinamik yer ayırma sistemini de yok eder; bunu, çıkan "Null pointer assignment." (Sıfır işaretçi ataması) mesajından da anlayabiliriz. Bu programı denedığınızda, derleyicinize, derleme için kullanılan bellek modeline vb göre hata mesajını görebilirsiniz veya göremeyebilirsiniz.

Bu örnektenden öğrenmemiz gereken şu: Nesneler fonksiyonlardan döndürüldüğünde, dönüş değerini etkilemeye çalışan geçici nesnenin kendi yıkıcı fonksiyonu da çağrılr. Böylelikle bu durumun tehlikeli olduğu nesneleri döndürmekten kaçınmalısınız. Bölüm 5'te öğreneceğiniz gibi, bu durumu halletmek için bir kopya yapıcı kullanmanız mümkündür.

Aliştırmalar

1. Bir nesnenin fonksiyondan döndürülürken tam olarak ne zaman oluşturulduğunu ve yok edildiğini göstermek için **who** adında bir sınıf oluşturun. **who**'ya ait yapıcının bir nesneyi tanımlamak içi kullanılabilecek olan bir karakter argümanı olsun. Bir yapıcının nesne oluştururken şuna benzer bir mesaj versin:

```
who #x oluşturuluyor
```

Burada **x**, nesnelerin indisidir. Bir nesne yok edilirken ise aşağıdakine benzer bir mesaj çıksın:

```
who #x yok ediliyor
```

Burada **x** yine indistir. Son olarak da, **who** tipinde bir nesne döndüren **make_who()** adında bir fonksiyon oluşturun. Nesnelerin her birine ayrı birer ad verin. Program tarafından verilen çıkışa dikkat edin.

2. Ayrılmış belleğin dinamik olarak yanlış bir şekilde serbest bırakılması durumu hariçinde, fonksiyondan nesne çağrımasının yanlış olacağı bir durum düşünün.

3.4. Arkadaş Fonksiyonlara Giriş

Bir fonksiyonun üyesi olmadığı bir sınıf ait private üyelerle erişim hakkını istedığınız zamanlar olacaktır. Bunun için C++ size arkadaş (friend) fonksiyonlarını sunmaktadır. Arkadaş fonksiyonları sınıf ait değildir, ama bu fonksiyonların o sınıf ait private elemlara erişim hakkı vardır.

Arkadaş fonksiyonlarının işimize yaradığı iki durum vardır: Operatörlerin üst üste yüklenmesi ve belli tiplerde I/O fonksiyonlarının oluşturulması. Arkadaş fonksiyonlarının bu iki şekilde nasıl kullanıldığını görmek için bir süre daha beklemeniz gerekecektir. Fakat, burada ele alacağımız üçüncü bir kullanım şekli daha var: Bir fonksiyonun *iki* veya *daha fazla* sınıfın private üyelerine erişmesinin istediği durum.

Bir arkadaş fonksiyonu düzgün ama üye olmayan bir fonksiyon olarak tanımlanır. Fakat bu fonksiyonun, kendisine arkadaş olduğu sınıf bildiriminin içerisinde **friend** anahtar kelimesi ile belirtilen bir prototipi mevcuttur. Bunun nasıl çalıştığını anlamak için aşağıdaki bu kısa örneği inceleyelim:

```
// Arkadaş fonksiyonlarına bir örnek.
#include <iostream>
using namespace std;
class myclass {
    int n, d;
public:
    myclass(int i, int j) { n = i; d = j; }
    // myclass'ın arkadaş fonksiyonunun deklare edilmesi
    friend int isfactor(myclass ob);
}
/* Arkadaş fonksiyonumun tanımı. Eğer n d'nin tam katı ise true değerini
döndürür. friend anahtar kelimesinin isfactor()'ün tanımlanması sırasında
kullanılmadığına dikkat edin.
*/
int isfactor(myclass ob)
{
    if(!(ob.n % ob.d)) return 1;
    else return 0;
}
```

```

int main()
{
    myclass ob1(10, 2), ob2(13, 3);
    if(isfactor(ob1)) cout << "10, 2'nin katıdır\n";
    else cout << "10, 2'nin katı değildir\n";
    if(isfactor(ob2)) cout << "13, 3'un katıdır\n";
    else cout << "13, 3'un katı değildir\n";

    return 0;
}

```

Bu örnekte **myclass** kendi yapıcı fonksiyonunu ve sınıf bildirimi içerisindeki arkadaş **isfactor()**'yı bildirmektedir. **isfactor()**, **myclass**'ın arkadaşı olduğundan **isfactor()**'un onun private üyelerine erişimi hakkı vardır. Bu, **isfactor()**'un içerisinde **ob.n** ve **ob.d**'ye doğrudan gönderme yapabilmesinin nedenidir.

Bir arkadaş fonksiyonunun arkadaş olduğu sınıfa ait olmadığını anlaşılması önemlidir. Böylece bir arkadaş fonksiyonunu nesne adını ve sınıf üyesi erişim operatörünü (nokta veya ok operatörü) kullanarak çağrırmamız mümkün değildir. Örneğin bir önceki örnek için aşağıdaki deyim yanlışır:

```
ob1.isfactor(); // yanlış; isfactor() üye bir fonksiyon değildir.
```

Halbuki arkadaşlar normal fonksiyonlar gibi çağrırlırlar.

Bir arkadaş fonksiyonu her ne kadar arkadaş olduğu sınıfın private elemanlarının bilgisine sahip olsa da onlara sadece bu sınıfa ait bir nesne üzerinden erişebilir. Yani **myclass**'ın **n** veya **d**'ye doğrudan gönderme yapabilen üç fonksiyonlarının aksine, bir arkadaş fonksiyonu bu değişkenlere sadece, arkadaş fonksiyonu içerisinde bildirilmiş veya arkadaş fonksiyonuna gönderilmiş bir nesne aracılığıyla erişebilir.

NOT

Bir önceki paragraf önemli bir mesaleyi gözlerimizin önüne seriyor. Üye fonksiyonlar private elemanlara doğrudan gönderme yaparlar, çünkü üye fonksiyonlar sadece, bu sınıfa ait bir nesneye bağlı olarak çalıştırılırlar. Böylece, üye fonksiyon private elemana gönderme yaptığından, derleyici bu private elemenin hangi nesneye ait olduğunu bilir. Bu bilgiyi üye fonksiyonun çağrılmış sırasında fonksiyona bağlanan nesneden alır. Fakat arkadaş fonksiyonu herhangi bir nesneye bağlı değildir, sadece sınıfın private elemanlarına erişim hakkına sahiptir. Böylece, arkadaş fonksiyonunun içerisinde, belirli bir nesne referans alınmadan private bir üye gönderme yapmanın bir anlamı yoktur.

Arkadaşlar sınıflara ait degillerdir, bu yüzden arkadaş oldukları sınıfa ait bir veya daha fazla nesne kendilerine gönderilecektir. Aynı şey **isfactor()** için de geçerlidir. Bu fonksiyona **ob** adında **myclass**'a ait bir nesne gönderiyoruz. Fakat **isfactor()**, **myclass**'ın arkadaşı olduğundan, **ob**'nin private elemanlarına erişebilir. **isfactor()**, **myclass**'a arkadaş olmasaydı **n** ve **d** **myclass**'in private üyeleri olduğu sürece **ob.d** veya **ob.n**'ye erişemeyecekti.

HATIRLATMA Arkadaş fonksiyonları üye değildir ve bir nesne oði tarafından nitelenemez. Normal bir fonksiyonla oynu şekilde çagnimalıdır.

Arkadaş fonksiyonlarının miras yoluyla aktarılması da mümkün değildir. Yani temel sınıfın arkadaş fonksiyonu türetilmiş sınıfın da arkadaþı değildir.

Bu fonksiyonlar hakkındaki bir diğer önemli nokta da arkadaş fonksiyonlarının birden fazla sınıfla arkadaş olabilmesidir.

Örnekler

1. Arkadaş fonksiyonunun sık karşılaşacaðınız (ve aynı zamanda oldukça iyi) bir başka kullanım şekli de farklı tipte iki sınıfın çeşitli değerlerinin karşılaştırılmasıdır. Örnek olarak aşağıdaki programı inceleyelim. Burada **car** (araba) ve **truck** (kamyon) adında iki sınıf oluşturuluyor. Bu sınıfların her biri, private değişkenlerinde gösterdikleri aracın hızını saklamaktadır:

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

class truck; // forward bildirim

class car {
    int passengers;
    int speed;
public:
    car(int p, int s) { passengers = p; speed = s; }
    friend int sp_greater(car c, truck t);
};

class truck {
    int weight;
    int speed;
public:
    truck(int w, int s) { weight = w, speed = s; }
    friend int sp_greater(car c, truck t);
};

/* Eğer car'ın (araba) hızı truck'tan (kamyon) fazlaysa pozitif gönderiz.
Hızlar aynıysa 0 döndürür. Eğer truck'in hızı car'dan büyükse negatif
döndürür.
*/
int sp_greater(car c, truck t)
{
    return c.speed - t.speed;
}
int main()
{
    int t;
    car c1(6, 55), c2(2, 120);
    truck t1(10000, 55), t2(20000, 72);

    cout << "c1 ve t1 karşılaştırılıyor:\n";
    t = sp_greater(c1, t1);
    if(t<0) cout << "Truck daha hızlı.\n";
}
```

```

else if(t==0) cout << "Car ve truck'ın hızları aynı.\n";
else cout << "Car daha hızlı.\n";

cout << "\n c2 ve t2 karşılaştırılıyor:\n";
t = sp_greater(c2, t2);
if(t<0) cout << "Truck daha hızlı.\n";
else if(t==0) cout << "Car ve truck'ın hızları aynı.\n";
else cout << "Car daha hızlı.\n";

return 0;
}

```

Bu program, **car** ve **truck** sınıflarının arkadaş fonksiyonu olan **sp_greater()** fonksiyonunu içermektedir. Belirtildiği gibi, bir fonksiyon iki veya daha fazla sınıf'a arkadaş olabilir. Bu fonksiyon, **car** nesnesi **truck** nesnesinden daha hızlı gidiyorsa pozitif, hızları eşitse 0 ve **truck** daha hızlı gidiyorsa negatif döndürür.

Bu program C++'ın önemli bir yazım elemanını göstermektedir: *forward bildirim* (bu aynı zamanda *forward referans* şeklinde de adlandırılır). **sp_greater()**, hem **car** hem de **truck** sınıfının parametrelerini aldığından, **sp_greater()**'i bu iki sınıf'a dahil etmeden önce sınıfları deklare etmek mantıken imkansızdır. Sonuç olarak derleyiciye sınıfı gerçekte deklare etmeden sınıf adını söylemenin bir yolu olmalıdır. Buna forward bildirim adı verilir. C++'da derleyiciye tanımlayıcının sınıf adı olduğunu söylemek için, sınıf adının ilk kullanımından önce aşağıdaki gibi bir satır kullanın:

```
class sınıf-adı;
```

Örneğin bir önceki programda forward bildirimini şu şekildeydi:

```
class truck;
```

Şimdi **truck**, **sp_greater()**'ın arkadaş bildiriminde bir derleme hatası meydana getirmeden kullanılabilir.

2. Bir fonksiyon bir sınıfın üyesi ve bir diğerinin de arkadaşı olabilir. Örneğin burada bir önceki örnek, **sp_greater()**, **car**'ın üyesi ve **truck**'ın arkadaşı olacak şekilde yeniden yazılmıştır:

```

#include <iostream>
using namespace std;

class truck; // bir forward bildirim

class car {
    int passengers;
    int speed;
public:
    car(int p, int s) { passengers = p; speed = s; }
    int sp_greater(truck t);
};

class truck {
    int weight;
    int speed;
};

```

```

public:
    truck(int w, int s) { weight = w, speed = s; }
    // kapsam çözümleme operatörünün yeni kullanımına dikkat edin.
    friend int car::sp_greater(truck t);
};

/* Eğer car'ın (araba) hızı truck'tan (kamyon) fazlaysa pozitif gönderir.
Hızlar aynıysa 0 döndürür. Eğer truck'in hızı car'dan büyükse negatif
döndürür.
*/
int car::sp_greater(truck t)
{
    /* sp_greater() car'a ait olduğu sürece, sadece bir truck nesnesi ona
    gönderilmelidir. */
    return speed-t.speed;
}
int main()
{
    int t;
    car c1(6, 55), c2(2, 120);
    truck t1(10000, 55), t2(20000, 72);

    cout << " c1 ve t1 karşılaştırılıyor:\n";
    t = c1.sp_greater(t1); // car'a ait bir fonksiyonmuş gibi uyandırılıyor
    if(t<0) cout << "Truck daha hızlı.\n";
    else if(t==0) cout << "Car ve truck'in hızları aynı.\n";
    else cout << "Car daha hızlı.\n";

    cout << "\n c2 ve t2 karşılaştırılıyor:\n";
    t = c2.sp_greater(t2); // car'a ait bir fonksiyonmuş gibi uyandırılıyor
    if(t<0) cout << "Truck daha hızlı.\n";
    else if(t==0) cout << "Car ve truck'in hızları aynı.\n";
    else cout << "Car daha hızlı.\n";

    return 0;
}

```

Erim alanı çözümleme operatörünün (scope resolution operator) **truck** sınıfının bildirimindeki arkadaş bildiriminde ortaya çıkan yeni kullanım şekline dikkat edin. Burada derleyiciye **sp_greater()** fonksiyonunun **car** sınıfına ait olduğunu söylemek için bu operatörü kullanıyoruz.

Erim alanı çözümleme operatörünün nasıl kullanılacağını kolayca hatırlamanın yolu şu: Üye adı, onu takip eden erim alanı çözümleme operatörü ve onu takip eden sınıf adı sınıfı ait bu üyesi tam olarak belirler.

Hatta, sınıfı ait bir üyeseye gönderme yaparken bu üyenin adını tam olarak belirtmemiz yanlış olmaz. Fakat nesne, üye fonksiyonları çağrılmak veya üye değişkenlere erişmek için kullanıldığında tam adın kullanılması gereksizdir ve bu çok seyrek yapılır. Örneğin,

```
t = c1.sp_greater(t1);
```

aşağıdaki gibi, (gereksiz olan) erim alanı çözümleme operatörü ve **car** sınıf adı kullanılarak yazılabilir:

```
t = c1.car::sp_greater(t1);
```

Fakat **c1**, **car** tipinde bir nesne olduğu sürece derleyici zaten **sp_greater()**'ın **car** sınıfına ait olduğunu bilir. Bu da sınıfın tam olarak tanımlamamızı gereksiz hale getirir.

Aliştırmalar

- pr1 ve pr2 adlarında iki sınıfın tek bir yazıcıyı paylaştığı bir durum düşünün. Programınızın diğer bölümlerinin de bu iki sınıfın herhangi biri tarafından yazıcının kullanılmış kullanılmadığını bilmesi gerekiyor. **inuse()** adında bir fonksiyon oluşturun. Bu fonksiyon, yazıcı bu sınıflardan herhangi tarafından kullanılıyorsa **true** ve aksi halde **false** göndersin. Fonksiyon hem **pr1**'in hem de **pr2**'nin arkadaşı olsun.

```
class pr1 {
    int printing;
    // ...
public:
    pr1() { printing = 0; }
    void set_print(int status) { printing = status; }
    // ...
};

class pr2 {
    int printing;
    // ...
public:
    pr2() { printing = 0; }
    void set_print(int status) { printing = status; }
    // ...
};
```

Pekiştirme Testi

Daha ileri konulara geçmeden önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

- Bir nesnenin diğer bir nesneye atanması için hangi gereksinim karşılanmalıdır?
- Size aşağıda **samp** sınıfının bir bölümü verilmektedir:

```
class samp {
    double *p;
public:
    samp(double d) {
        p = (double *) malloc(sizeof(double));
        if(!p) exit(1); // bellekte yer ayırma hatası
        *p = d;
    }
    ~samp() { free(p); }
    // ...
};

// ...
samp ob1(123.09), ob2(0.0);
// ...
```

```
ob2 = ob1;
```

ob1'in ob2'ye atanması sonucu ne gibi bir problem ortaya çıkmaktadır?

3. Size aşağıdaki sınıf veriliyor,

```
class planet {
    int moons;
    double dist_from_sun; // mil cinsinden
    double diameter;
    double mass;
public:
    //...
    double get_miles() { return dist_from_sun; }
};
```

light() adında bir fonksiyon oluşturun. Bu fonksiyonun argümanı **planet** tipinde bir nesnedir ve bu fonksiyon ışığın güneşten gezegene gelmesinin kaç saniye süredüğünü döndürür. Işığın saniyede 186,000 mil hareket ettiğini ve **dist_from_sun**'m mil cinsinden belirlendiğini varsayıñ.

4. Bir nesnenin adresi bir fonksiyona argüman olarak gönderilebilir mi?
5. **stack** sınıfını kullanarak **loadstack()** adında bir fonksiyon yazın. Bu fonksiyon alfabetin harfleriyle (a-z) yüklü bir yiğin döndürsün. Bu yiğini, çağrıma rutinindeki başka bir nesneye atayın ve alfabeyi içerdigini ispatlayın. Yiğinin büyüklüğünü alfabetin sıgacığı büyülükle getirin.
6. Bir fonksiyona nesne gönderirken veya fonksiyondan nesne döndürürken neden dikkatli olmanız gerektiğini açıklayın.
7. Arkadaş fonksiyonu nedir?

Bütünleştirme Testi

Burada bir önceki bölümle bu bölümde öğrendiklerinizi ne kadar iyi bir araya getirebildiğiniz kontrol edilmektedir.

1. Fonksiyonlar parametrelerinin sayısı veya tipi değişik olduğu sürece aşırı yüklenebilir. Pekiştirme Testi Aşkırmı 5'deki **loadstack()**: **upper** adında bir tam sayı parametre alacak şekilde üst üste yükleyin. Fonksiyonun üst üste yüklenmiş halinde **upper** 1 değerindeyse, yiğini büyük harflerle yükleyin. Aksi halde yiğini küçük harflerle yükleyin.
2. Konu 3.1, Örnek 3'te verilen **strtype** sınıfını kullanarak bir arkadaş fonksiyonu ekleyin. Bu fonksiyonun argümanı **strtype** tipinde bir nesneyi gösteren bir işaretçi olsun ve bu fonksiyon, bu nesne tarafından işaret edilen katara bir işaretçi döndürsün. Yani, bu fonksiyon **p**'yi döndürsün. Bu fonksiyona **get_string()** adını verin.

3. Deneme: Türetilmiş sınıfa ait bir nesne, aynı türetilmiş sınıfa ait bir başka nesneye atandığında, temel sınıfa bağlı veri de kopyalanır mı? Bunu öğrenmek için aşağıdaki iki sınıfı kullanın ve ne olduğunu gösteren bir program yazın.

```
class base {  
    int a;  
public:  
    void load_a(int n) { a = n; }  
    int get_a() { return a; }  
};  
  
class derived : public base {  
    int b;  
public:  
    void load_b(int n) { b = n; }  
    int get_b() { return b; }  
};
```

BÖLÜM 4

Diziler, İşaretçiler ve Başvurular

Nesne Dizileri

Nesnelere İşaretçi Kullanmak

this İşaretçisi

new ve delete Kullanmak

new ve delete ile İlgili Ek Bilgiler

Başvurular

Nesnelere Başvuru Geçirme

Döndürülen Başvurular

Bağımsız Başvurular ve Sınırlamalar

Bu bölüm, nesne dizileri, nesnelere işaretçi gibi bazı önemli konuları ele almaktadır. Bölüm sonunda C++'ın en önemli parçalarından olanı başvurulardan bahsedilecektir. Başvuru C++'ın birçok yeniliğine temel olan bir yapıdır. Bu yüzden dikkatli bir biçimde okumayı sürdürün.

Gözden Geçirme Testi

Bu bölüme başlamadan önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

1. Bir nesne diğerine atandığı zaman tam olarak hangi işleyiş meydana gelir?
2. Bir nesne diğerine atandığı zaman herhangi bir olumsuzluk veya yan etki meydana gelebilir mi?
3. Bir nesne bir fonksiyonun argümanı olarak gönderildiğinde o nesnenin bir kopyası oluşturulur. Kopya nesnenin constructor (yapıldırma, yapıcı) ya da destructor (yok edici) fonksiyonu çağrılr mı?
4. Normalde nesneler fonksiyonlara bir değer olarak gönderilir. Bu da, fonksiyonun içindeki kopyaya ne oluyorsa bunun, çağrıda argümanı etkilemesi beklenmez anlamına gelir. Bu kuralı bozan bir şey varsa buna bir örnek verin.
5. Size aşağıdaki sınıf veriliyor. **Summation** tipinde bir nesne döndüren **make_sum()** adında bir fonksiyon oluşturun. Bu fonksiyon kullanıcıdan bir sayı istesin ve bu değeri taşıyan bir nesne oluştursun. Daha sonra onu çağrıran yordama döndürsün.

Fonksiyonun çalıştığını gösterin.

```
class summation {
    int num;
    long sum; // num'ların toplamı
public:
    void set_sum(int n);
    void show_sum() {
        cout << num << " toplam " << sum << "\n";
    }
};

void summation::set_sum(int n)
{
    int i;
    num = n;
    sum = 0;
    for(i=1; i<=n; i++)
        sum += i;
}
```

6. Bir önceki soruda **set_sum()** fonksiyonunun **summation** sınıf deklarasyonun içerisinde yer almadığını görüyoruz. Bunun bazı derleyiciler için neden gerekli olabileceği dair bir sebep gösterin.

7. Aşağıda bir sınıf daha verilmiştir. `myclass` tipinde bir parametre alan ve `num` negatif ise true pozitif ise false değeri döndüren `isneg()` adında bir arkadaş fonksiyonun nasıl ekleneceğini gösterin.

```
class myclass {
    int num;
public:
    myclass(int x) { num = x; }
};
```

8. Bir arkadaş fonksiyon birden fazla sınıfla arkadaş olabilir mi?

4.1. Nesne Dizileri

Daha önce belirttiğim gibi nesneler de, diğer tüm değişkenlerle aynı özellikleri ve yetenekleri olan bir tür değişkendir. Bu sebeple nesnelerin diziler halinde kullanılmasında da hiçbir sakınca yoktur. Bir dizi nesnenin deklare şekli de diğer herhangi bir değişken dizisinin deklare筷linden farklı değildir. Açıkçası, biraz daha ileri gidersek, nesne dizilerine erişim biçimini, diğer tipteki değişken dizilerinininkine tamamen aynıdır.

Örnekler

1. İşte size nesne dizileri ile ilgili bir örnek.

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

class samp {
    int a;
public:
    void set_a(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob[4];
    int i;

    for(i=0; i<4; i++) ob[i].set_a(i);
    for(i=0; i<4; i++) cout << ob[i].get_a() << endl;
    cout << "\n";
    return 0;
}
```

Bu program, `samp` tipinde nesnelerden oluşan dört elemanlı bir dizi oluşturuyor. Daha sonra her elemana `a` ile birlikte 0 ile 3 arasında değerler yükliyor. Üye fonksiyonların her bir dizi elemanına nasıl bağlı şekilde çağrıldığını dikkat edin. Bu örnekte `ob` olan dizi ismi indekslenmiştir. Daha sonra üyeye erişim operatörünü kullanmış ve bunu takiben çağrılan üye fonksiyonun ismi verilmiştir.

2. Eğer sınıf tipi bir yapılandırcı içeriyorsa dizideki nesneler hazırlanabilir.(oluşturulma esnasında görev yüklenebilir). Örneğin, burada **ob** dizisi elemanları hazırlanıyor.

```
// Diziyi hazırlama.
#include <iostream>
using namespace std;

class samp {
    int a;
public:
    samp(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob[4] = { -1, -2, -3, -4 };
    int i;
    for(i=0; i<4; i++) cout << ob[i].get_a() << ' ';
    cout << "\n";

    return 0;
}
```

Bu program ekranda **-1 -2 -3 -4** yazısını gösterir. Bu örnekte -1'den -4'ye kadar olan değerler **ob** yapılandırcı fonksiyonuna geçirilir.

Programdaki nesne hazırlama şekli aşağıdaki kinin kısaltılmış halidir. (İlk olarak bölüm 2'de gösterilmiştir)

```
samp ob[4] = { samp(-1), samp(-2), samp(-3), samp(-4) };
```

Aşında programda kullanılan dizim nesne hazırlamanın en bilinen şeklidir.(Daha sonra göreceksiniz ki bu tipteki söz dizimi, yapılandırcıları sadece bir argüman içeren dizilerde geçerlidir.)

3. Çok boyutlu nesne dizileri de kullanabilirisiniz. İşte size 2 boyutlu bir nesne dizisi oluşturan ve bu nesneleri hazırlayan bir program.

```
// İki boyutlu bir nesne dizisi oluşturur.
#include <iostream>
using namespace std;

class samp {
    int a;
public:
    samp(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob[4][2] = {
        { 1, 2 },
        { 3, 4 },
        { 5, 6 },
        { 7, 8 }
    };
}
```

```

int i;
for(i=0; i<4; i++) {
    cout << ob[i][0].get_a() << ' ';
    cout << ob[i][1].get_a() << "\n";
}
cout << "\n";

return 0;
}

```

Bu program aşağıdakini gösterir.

```

1 2
3 4
5 6
7 8

```

4. Sizinde bildiğiniz gibi, bir yapılandırıcı, bir argümandan daha fazlasını alabilir. Yapılandırıcısı birden daha fazla argüman içeren nesne dizilerini hazırlarken, daha önce belirtilen alternatif hazırlama dizimini kullanmalısınız. Gelin bir örnek ile açıklayalım:

```

#include <iostream>
using namespace std;

class samp {
    int a, b;
public:
    samp(int n, int m) { a = n; b = m; }
    int get_a() { return a; }
    int get_b() { return b; }
};

int main()
{
    samp ob[4][2] = {
        samp(1, 2), samp(3, 4),
        samp(5, 6), samp(7, 8),
        samp(9, 10), samp(11, 12),
        samp(13, 14), samp(15, 16)
    };
    int i;

    for(i=0; i<4; i++) {
        cout << ob[i][0].get_a() << ' ';
        cout << ob[i][0].get_b() << "\n";
        cout << ob[i][1].get_a() << ' ';
        cout << ob[i][1].get_b() << "\n";
    }

    cout << "\n";

    return 0;
}

```

Bu örnekte **samp**'in yapılandırıcısı iki argüman alır. Burada **ob** dizisi, **samp**'in yapılandırıcısına doğrudan çağrı kullanarak **main()** içerisinde deklare edilmiş ve hazırlanmıştır. Bu gereklidir çünkü standart C++ biçimde virgülle ayrılmış listede, bir

seferde ancak bir argümana izin verir. Örneğin listedeki her bir girdi için iki(veya daha fazla) argüman belirtmenin yolu yoktur. Bu sebeple kısaltılmış hazırlama söz diziminin yerine uzun olanı kullanmak zorundasınız.

NOT

Nesne bir argüman olsa bile uzun olan hazırlama yöntemini her zaman kullanabilirsiniz. Kısa yöntem bu durumda biraz daha kullanışlı olduğu için kullanıldı.

Bir önceki programın çıktısı:

```
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
```

Aliştırmalar

1. Aşağıdaki sınıf deklarasyonunu kullanarak on elemanlı bir dizi oluşturun ve **ch** birimini A'dan J'ye kadar değerler vererek hazırlayın. Dizinin bu değerleri taşıma anlamında davranışını gösterin.

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

class letters {
    char ch;
public:
    letters(char c) { ch = c; }
    char get_ch() { return ch; }
};
```

2. Verilen sınıf bildirimini kullanarak 10 elemanlı bir dizi oluşturun. Num'u 1'den 10'a kadar değerler vererek hazırlayın. Ardından **sqr**'yi, **num**'un karesi şeklinde gösterin.

```
#include <iostream>
using namespace std;
class squares {
    int num, sqr;
public:
    squares(int a, int b) { num = a; sqr = b; }
    void show() { cout << num << ' ' << sqr << "\n"; }
};
```

3. Aliştırma 1'deki hazırlama bölümünü, uzun hali kullanacak şekilde değiştirin. (Yani hazırlama list'indeki **letters** yapılandırıcısını değiştirin)

4.2. Nesnelere işaretçi kullanmak

Bölüm 2'de gördüğümüz gibi nesnelere işaretçiler vasıtasıyla da erişebiliriz. Bildiğiniz gibi nesnelere işaretçi kullanıldığında, nesnenin üyelerine başvurular (.) ile değil (->) operatörü ile gerçekleştirilir.

Düzen veri tipleri için geçerli olan işaretçi işlemleri nesne işaretçileri için de geçerlidir: Bu nesnenin tipine bağlı olarak gerçekleştirilir. Örneğin bir nesne işaretçisi bir arttırmışında bir sonraki nesneyi gösterir. Yine bir nesne işaretçisi bir eksiltildiğinde bir önceki nesneyi gösterir.

Örnek

- İste size bir nesne işaretçisi işlemi

```
// Nesnelere işaretçiler.
#include <iostream>
using namespace std;

class samp {
    int a, b;
public:
    samp(int n, int m) { a = n; b = m; }
    int get_a() { return a; }
    int get_b() { return b; }
};

int main()
{
    samp ob[4] = { samp(1, 2), samp(3, 4), samp(5, 6), samp(7, 8) };
    int i;
    samp *p;
    p = ob; // Dizinin başlangıç adresini al
    for(i=0; i<4; i++) {
        cout << p->get_a() << ' ';
        cout << p->get_b() << "\n";
        p++; // Bir sonraki nesneye geç
    }
    cout << "\n";
    return 0;
}
```

Bu program şunu gösterir.

```
1 2
3 4
5 6
7 8
```

Cıktıstan görüldüğü gibi p her arttırmışında dizideki bir sonraki nesneyi gösteriyor.

Aliştırmalar

1. Örnek 1'deki programı **ob** dizisinin içeriğini tersten gösterecek şekilde yeniden yazın.
2. Küsim 4.1 Örnek 3'teki programı iki boyutlu diziye işaretçi ile erişilebilecek şekilde değiştürün. İpucu: C++'da C'de olduğu gibi, bütün diziler devamlı olarak saklanırlar. Soldan sağa, küçükten büyüğe doğru...

4.3. this İşaretçisi

C++ **this** adı verilen yeni bir işaretçi içerir, **this**, çağrıldığı zaman kendiliğinden üye fonksiyona geçirilen bir işaretçidir. Ayrıca çağrıyı üreten nesneye de işaretçidir. Örneğin

```
ob.f1(); // ob'nin bir nesne olduğunu farkedin.
```

Fonksiyon **f1()** kendiliğinden **ob**'ye bir işaretçi geçer. Bu da çağrıyı gerçekleştiren nesnedir. Bu işaretçi **this** olarak bilinir. Sadece üye fonksiyonlarının **this** işaretçisi ile geçirildiğini kavramak önemlidir. Örneğin bir arkadaşın **this** işaretçisi yoktur

Örnek

1. Gördüğünüz gibi bir üye fonksiyon sınıfın diğer bir üyesine başvurursa, bu işi öyle doğrudan gerçekleştirir ki üyenin bir sınıf veya nesne özelliği olduğunu denetlemez. Örneğin, basit bir iç sınıf yaratan kosa bir programı inceleyelim:

```
// this işaretçisini gösterir
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
    char item[20];
    double cost;
    int on_hand;
public:
    inventory(char *i, double c, int o)
    {
        strcpy(item, i);
        cost = c;
        on_hand = o;
    }
    void show();
};

void inventory::show()
{
    cout << item;
    cout << ": $" << cost;
    cout << " Elde: " << on_hand << "\n";
}
```

BÖLÜM 4: Diziler, İşaretçiler ve Başvurular

```

int main()
{
    inventory ob("wrench", 4.95, 4);

    ob.show();

    return 0;
}

```

Gördüğünüz gibi yapıcı **inventory()** ve üye fonksiyon **show()** içerisinde üye değişkenleri olan **item**, **cost** ve **on_hand()**'e doğrudan erişilmiştir. Bu, üye fonksiyonun sadece nesne ile birlikte çağrılabilmesinden kaynaklanır. Böylece derleyici hangi nesnenin verisine başvurulduğunu bilir.

Yine de bundan daha da açıklayıcı bir anlatım yapılabilir. Bir üye fonksiyon çağrılarında çağrıyı gerçekleştiren nesneye kendiliğinden bir **this** işaretçisi ile geçirilir. Sonuç olarak bir önceki program değiştirilerek yeniden yazılabilir.

```

// this işaretçisini gösterir
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
    char item[20];
    double cost;
    int on_hand;
public:
    inventory(char *i, double c, int o)
    {
        strcpy(this->item, i); // Üyelere
        this->cost = c; // bu işaretçi üzerinden
        this->on_hand = o; // eriş
    }
    void show();
};

void inventory::show()
{
    cout << this->item; // Üyelere erişmek için bunu kullanın
    cout << ": $" << this->cost;
    cout << " On hand: " << this->on_hand << "\n";
}

int main()
{
    inventory ob("wrench", 4.95, 4);
    ob.show();
    return 0;
}

```

İşte burada üye değişkenlerine **this** işaretçisi üzerinden erişiliyor. Böylelikle **show()** içerisindeki bu iki söz dizimi de aynı manayı taşır:

```

cost = 123.23;
this->cost = 123.23;

```

Aşında ilk dizim ikincisine göre biraz daha zayıf ama kısa bir gösterimdir.

İkinci kullanım daha kolay olduğundan, C++ programcılar gösterildiği gibi üye fonksiyonlara erişmek için **this** işaretçisini kullanacaktır. Fakat kısa dizimin de neyi ifade ettiğini bilmek önemlidir.

this işaretçisinin çeşitli kullanım alanları vardır. Bunların içerisinde operatörlere aside yüklemeye yardımcı olması da vardır. Bu kullanım bölüm 6'da ayrıntılıyla ele alacağız. Simdilik normalde anlamamız gereken önemli husus bütün üye fonksiyonlarının çağrılan nesneye bir işaretçi ile geçirilmesidir.

Aliştirma

- Size aşağıdaki program veriliyor. **this** işaretçi başvurularını açık hale getirmek için uygun olan bütün başvuruları sınıf üyelerine dönüştürün.

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

class myclass {
    int a, b;
public:
    myclass(int n, int m) { a = n; b = m; }
    int add() { return a+b; }
    void show();
};
void myclass::show()
{
    int t;
    t = add(); // üye fonksiyonu çağır.
    cout << t << "\n";
}

int main()
{
    myclass ob(10, 14);
    ob.show();
    return 0;
}
```

4.4. new ve delete'i Kullanmak

Şimdiye kadar bellekte bir yer ayırmak gerekliliği olduğu durumlarda **malloc()**'u kullanıyoruz. Ayırdığınız yeri geri vermek içinde **free()**'yi kullanıyoruz. Bunlar elbette standart C'nin yer ayırma fonksiyonlarıdır. Bu fonksiyonlar C++'da kullanılsa da C++ bu tür bellek işlemleri için daha güvenli ve yapısal imkanlar sunar. C++'da belegi **new** kullanarak ayıralabilir ve **delete** kullanarak geri verebilirsiniz. Bu operatörlerin genel kullanım formu aşağıdaki gibidir

```
p-var = new type;
delete p-var;
```

Buradaki *type*, belkete yer ayırmak istediğiniz nesnenin tipidir. *P-var* ise o tipe olan işaretçidir. *new* operatörü, *type* ile tipi belirtilen nesneyi taşıyacak kadar genişliği olan, dinamik olarak ayrılmış belleğe bir işaretçi döndürür. *delete* ise ihtiyaç kalmadığında o belleği geri verir. *delete* sadece daha önce *new* ile ayrılmış belleğin işaretçisi ile çağrılabılır. Eğer *delete*'i yanlış bir işaretçi ile kullanırsanız yer ayırma sistemi yok edilir ve büyük ihtimalle programınız kilitlenir.

Yer ayırma işleminin gerçekleşmesi için yeteri kadar bellek bulunmuyorsa şu iki durumdan biri meydana gelir. Ya *new* boş bir işaretçi döndürür ya da bir ayrıcalıklı durum meydana gelir. (ayrıcalıklı durumlar ve onların işlenmesi komularından daha sonra bahsedeceğiz. Ne yazık ki bu durumlar çalışma anında ortaya çıkan hatalardır. Bunların yönetimi ancak yapısal bir düşünce ile gerçekleştirilebilir.) Standart C++'da *new*'in normal davranışı, yer ayırma işlemini gerçekleştiremediği durumda bir ayrıcalıklı durum oluşturmaktır. Eğer programınız bu durumu işlemez ise program sonlandırılır. İşin kötü tarafı bu hata durumunda *new*'e yüklenen görevin son birkaç yılda değiştirilmesidir. Bu yüzden derleyicinizin *new*'i standart C++ gibi işlememesi ihtiyat dahilindedir.

C++ ilk bulunduğuanda *new* hata durumunda geriye boş döndürüyordu (*null*). Daha sonra bu hataya cevap şekli ayrıcalıklı bir durum oluşturacak şekilde değiştirildi. Son olarak *new* hatasının normalde bir ayrıcalıklı durum olusurmasına, fakat bir seçenek ayarı ile bir boş işaretçi döndürebilmesine karar verildi. Böylelikle *new* fonksiyonu derleyici üreticileri tarafından farklı yorumlanmış oldu. Örneğin ben kitabı hazırlayıorken *new* hata ürettiğinde, Microsoft Visual C++ boş işaretçi döndürürken Borland C++ ayrıcalıklı bir durum meydana getiriyordu. Bütün derleyiciler *new*'i Standart C++'a yakın bir şekilde yorumlasa da, gerçek tepkiyi öğrenmenin tek yolu derleyicinizin dokümanlarına bakmaktır.

new yer ayırma işleminde iki farklı hata üretebileceğine sahipse de, aynı zamanda farklı derleyiciler, bu hatayı farklı yorumlayabilseler de, kitaptaki kod bu iki duruma da yer verecek şekilde yazılmıştır. Bu kitaptaki kodun tümü, *new*'in döndürdüğü işaretçinin boş olup olmadığını kontrol edecek şekilde yazılmıştır. Bu işlem, hataya ayrıcalıklı durum yaratarak cevap veren derleyicilerde zararsızken, hatayı boş döndürerek yorumlayan derleyiciler için durumu kontrol eder. Eğer *new* bir hata üretir ve ayrıcalıklı bir durum meydana getirirse programınız sonlanacak, daha sonra eğer ayrıcalıklı durum işlenmişse *new* tekrar denetlenecektir. Bu yapılanların sayesinde siz de yer ayırma hatasının daha iyi nasıl ele alınacağını göreceksiniz. Ayrıca *new* fonksiyonunun, bir hata meydana geldiğinde her zaman boş döndürmesinin dışında alternatif bir formunda meydana geldiğini öğreneceksiniz.

Son olarak: Az miktarda yer ayırması istendiğinden bu örneklerden hiçbir *new* komutunun çalışmamasına sebep olmayacağı.

new ve *delete*, *malloc()* ve *free()* ile benzer işleve sahip olsa da bazı avantajları mevcuttur. Öncelikle *new*, kendiliğinden belirtilen nesnenin tipine göre yeterli miktarda bellek ayırır. Örneğin, ne kadar byte gerektiğini anlamak için *sizeof* komutunu kullanmak mecburiyetinde değilsiniz. Bu durum hata riskini azaltır. İkincisi *new* kendiliğinden belirtilen tip için bir işaretçi döndürür. *malloc()*'ta yaptığınız gibi yer ayırma duru-

munda bir tip zorlaması yapmak mecburiyetinde degilsiniz. (bir sonraki nota bakın). Üçüncüsü, new ve delete komutlarının her ikisi de yeniden yüklenebilir. Bu sayede kendi özel yer ayırma sisteminizi oluşturma imkanı tanınır. Dördüncüsü, dinamik olarak oluşturulmuş bir nesneye hazırlık amacıyla başlangıç öncesi iş yaptırmak mümkündür. Son olarak <cstdlib> satırını da program içerisinde bulundurmak zorunda degilsiniz.

NOT C'de, malloc()'un döndürdüğünü bir işaretçiye atamak için tip zorlaması yapmak gereklidir. Çünkü malloc() tarafından döndürülen void*, atanın solundaki işaretçinin tipine göre kendiliğinden uyumlu bir işaretçiye çevrilir. Fakat bu durum, malloc() kullanımında tip zorlamasının da kullanılması geraken C++'da geçerli değildir. Bu farkın amacı C++'in daha sağlam bir tip kontrolü sağlama imkanı verir.

Artık new ve delete anlatıldığına göre malloc() ve free() komutlarının yerine onları kullanalım.

Örnekler

- İşte ilk örnek için kısa bir program. Bir tamsayı için bellekten yer ayırr.

```
// new ve delete'in basit bir kullanımı
#include <iostream>
using namespace std;

int main()
{
    int *p;

    p = new int; // tamsayı için bellekten bir yer ayır

    if(!p) {
        cout << "Bellekte yer ayırma hatası \n";
        return 1;
    }

    *p = 1000;
    cout << "p'deki tamsayı: " << *p << "\n";

    delete p; // belleği serbest bırak

    return 0;
}
```

new'in döndürdüğü değerin kullanılmadan önce kontrol edildiğine dikkat edin. Daha önce belirttiğim gibi bu kontrol, sadece derleyiciniz new'i hata durumunda boş döndürecek şekilde yorumlarsa, anlam kazanır.

- İşte size nesneler için dinamik olarak yer ayıran program.

```
// Dinamik nesneler için yer ayırma
#include <iostream>
using namespace std;

class samp {
```

```

int i, j;
public:
    void set_ij(int a, int b) { i=a; j=b; }
    int get_product() { return i*j; }
};

int main()
{
    samp *p;

    p = new samp; // nesne için yer al
    if(!p) {
        cout << "Beldekte yer ayırma hatası \n";
        return 1;
    }

    p->set_ij(4, 5);

    cout << "Çarpım: " << p->get_product() << "\n";
    return 0;
}

```

Alıştırmalar

1. Bir **float**, **long** ve **char** için bellekten yer ayıran bir program yazın. Bu dinamik değişkenlere bir değer verin ve bunları ekranda gösterin. Son olarak dinamik olarak ayırdığınız belleği, **delete** kullanarak serbest bırakın.
2. Kişinin ismini ve telefon numarasını saklayan bir sınıf oluşturun. **new** kullanarak bu sınıfın bir nesne için bellekten yer ayırin. Daha sonra bu nesneye isminizi ve telefon numaranızı girin.
3. **new**'in yer ayırma hatası durumunda göstereceği iki farklı tepki nedir?

4.5. new ve delete için Ek Bilgi

Bu kısımda **new** ve **delete** için iki farklı yenilikten bahsedeceğiz. Birincisi, dinamik olarak yer ayrılan nesnelere ilk değer verilmesi. İkincisi, dinamik olarak dizilerin ayrılabilmesi. **new**'i aşağıdaki formda kullanarak dinamik olarak yer ayrılan bir nesneye ilk değer atayabilirsiniz.

```
p-var = new type (ilk-değer);
```

new kullanarak bir boyutlu bir diziye dinamik olarak yer ayırmak istiyorsanız:

```
p-var = new type [adet];
```

Bu komut çalıştırıldığında *p-var*, belirtilen tipteki *adet* sayısındaki elemanların başlangıcını gösterecektir. Bazı teknik nedenlerden dolayı dinamik olarak ayrılan dizilere ilk değer atamak mümkün değildir.

Dinamik olarak yer ayrılmış diziyi silmek için **delete**'i şu şekilde kullanın:

```
delete ( 1 p-var;
```

Bu gösterim derleyicinin dizideki her elemannın yok edici fonksiyonunu çağırmasını sağlayacaktır. *p-var*'ın defalarca serbest bırakılmasına neden olmaz. *p-var* sadece bir kere serbest bırakılır.

NOT

Eski derleyicilerde, sildiğiniz dizinin boyutunu *delete* komutunun yanında köşeli parantez içerisinde belirtmek gereklidir. Bu işlem ilk C++ derleyicilerinde gerekliydi. Fakat yeni derleyicilerde bu artık kullanılmıyor.

Örnekler

1. Bu program bir tamsayı için yer ayırıyor ve kullanıma hazırlıyor.

```
// Dinamik bir değişkeni hazırlamaya ilişkin örnek
#include <iostream>
using namespace std;
int main()
{
    int *p;
    p = new int(9); // give initial value of 9
    if(!p) {
        cout << "Bellekte yer ayırtma hatası \n";
        return 1;
    }
    cout << "p'deki tamsayı: " << *p << "\n";
    delete p; // belleği bırakma işlemi
    return 0;
}
```

Tahmin edebileceğiniz gibi bu program 9 değerini gösteriyor. Bu da p tarafından bellekte gösterilen yerin ilk değeridir.

2. Aşağıdaki program dinamik olarak ayrılan nesnelere ilk değerleri geçirir.

```
// Dinamik nesnelere yer ayırtma
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    samp(int a, int b) { i=a; j=b; }
    int get_product() { return i*j; }
};

int main()
{
    samp *p;

    p = new samp(6, 5); // ilk değer atayarak dinamik nesnelere yer ayırtma
    if(!p) {
        cout << "Bellekte yer ayırtma hatası \n";
        return 1;
    }
}
```

BÖLÜM 4: Diziler, İşaretçiler ve Başvurular

```

cout << "Çarpım: " << p->get_product() << "\n";
delete p;
return 0;
}

```

samp nesnesi ayrıldığında yapılandırıcı fonksiyonu kendiliğinden çağrılr ve 5,6 değerleri geçirilir.

- Bu program bir tamsayı dizisine yer ayırr.

```

// new ve delete Üzerine basit bir örnek
#include <iostream>
using namespace std;

int main()
{
    int *p;

    p = new int [5]; // 5 tamsayı için yer ayırma işlemi

    // her zaman işlemin başarılı olup olamadığını denetleyin
    if(!p) {
        cout << "Bellekte yer ayırma hatası \n";
        return 1;
    }

    int i;

    for(i=0; i<5; i++) p[i] = i;

    for(i=0; i<5; i++) {
        cout << "p'deki tamsayı (" << i << "): ";
        cout << p[i] << "\n";
    }

    delete [] p; // belleği bırak

    return 0;
}

```

Bu program aşağıdaki gibi bir çıkış verir.

```

Here is integer at p[0]: 0
Here is integer at p[1]: 1
Here is integer at p[2]: 2
Here is integer at p[3]: 3
Here is integer at p[4]: 4

```

- Aşağıdaki program dinamik olarak nesne dizisi oluşturur.

```

// Dinamik nesnelere yer ayırma
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    void set_ij(int a, int b) { i=a; j=b; }

```

```

    int get_product() { return i*j; }

}

int main()
{
    samp *p;
    int i;

    p = new samp [10]; // nesne dizisine yer ayırma işlemi
    if(!p) {
        cout << "Bellekte yer ayırma hatası \n";
        return 1;
    }
    for(i=0; i<10; i++)
        p[i].set_ij(i, i);
    for(i=0; i<10; i++) {
        cout << "Çarpım [" << i << "]": " ";
        cout << p[i].get_product() << "\n";
    }
    delete [] p;

    return 0;
}

```

Program aşağıdaki çıktıyı gösterir.

```

Çarpım [0]: 0
Çarpım [1]: 1
Çarpım [2]: 4
Çarpım [3]: 9
Çarpım [4]: 16
Çarpım [5]: 25
Çarpım [6]: 36
Çarpım [7]: 49
Çarpım [8]: 64
Çarpım [9]: 81

```

5. Bir önceki programın yeni sürümü olan bu program samp'e bir destructor tanımlar. Artık p bırakıldığında (free) her elemanın yok-edicisi çağrılacaktır.

```

// Dinamik olarak nesnelere yer ayırma
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    void set_ij(int a, int b) { i=a; j=b; }
    ~samp() { cout << "Yok ediliyor...\n"; }
    int get_product() { return i*j; }
};

int main()
{
    samp *p;
    int i;

    p = new samp [10]; // nesne dizisi için yer ayırma işlemi
    if(!p) {

```

```

        cout << "Bellekte yer ayırma hatası \n";
        return 1;
    }

    for(i=0; i<10; i++)
        p[i].set_ij(i, i);

    for(i=0; i<10; i++) {
        cout << "Çarpım [" << i << "]: ";
        cout << p[i].get_product() << "\n";
    }

    delete [] p;
    return 0;
}

```

Bu program aşağıdaki çıktıyi gösterir.

```

Çarpım [0]: 0
Çarpım [1]: 1
Çarpım [2]: 4
Çarpım [3]: 9
Çarpım [4]: 16
Çarpım [5]: 25
Çarpım [6]: 36
Çarpım [7]: 49
Çarpım [8]: 64
Çarpım [9]: 81
Yok ediliyor...
Yok ediliyor...
Yok ediliyor...
Yok ediliyor...
Yok ediliyor...
Yok ediliyor...
Yok ediliyor...
Yok ediliyor...
Yok ediliyor...
Yok ediliyor...

```

Gördüğünüz gibi samp yok-edicisi 10 kere çağrılmıştır- dizideki her eleman için bir kere.

Alıştırmalar

1. **new**'i kullanarak aşağıdaki kod ile aynı işi görecek bir kod hazırlayın.

```

char *p;
p = (char *) malloc(100);
// ...
strcpy(p, "Bu bir deneme ");

```

İPUCU Katarlann bir karakter dizisi olduğunu hatırlayın.

2. **new** kullanarak bir **double** için yer ayırin ve ilk değer olarak -123.0987 sayısını verin.

4.6. Başvurular

C++ işaretçilerle ilişkili bir yenilik daha içerir. Bu yenilik *başvurudur*. Başvuru, bütün amaçlı bir değişkenin diğer ismi gibi davranmak olan bir işaretcidir. Başvuruları üç farklı şekilde kullanabilirsiniz. Birincisi başvurular bir fonksiyona geçirilebilirler, ikincisi, bir fonksiyon tarafından döndürülebilirler. Üçüncüsü de, bağımsız bir başvuru oluşturulabilir. Başvuru parametrelerinden başlayarak bütün bu uygulamalar burada incelenmiştir.

Hiç şüphesiz başvurunun en önemli kullanımı bir fonksiyona parametre olarak geçirilmesidir. Başvuru parametresini ve nasıl çalıştığını anlamamıza yardımcı olması açısından gelin önce bir işaretçiyi (başvuruyu değil), fonksiyona parametre olarak geçiren bir program yazalım.

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

void f(int *n) // işaretçi parametresi kullanma işlemi

int main()
{
    int i = 0;

    f(&i);
    cout << "i'nin yeni değeri: " << i << '\n';

    return 0;
}

void f(int *r)
{
    *r = 100; // n tarafından gösterilen argümana 100 yerlestirme
}
```

Burada *f()*, *n* tarafından gösterilen tamsayıya 100 değerini yükliyor. Bu programda *f()*, *main()* içerisindeki *i*'nin adresi ile birlikte çağrılmıyor. Bu yüzden *f()* tamamlandığında *i*, 100 değerine sahip oluyor.

Bu program parametre olarak bir işaretçinin, "başvuru parametresi geçirme mekanizması" ile manuel çağrı oluşturmak için nasıl kullanıldığını gösteriyor. Bir C programında bu işlem başvuru ile çağrıyı gerçekleştirmenin tek yoludur. Fakat C++'da başvuru parametresi kullanarak bu işlemi otomatikleştirebilirsiniz. Nasıl olduğunu görmek için bir önceki program üzerinde yeniden duralım. İşte size programın başvuru parametresi kullanan yeni sürümü:

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

void f(int &n); // başvuru parametresi bildirme

int main()
{
    int i = 0;
```

```

    f(i);
    cout << "i'nin yeni değeri: " << i << endl;
    return 0;
}

// f() artık başvuru parametresi kullanıyor
void f(int *n)
{
    // şimdiki komut için * operatörüne gerek olmadığına dikkat edin
    n = 100; // f() çağrırmak için kullanılan argümana 100 koy
}

```

Bu programı dikkatlice inceleyin. Önce bir başvuru değişkeni veya parametresi bildirmek için değişken ismini & işaretini ile belirtirsiniz. Bu **n**'nin **f()**'e parametre olarak bildirilmesi işlemidir. Artık **n** bir başvuruudur ve daha fazla gerekli değildir. Hatta onu ***** operatörü ile kullanmak bile geçerlidir. **f()** ile birlikte **n**, her kullanıldığından, kendiliğinden **f()**'i çağrırmak için kullanılan argümana işaretçi olması üzerine uyarılır.

n = 100;

İfadesi gerçekten **f()**'i çağrırmak için kullanılan değişkenin içerisinde 100 değerini yerleştirir. Bu programda o değişken **i**'dır. Daha da ileri gidersek **f()**'i çağrıdığımızda & işaretini kullanmamıza gerek yoktur. Çünkü **f()**'in bir başvuru parametresi olması gerekiği belirtilmiştir. Argümanın adresi kendiliğinden **f()**'e geçirilir.

Özetlersek, bir başvuru parametresi kullandığınızda derleyici, argüman olarak kullanılan değişkenin adresini kendiliğinden geçirir. & ile birlikte argüman adresini oluşturmanıza gerek yoktur (Aşında mümkün değildir). Fonksiyon içerisinde, derleyici, kendiliğinden başvuru parametresi tarafından gösterilen değişkeni kullanır. ***** işaretini kullanmaya gerek yoktur. (Tabi ki izin yine verilmez.) Böylece başvuru parametreleri, çağrıyı, "başvuru parametreleri geçirme mekanizması" ile tamamen güvenilir biçimde otomatikleştirir.

Başvurunun işaret ettiğini değiştiremeyeceğinizi anlamak çok önemlidir: Örneğin buradaki

n++;

İşlemi bir önceki programda **f()**'in içerisinde koyduğunuzda **n** yine **main()**'nın içerisindeki **i**'yi gösterecektir. Bu komut **n**'i bir artırmak yerine başvurudurulan değişkeni bir artırır. (Bu örnekte **i**)

Başvuru parametreleri diğer esdeger işaretçi alternatiflerine göre bazı avantajlar sunar. Olayı pratiklik açısından incelersek birincisi, argümanın adresini geçirmeyi hatırlamak zorunda değilsiniz. Bir başvuru parametresi kullanıldığında, adresi kendiliğinden geçirilir. İkincisi bir çok programcı başvuru parametrelerini eski yetenek özürlü işaretçi mekanizmasına göre daha güzel ve anlaşılır buluyor. Üçüncüsü, bir sonraki kısımda göreceğiz, bir nesne fonksiyona başvuru olarak gönderildiğinde kopyası oluşturulmaz. Bu işlem yokedici fonksiyonu çağrıldığında, bir argüman kopyasının programda gerekli olan bir yere zarar verme ihtimalini ortadan kaldırmanın bir yoldur.

Örnekler

- Aşağıda başvuru yoluyla geçirilen argümanların değerlerinin değiştirilmesi ile ilgili klasik bir örnek var. İşte size başvuruyla iki argümanın değerlerini yer değiştiren **swapargs()** adında bir fonksiyon.

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

void swapargs(int &x, int &y);

int main()
{
    int i, j;

    i = 10;
    j = 19;

    cout << "i: " << i << ", ";
    cout << "j: " << j << "\n";

    swapargs(i, j);

    cout << "After swapping: ";
    cout << "i: " << i << ", ";
    cout << "j: " << j << "\n";

    return 0;
}

void swapargs(int &x, int &y)
{
    int t;

    t = x;
    x = y;
    y = t;
}
```

Eğer **swapargs()** işaretçi kullanılarak yazılısaydı aşağıdaki gibi bir kod meydana gelirdi.

```
void swapargs(int *x, int *y)
{
    int t;

    t = *x;
    *x = *y;
    *y = t;
}
```

Gördüğünüz gibi, **swaparg()**'ın başvuru sürümünü kullanarak, * operatörünün gereksinimi ortadan kaldırılmıştır.

- round()** fonksiyonu kullanarak bir **double** değerini yuvarlatan programa göz atalım. Yuvarlanacak değer başvuru ile geçiriliyor.

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

void round(double &num);

int main()
{
    double i = 100.4;

    cout << i << " 'nin yuvarlatılmış " ;
    round(i);
    cout << i << "\n";

    i = 10.9;
    cout << i << " 'nin yuvarlatılmış " ;
    round(i);
    cout << i << "\n";

    return 0;
}

void round(double &num)
{
    double frac;
    double val;
    // num'u tam ve kesirli kısımlarına ayırmak
    // modf() fonksiyonunu kullanır.
    if(frac < 0.5) num = val;
    else num = val+1.0;
}
```

`round()`, bir sayıyı tam ve kesirli kısımlarına ayırmak için kullanılan ve standart kütüphane fonksiyonu olan `modf()` fonksiyonunu dolaylı olarak kullanır. Kesirli kısmı geri döndürülür. Tam kısmı `modf()`'in ikinci parametresi tarafından gösterilen değişkenle konur.

Aliştırmalar

1. Bildirilen tamsayı parametresinin işaretini tersine çeviren `neg()` adında bir fonksiyon yazın. Bu işlemi iki farklı şekilde gerçekleştirin. Birincisi işaretçi parametresi kullanın, ikincisi ise başvuru parametresiyle işi bitirsin. Bu fonksiyonları tanıtan küçük bir de program hazırlayın.
2. Aşağıdaki programda hata nedir?

```
// Bu programda bir hata var
#include <iostream>
using namespace std;

void triple(double &num);

int main()
{
    double d = 7.0;
```

```

    triple(&d);
    cout << d;
    return 0;
}

// num'un değerini üç katına çıkarır
void triple(double &num)
{
    num = 3 * num;
}

```

- Başvuru parametreleri ile ilgili avantajlar nedir?

4.7. Nesnelere Başvuru Geçirme

Bölüm 3'te öğrendiğiniz gibi, "değer parametresi geçirme mekanizması" ile mevcut çağrıının kullanılmasıyla, bir nesne fonksiyona geçirildiğinde, o nesnenin bir kopyası oluşturuluyordu. Parametrenin yapılandırıcı fonksiyonu çağrılmasa da fonksiyon bittiğinde, parametrenin yok-edici fonksiyonu çağrıliyordu. Şunu hatırlamanız gereklidir ki, bazı zamanlarda bu durum bazı ciddi problemlere yol açabilir. Mesela yok-edici fonksiyonun dinamik belgeyi geri vermesi gibi durumlar...

Bu problemin çözümü ise nesneyi başvuru ile geçirmektir. (Diğer çözüm Bölüm 5'te tartışıldığı gibi, kopya yapılandırıcıların kullanımını sağlar.) Bir nesneyi başvuru ile geçirdiğinizde kopyası yapılmaz. Bu sayede fonksiyon sona erdiğinde nesnenin yok-edici fonksiyonu da çağrılmaz. Yalnız şunu hatırlayın, bu durumda fonksiyon içerisinde nesneye yaptığı değişiklikler argüman olarak kullanılan nesneyi de etkileyecektir.

NOT

Başvurunun işarettiği olmadığını anlamak çok kritik bir adımdır. Bu sebeple, bir nesne boşvuru olarak gönderildiğinde, üye erişim operatörü(.) olarak kalır. (->) operatörü yanlışlıt...

Örnek

- Şimdiki örnek nesneyi başvuru olarak geçirmenin kullanışlığını gösterecek. İşte **myclass** adında bir nesnenin, **f()** adı verilen bir fonksiyona geçirilmesini gerçekleştiren bir program.

```

#include <iostream>
using namespace std;

class myclass {
    int who;
public:
    myclass(int n) {
        who = n;
        cout << "Oluşturuluyor " << who << "\n";
    }
    ~myclass() { cout << "Yok ediliyor " << who << "\n"; }
}

```

BÖLÜM 4: Diziler, İsgaretçiler ve Başvurular

```

int id() { return who; }

// o değer olarak geçiriliyor
void f(myclass o)
{
    cout << o.id() << "alındı" << "\n";
}

int main()
{
    myclass x(1);

    f(x);

    return 0;
}

```

Bu fonksiyon aşağıdaki çıktıyı verir.

```

Oluşturuluyor 1
1 alındı
Yok ediliyor 1
Yok ediliyor 1

```

Gördüğünüz gibi yok-edici fonksiyon iki kere çağrılmıştır. Birincisi, `f()` fonksiyonu bitmesiyle, nesne 1'in kopyası yok-edildiğinde. İkincisi ise program sona erdiğinde. Fakat program `f()`'in başvuru parametresi kullanmasa sağlanarak değiştirilse, kopya oluşturulmaz ve bu sayede `f()` bittiğinde yok-edici fonksiyon da çağrılmaz:

```

#include <iostream>
using namespace std;

class myclass {
    int who;
public:
    myclass(int n) {
        who = n;
        cout << "Oluşturuluyor " << who << "\n";
    }
    ~myclass() { cout << "Yok ediliyor " << who << "\n"; }
    int id() { return who; }
};

// Şimdi o başvuru olarak geçiriliyor
void f(myclass &o)
{
    // . operatörünün hala kullanıldığına dikkat edin
    cout << o.id() << "alındı" << "\n";
}

int main()
{
    myclass x(1);
    f(x);
    return 0;
}

```

Programın bu sürümü aşağıdaki çıktıyi verir.

```
Olusturuluyor 1
1 alindi
Yok ediliyor 1
```

HATIRLATMA Nesne üyelerine başvuru kullanarak erişirken, ok operatörünü değil nokta operatörünü kullanın.

Aliştırma

1. Aşağıdaki programda hata nedir? Başvuru parametresi kullanarak nasıl düzeltileceğini gösterin.

```
// Bu programda hata var
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
public:
    strtype(char *s);
    ~strtype() { delete [] p; }
    char *get() { return p; }
};

strtype::strtype(char *s)
{
    int l;

    l = strlen(s)+1;
    p = new char [l];
    if(!p) {
        cout << "Bellekte yer ayirma hatasi \n";
        exit(1);
    }
    strcpy(p, s);
}

void show(strtype x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}

int main()
{
    strtype a("Merhaba"), b("Nasil");

    show(a);
    show(b);
    return 0;
}
```

4.8. Başvuruları Döndürme

Bir fonksiyon başvuru döndürebilir. Bölüm 6'da göreceğiniz gibi, belli operatörlerle asın yükleme gerçekleştirirken, başvuru döndürmek çok faydalı olabilir. Hatta fonksiyona ata işleminde sol tarafta bulunma görevini bile yükleyebilir. Bu ikisinin etkisi çok kullanışlı ve güçlündür.

Örnekler

1. Başlamak için işte size başvuru döndüren fonksiyon içeren bir program veriyorum.

```
//Başvuru döndüren bir fonksiyona ilişkin basit bir örnek
#include <iostream>
using namespace std;

int &f(); // başvuru döndür
int x;

int main()
{
    f() = 100; // f() tarafından döndürülen başvuruya 100 atı
    cout << x << "\n";

    return 0;
}

//int başvurusunu döndür
int &f()
{
    return x; // x'e başvuru döndür.
}
```

Buradaki **f()** fonksiyonu bir tamsayıya başvuru döndürecek şekilde bildirilmiştir. Fonksiyonun içerisinde

```
return x;
```

komutu global değişken olan **x**'in değerini döndürmez. Onun yerine kendiliğinden **X**'in adresini döndürür. (başvuru yapısı altında). Bu sebeple **main()** içerisindeki:

```
f() = 100;
```

komut satırı 100 değerini **x**'e koyar çünkü **f()** ona bir başvuru döndürmüştür.

Özetlersek, **f()** fonksiyonu bir başvuru döndürür. Böylelikle **f()**, atama işleminin sol tarafında kullanılırsa, **f()** tarafından döndürülen, ve üzerine atanılan bir **this** başvurusudur. **f()**, **x**'e bir başvuru döndürdüğünden, 100 değerini alan **x**'tir. (Bu örnek için)

2. Başvuru döndürürken, başvurulan nesnenin yapı içerisinde bulunmasına dikkat edin. Bu olayı şu işlemle ele alalım.

```
// int başvurusu döndür
int sf()
{
    int x; // x artık lokal bir değişken
    return x; // x'e başvuru döndür
}
```

x artık **f()** için lokal bir değişkendir ve **f()** sona erdiğinde alan dışına çıkacaktır. **f()** tarafından gönderilen başvuru bu durumda işe yaramaz.

NOT Bazı C++ derleyicileri lokal değişkenlere başvuru döndürmenize izin vermez. Fakat bu tip bir problem başka yollarla kendini telafi edebilir. Örneğin, nesnelerin dinamik olarak bellekten ayrılmaları gibi durumlarda...

3. Sınırlı bir dizi tipi oluşturulduğunda başvuru döndürmenin çok iyi bir kullanım şekli göze çarpar. Bildiğiniz gibi C ve C++'da dizi sınırları kontrol edilmez. Bu sebeple dizilerin alt ve üst sınırlarını aşmak mümkündür. Fakat C++'da sınır kontrolü gerçekleştiren bir "dizi sınıfı" oluşturabilirsiniz. İki götüren iki fonksiyon bulunan bir sınıf.-Bir tanesi diziye bilgiyi yerleştiren fonksiyon, diğeri ise bilgiyi çeken fonksiyon. Bu fonksiyonlar çalışma anında dizi sınırlarının dışına çıktııp çıkmadığını kontrol edebilirler.

Aşağıdaki program karakterler için dizi sınırlarını kontrol eden bir yapıyı gösteriyor.

```
// Sınırlı dizi örneği
#include <iostream>
#include <cstdlib>
using namespace std;

class array {
    int size;
    char *p;
public:
    array(int num);
    ~array() { delete [] p; }
    char tput(int i);
    char get(int i);
};

array::array(int num)
{
    p = new char [num];
    if(!p) {
        cout << "Bellekte yer ayırma hatası \n";
        exit(1);
    }
    size = num;
}

// Diziye birşeyler ata
char &array::put(int i)
{
    if(i<0 || i>=size) {
        cout << "Sınır hatası!!!\n";
        exit(1);
    }
    return p[i];
}
```

```
    }
    return p[i]; // return reference to p[i]
}

// Diziden birşeyler alma
char array::get(int i)
{
    if(i<0 || i>=size) {
        cout << "Sınır hatası!!!\n";
        exit(1);
    }
    return p[i]; // karakter döndür
}

int main()
{
    array a[10];

    a.put(3) = 'X';
    a.put(2) = 'R';

    cout << a.get(3) << a.get(2);
    cout << "\n";

    // Şimdi çalışma anı sınır hatası üret
    a.put(11) = '!';
}

return 0;
}
```

Bu örnek başvuru döndüren fonksiyonların pratik bir kullanımını gösteriyor. Programı dikkatlice incelemelisiniz. **Put()** fonksiyonunun, *i* parametresi tarafından tanımlanan dizi elemanına başvuru döndürdüğüne dikkat edin. Bu başvuru, diziye bilgi aktarmak için bir atama işleminde sol tarafta kullanılabilir. Eğer belirtilen *i* sınır dışında değilse tabi... Eğer sayıç sınır içerisindeyken belirtilen sayıca göre depolanan değeri döndüren fonksiyon ise **get()** fonksiyonudur. Bu yol ile elde edilen dizilere bazı zamanlar *güvenli dizi(safe array)* denir. (Bölüm 6'da güvenli dizileri oluşturmaının daha iyi yollarını göreceksiniz.)

Bir önceki programda dikkat etmeniz gereken başka bir nokta ise dizinin **new** kullanarak bellekten ayrılmışıdır. Bu işlem değişik uzunluklarda dizilerin bildirilmesini sağlar.

Belirtildiği gibi bu programda gerçekleştirilen sınır kontrolü C++ ile gerçekleştirilmiş pratik bir uygulamadır. Eğer dizi sınırlarının kontrolünü çalışma anında kontrol etmek istiyorsanız, bu işlem bunu gerçekleştirmenin bir yoludur. Fakat sınır denetiminin diziye erişimi yavaşlattığını unutmayın. Bu sebeple sınır denetimini gerçekten sınır aşımının gerçekten olabileceği durumlarda gerçekleştirmek daha verimli olacaktır.

Aliştırmalar

1. Tamsayılar için 2'ye 3 güvenli dizi oluşturan bir program hazırlayın. Dizinin işlemini gösterin.
2. Aşağıdaki kod parçası geçerli midir? Değilse neden?

```
int *f();
{
    ...
    int *x;
    x = f();
```

4.9. Bağımsız Başvurular Ve Sınırlamalar

Pek kullanılmasa da C++'da, bağımsız başvuru denen bir başvuru tipi vardır. Bağımsız başvuru, başka bir değişken için farklı isim veren bir başvuru değişkenidir. Çünkü başvurulara yeni değer atanamaz. Bildirildiğinde, bağımsız başvurunun hazırlanması gereklidir.

NOT

Bağımsız başvurular bazı zamanlar kullanıldığından bu konuyu öğrenmeniz önemlidir. Fakat, birçok programcı bu tür bir yapıya ihtiyaç olmadığını düşünür. Bu sebeple programı karıştırabilirler. İşin daha da ötesinde bağımsız başvurular C++ içerisinde sıkça kullanılmaktadır gerek onları yok saymak için ortada kötü bir neden yoktur. Fakat çoğu kısımda kullanımları göz ardı edilecektir.

Tüm başvuru tiplerine uygulanması gereken bazı kısıtlamalar vardır. Bir başvuruya başvuramazsınız. Başvurunun adresini elde edemezsiniz. Başvurularдан oluşan bir dizi oluşturamazsınız. Ayrıca bir bit alanına başvuru gösteremezsiniz. Bir sınıfın üyesi, dönüş değeri veya fonksiyon parametresi olmadıkça başvuruları hazır hale getirmek zorundasınız.

HATIRLATMA: Başvurular işaretçilere benzer fakat işaretçi değildir.

Aliştırmalar

1. İşte size bağımsız başvuru içeren bir program

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    int &ref = x; // bağımsız başvuru oluştur
    x = 10; // bu iki atama
    ref = 10; // işlevsel olarak eşdeğerdir
    ref = 100;
    // bu 100 sayısını iki kere basar
    cout << x << " " << ref << "\n";
    return 0;
}
```

Bu programda, **ref** adındaki bağımsız başvuru **x** için farklı bir isim sunar. Olayı pratik bir şekilde incelersek **x** ve **ref** aslında eşdeğerdir.

2. Bağımsız başvuru bir sabiti de gösterebilir. Örneğin aşağıdaki gösterim geçerlidir.

```
const int &ref = 10;
```

Yine bu tip başvuruların bir avantajı vardır fakat bu durumu farklı zamanlarda programa göre değişik şekillerde görebilirsiniz.

Aliştırma

1. Kendi başına bağımsız başvuru için iyi bir kullanım düşünün.

Pekiştirme Testi

Bu noktada aşağıdaki alıştırmaları yapabiliyor ve soruları cevaplayabiliyor olmalısınız.

1. Size aşağıdaki sınıf veriliyor. 2 boyutlu 2'ye 5 bir dizi oluşturun. Dizideki her nesneye seçtiğiniz bir ilk değeri verin. Daha sonra dizinin içeriğini gösterin.

```
class a_type {
    double a, b;
public:
    a_type(double x, double y) {
        a = x;
        b = y;
    }
    void show() { cout << a << " " << b << "\n"; }
};
```

2. Bir önceki problemdeki çözümünü, diziye işaretçi ile erişilecek şekilde değiştirelim.
3. **this** işaretçisi nedir?
4. **new** ve **delete**'in genel formlarını gösterin. **malloc()** ve **free()**'ye göre avantajları nelerdir?
5. Başvuru nedir? Başvuru parametresi kullanmanın bir avantajı nedir?
6. **double** şeklinde bir başvuru parametresi alan **recip()** adında bir fonksiyon oluşturun. Fonksiyonun o parametreyi değiştirip, onun yerine değiştirilmişini atamasını sağlayın. Bu işlemin işleyişini gösteren bir program hazırlayın.

Bütünleştirme Testi

Bu kısım sizin geçmiş bölümlerdeki konular da dahil olmak üzere, bu bölümde geçen konuları ne kadar kavradığınızı kontrol edecektir.

1. Nesneye bir işaretçi verilmiş olsun. Bu nesnenin üyesine hangi operatör ile erişirsiniz?
2. Bölüm 2'de bir katar için dinamik olarak yer ayıran **strtype** sınıfı oluşturulmuştur. **new** ve **delete**'i kullanmak suretiyle **strtype** sınıfı üzerinde yeniden çalışın (Program sizin için yeniden veriliyor)

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

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype();
    void show();
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr);
    p = (char *) malloc(len+1);
    if(!p) {
        cout << "Bellekte yer ayırma hatası \n";
        exit(1);
    }
    strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "p serbest bırakılıyor\n";
    free(p);
}

void strtype::show()
{
    cout << p << " - uzunluk: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("Bu bir deneme."), s2("Ben C++'ı seviyorum.");
    s1.show();
    s2.show();
    return 0;
}
```

3. Kendi başına bir önceki bölümünden herhangi bir programı alıp başvuru kullanacak hale getirin.

BÖLÜM 5

Fonksiyonların Aşırı Yüklenmesi

Constructor Fonksiyonlarının Aşırı Yüklenmesi

Kopya Yapılandırıcı Yaratmak ve Kullanmak

Overload Anakronizm

Varsayılan Argümanları Kullanmak

Aşırı Yükleme ve Belirsizlik

Aşırı Yüklenmiş Fonksiyonların Adresleri

Bu bölümde aşırı yükleme fonksiyonları hakkında daha fazla bilgi edineceğiz. Her ne kadar bu konu kitapta daha önceleri tanıtıldıysa da, bilinmesi gereken birkaç şey daha var. Konular arasında, yapılandırıcı fonksiyonların nasıl aşırı yüklenileceği, bir kopya yapılandırıcının nasıl yaratılacağı, fonksiyonlara varsayılan argümanların nasıl verileceği, ve aşırı yükleme sırasında belirsizlikten nasıl kaçınılacağı da bulunmaktadır.

Gözden Geçirme Testi

Bu bölümme başlamadan önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

1. Referans nedir? İki önemli kullanımını veriniz.
2. Bir **float**'ın ve **int**'in, **new** kullanılarak nasıl bellekten ayrılacagini ve **delete**'i kullanarak nasıl bırakılacağını gösteriniz.
3. Dinamik bir değişkeni başlangıç durumuna getirmek(ilk değer) için kullanılan **new**'in genel formu nedir? Somut bir örnek veriniz.
4. Aşağıda verilmiş olan sınıf için, 10 elemanlı bir dizinin, x'den 10'a kadar değer alacak şekilde nasıl hazırlanacağını gösteriniz.

```
class samp {
    int x;
public:
    samp(int n) { x = n; }
    int getx() { return x; }
};
```

5. Referans parametrelerinin avantajını ve dezavantajını belirtiniz.
6. Dinamik olarak ayrılmış diziler başlama durumuna getirilebilir mi?
7. **num**'u, **order** tarafından belirtilen sayıya göre derecesini artıran, verilmiş prototipi kullanarak **mag()** isimli bir fonksiyon yaratınız.

```
void mag(long &num, long order);
```

Örneğin, **num** 4 ve **order** 2 ise **mag()** döndüğünde **num** 400 olacaktır. Bir programda fonksiyonun çalıştığını gösterin.

5.1. Constructor Fonksiyonlarının Aşırı Yüklenmesi

Bir sınıfın yapılandırıcı fonksiyonunu aşırı yüklemek mümkündür, Hatta sık karşılaşılan bir durumdur (ancak diğer taraftan yok ediciyi aşırı yüklemek mümkün değildir.) Bir yapılandırcıyı aşırı yüklemeyi istemek için üç ana neden vardır: esneklik kazanmak, dizileri desteklemek, ve kopya yapılandırıcıları yaratmak. Bunların ilk ikisinden bu bölümde bahsedeceğiz. Kopya yapılandırıcılar bir sonraki bölümde tartışılmıştır.

Örnekleri çalışırken, sınıfların tüm oluşturulmuş şekilleri için ayrı constructor fonksiyonları bulunması gerektiğini akılmızdan çıkarmamalıyız. Eğer karşılık gelen hiçbir yapılandırıcı yokken program, nesne yaratmaya çalışırsa derleme sırasında hata çıkar. İşte bu yüzden, aşırı yüklenmiş yapılandırıcı fonksiyonlar C++'ta bu kadar yaygındır.

Örnekler

- Aşırı yüklenmiş yapılandırıcıların en sık kullanımı belki de bir nesneyi hazırlayıp, hazırlamama seçeneğini sağlamak içindir. Örneğin aşağıdaki programda **o1**'e bir başlangıç değeri verilmiş, fakat **o2**'ye verilmemiştir. Eğer argüman listesi boş olan yapılandırıcıyı kaldırırsak program derlenemeyecektir. Çünkü **samp** tipinden hazırlanmamış nesneye uygun bir yapılandırıcı yoktur. Bunun tersi de doğrudur: eğer parametrelendirilmiş yapılandırıcıyı kaldırırsanız, hazırlanmış nesne ile bağlantı kurulamayacağından, program yine derlenemeyecektir. Programın doğru şekilde derlenebilmesi için her ikisi de gereklidir.

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

class myclass {
    int x;
public:
    // iki yolla yapılandırıcıyı aşırı yükleyin
    myclass() { x = 0; } // hazırlayıcı yok
    myclass(int n) { x = n; } // var
    int getx() { return x; }
};

int main()
{
    myclass o1(10); // ilk değerle bildir
    myclass o2; // hazırlayıcı olmadan bildir.
    cout << "o1: " << o1.getx() << '\n';
    cout << "o2: " << o2.getx() << '\n';
    return 0;
}
```

- Yapılardırıcı fonksiyonların aşırı yüklenmesinin gerekliliğine bir diğer sebep de, hem tek tek nesnelerin hem de nesne dizilerinin, bir program içerisinde ortayamasına olanak sağlamaktır. Belki tecrübelerinizden biliyorsunuz, tek bir değişkeni hazırlamak alışılmış olup, bir diziyi hazırlamak pek yaygın değildir. (Genellikle, dizilere program işletilirken bilinen değerler atanır.) Böylelikle, hazırlanmış nesne dizileriyle hazırlanmamış nesne dizilerine ortam sağlamak için, hazırlamayı gerçekleştiren ve de gerçekleştirmeyen birer yapılandırıcıyı dahil etmeliyiz. Mesela, Örnek 1'deki **myclass** sınıfını düşünelim, bu bildirimlerin her ikisi de geçerlidir.

```
myclass ob(10);
myclass ob[5];
```

Hem parametrelendirilmiş hem de parametresiz yapılandırıcıları kullanarak, programınız istege bağlı olarak hazırlanmış veya hazırlanmamış nesnelerin yaratılmasına imkan sağlar. Tahmin edeceğiniz gibi, bir kere parametrelendirilmiş veya parametresiz yapılandırıcılar tanımlanırsa, bunları hazırlanmış yada hazırlanmamış dizileri yaratmakta kullanabilirsiniz.

Örnek vermek gerekirse, aşağıdaki program biri hazırlanmış diğer准备好未被准备的 myclass tipinde iki diziyi belirtmektedir.

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

class myclass {
    int x;
public:
    // iki yolla yapılandırıcıyı aşırı yükle
    myclass() { x = 0; } // hazırlayıcı yok
    myclass(int n) { x = n; } // var
    int getx() { return x; }
};

int main()
{
    myclass o1[10]; // diziyi ilk değerler olmadan bildir
    // ilk değerlerle bildir
    myclass o2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int i;
    for(i=0; i<10; i++) {
        cout << "o1[" << i << "]: " << o1[i].getx();
        cout << '\n';
        cout << "o2[" << i << "]: " << o2[i].getx();
        cout << '\n';
    }

    return 0;
}
```

Bu örnekte **o1**'in bütün elemanları yapılandırıcı fonksiyonu tarafından 0'a ayarlanmıştır. **o2**'nin elemanları programda gösterildiği gibi hazırlanmıştır.

- Yapılandırıcı fonksiyonlarının aşırı yüklenmesinin bir başka sebebi programcıya bir nesneyi başlangıç durumuna getirmekte en uygun metodu seçebilmesine olanak sağlamak içindir. Nasıl olduğunu anlamak için öncelikli olarak takvim tarihini tutabilen bir sınıf yaratan sonraki örneği incelemeniz faydalı olacaktır. **date()** yapılandırıcısını iki şekilde aşırı yükler. Bir formda tarihi bir karakter dizisi şeklinde kabul eder. Diğer formda tarih üç tam sayı şeklinde gönderilir.

```
#include <iostream>
#include <cstdio> // sscanf() için dahil edildi
using namespace std;

class date {
```

BÖLÜM 5: Fonksiyonların Aşırı Yüklenmesi

```

int day, month, year;
public:
    date(char *str);
    date (int m, int d, int y) {
        day = d;
        month = m;
        year = y;
    }
    void show() {
        cout << month << '/' << day << '/';
        cout << year << '\n';
    }
};

date::date(char *str)
{
    sscanf(str, "%d/%d/%d", &month, &day, &year);
}

int main()
{
    // katar kullanarak date nesnesini yapılandır
    date sdate("12/31/99");

    // tamsayılar kullanarak date nesnesini yapılandır.
    date idate(12, 31, 99);

    sdate.show();
    idate.show();
    return 0;
}

```

Bu programda da gösterildiği gibi **date()** yapılandırıcısının aşırı yüklenmesinin avantajı, hangi seçenek durumumuza en uygun ise onu seçebilme ve kullanma rahatlığı sağlamasıdır. Örneğin eğer bir **date** nesnesi kullanıcı tarafından girilen değere göre yaratılıyorsa, programın katarlı sürümü, kullanılabilecek en kolay yoldur. Yine de, eğer nesne birtakım içsel hesaplamalar sonrasında yapılandırlıyorsa, üç-tamsayı parametreli sürümü daha etkili olabilir.

Her ne kadar bir yapılandırıcıyı istenildiği kadar aşırı yüklemek mümkün olsa da, bu işlemi gereğinden çok yapmak sınıf üzerinde yıkıcı bir etki yapar. Teknik olarak, en iyisi bir yapılandırıcıyı, sıklıkla meydana gelebilecek durumlar için aşırı yüklemektir. Örneğin, **date()**'in bir üçüncü kez aşırı yüklenmesi, tarihin milisaniyeler mertebesinde girilebilmesi gibi ufak bir etki yapar. Ancak, **time_t** tipinde (sistemin tarihini ve zamanını tutan bir tip) bir nesnenin kabulü için aşırı yüklemek çok değerli olabilir. (Pekiştirme Testi'ne bakınız, örnek olarak bu tip alıştırmalara göz atınız.)

4. Bir sınıfın yapılandırıcı fonksiyonunun aşırı yüklenmesini gerektirecek bir durum daha vardır ki bu durum o sınıfın bir dinamik dizi için bellekte yer ayrılmıştır. İlerdeki bölümlerde de göreceğimiz gibi dinamik bir dizi hazırlanamaz. Böylelikle, eğer sınıf hazırlayıcı alan bir yapılandırıcıyı içeriyorsa, hazırlayıcı almayan aşırı

yüklenmiş sürümü eklemelisiniz. Örneğin şimdi dinamik olarak bir nesne dizisi için yer ayıran bir program yazalım.

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

class myclass {
    int x;
public:
    // iki yolla yapılandırıcıyı aşırı yükle
    myclass() { x = 0; } // hazırlayıcı yok
    myclass(int n) { x = n; } // var
    int getx() { return x; }
    void setx(int n) { x = n; }
};

int main()
{
    myclass *p;
    myclass ob(10); // tek değişkeni hazırla

    p = new myclass[10]; // burada hazırlayıcıları kullanılamaz
    if(!p) {
        cout << "Allocation error\n";
        return 1;
    }

    int i;

    // bütün elemanları ob'a ayarla
    for(i=0; i<10; i++) p[i] = ob;
    for(i=0; i<10; i++) {
        cout << "p[" << i << "] : " << p[i].getx();
        cout << '\n';
    }

    return 0;
}
```

Hazırlayıcısı olmayan, `myclass()`'ın aşırı yüklenmiş sürümü olmadıkça, `new`, bir derleme aşamasında hata doğuracaktır ve program derlenmeyecektir.

Alıştırmalar

1. Kısmen tanımlanmış olarak verilen bu sınıfı

```
class strtype {
    char *p;
    int len;
public:
    char *getstring() { return p; }
    int getlength() { return len; }
};
```

İki yapılandırıcı fonksiyonu ekleyiniz. İlkini parametre almayacak şekilde olsun. Bu (`new`'i kullanarak) bellekten 255 byte ayırsın, bu belleği boş katarak ayarlayın

ve `Ien`'e 255 değerini verin. Diğer yapılandırıcıyı iki parametre alacak şekilde hazırlayın. İlkı başlangıç değeri atamak için kullanılacak ve diğerı ise ayrılacak byte miktarını belirleyecektir. Bu sürümün belirtilen miktarda yeri bellekten ayırmasını ve katarı oraya yerleştirmesini sağlayın. Tüm gerekli sınır kontrollerini yapın ve bir program ile yapılandırıcıların çalıştığını gösteriniz.

- İkinci bölümün konu 2.1, ahşurma 2'de `stopwatch`'ı canlandırdınız. Çözümünüzü `stopwatch` sınıfını, hem parametresiz bir yapılandırıcıyı(ki zaten öyle yapıyor) hem de sistem zamanını standart `clock()` fonksiyonu tarafından döndürülen formda parametre olarak alan aşırı yüklenmiş yapılandırıcıyı destekleyecek şekilde genişletiniz. Çalışmalarınızın işleyişini gösteriniz.
- Kendi başınıza, aşırı yüklenmiş yapılandırıcı fonksiyonunun, programlama amaçlarınıza faydalayıabileceğini yönlerini düşününüz.

5.2. Kopya Yapılandırıcı Yaratılması Ve Kullanılması

Aşırı yüklenmiş yapılandırıcının en önemli formlarından biri de kopya yapılandırıcılarıdır. İlerleyen konuların örneklerinde de gösterildiği gibi, problemler bir nesne fonksiyondan döndüğünde ya da fonksiyona gönderildiğinde ortaya çıkar. Bu bölümde göreceğimiz gibi, bu problemlerden kaçınmanın bir yolu kopya yapılandırıcılar kullanmaktır.

Başlangıç olarak, bir kopya yapılandırıcının çözmesi için bir problem oluşturalım. Bir nesne bir fonksiyona gönderildiğinde, nesnenin bit bit kopyası yapılır ve nesneyi alan fonksiyona parametre olarak verilir. Bu arada, bu tanımlayıcı kopya istenmeyen durumlarda kalabilir. Örnek olarak, eğer nesne ayrılmış bellek için işaretçi içeriyorsa, kopya da orijinal nesne ile aynı hafızayı gösterecektir. Bu sebeple, eğer kopya bu hafızanın içerisinde bir değişiklik yaparsa, orijinal nesne içinde değişiklik yapılmış olacaktır. Aynı zamanda, fonksiyon sonlandığında kopyada yok edilecektir. Bu da yok-edicinin çağrılmasına sebep olur. Olay, orijinal nesneyi de etkileyeyecek, istenmeyen yan etkiler de oluşturabilir.

Benzer bir durum, nesnenin bir fonksiyon tarafından döndürülmesi halinde meydana gelir. Genellikle derleyici, fonksiyon tarafından döndürülen değerin bir kopyasını tutan geçici bir nesne oluşturur.(Bu otomatik olarak ve sizin kontrolünüzün dışında meydana gelir Çağırılan rutine değer bir kere döndürüldüğünde geçici nesne kapsam dışına çıkar. Geçici nesnenin yok-edicisinin çağrılmasına sebep olur. Bu arada, eğer yok-edici çağrıci rutine gerekli olabilecek bir şeyi yok ederse (örneğin, eğer dinamik olarak ayrılmış belleği bırakırsa), ortaya sorun çıkacaktır.

Bu problemlerin özü olay nesnenin bit bit kopyasının yapılmasıından ileri gelir. Bu sorunları önlemek için, siz programcılar, bir nesnenin kopyası yapıldığında ne gibi sorunların meydana gelebileceğini önceden tanımlayabilmelisiniz ki istenmeyen yan etkilerden kaçınabilesiniz. Bunu gerçekleştirmenin yolu bir kopya yapılandırıcı yaratmaktadır. Bir kopya yapılandırıcıyı tanımlayarak, bir nesnenin kopyası yapıldığında tam olarak ne olacağını özel-leştirebilirisiniz.

C++'ın, bir nesnenin değerinin bir başkasına verildiğinde ortaya çıkan iki ayrı durumu tanımladığını anlamınız önemlidir. İlk durum atamadır. İkinci durum ise 3 şekilde meydana gelebilen hazırlamadır:

- Bildirim ifadesinde bir nesnenin diğerin hazırlamak için kullanıldığından
- Bir nesnenin fonksiyona parametre olarak gönderilmesinde
- Bir fonksiyon tarafından döndürülen değer olarak kullanılacak nesnenin geçici olarak oluşturulmasında.

Kopya yapılandırıcılar hazırlayıcılara uygulanır, atamalara ise uygulanamaz.

Normalde, bir hazırlık yapıldığında, derleyici otomatik olarak bit bit kopyayı gerçekleştiricektir. (Yani burada C++ nesnenin kopyasını kendiliğinden oluşturan bir kopya yapılandırıcıları tanımlar) Ancak kopya yapılandırıcı tanımlayarak bir nesnenin diğerini hazırlaması sağlanabilir. Bir kere tanımlandıktan sonra nesne bir diğerini hazırlayıp kopya çağrırlar.

HATIRLATMA Kopya yapılandırıcıları atama işlemlerini etkilemezler.

Kopya yapılandırıcının en genel formu aşağıda gösterilmiştir.

```
classname (const classname &obj) {
    //yapılandırıcının içi
}
```

Burada *obj*, bir nesnenin hazırlamak için kullanılan nesneye referanstır. Örneğin, **myclass** olarak isimlendirilmiş bir sınıfı ele alalım. *y*, **myclass** tipinin bir nesnesi olmak üzere, şu atama **myclass** kopya yapılandırıcısını uyaracaktır.

```
myclass x = y; // y açık olarak x'i hazırlar
func1(y); // y parametre olarak gönderilir
y = func2(); // y döndürülen nesneyi alır
```

İlk iki durumda, *y*'nin referansı, kopya yapılandırıcıya gönderilecektir. Üçüncü satırda **func2()** tarafından döndürülen nesnenin referansı kopya yapılandırıcıya gönderilir.

Örnekler

1. Şimdi açık bir kopya yapılandırıcı fonksiyona neden ihtiyaç duyulduğunu gösteren bir örnek verelim. Bu program, dizi sınırlarının aşılmasını engelleyen, çok kısıtlı "güvenli" tamsayı dizi tipi yaratmaktadır. Her dizinin depolanacağı bölge **new** kullanılarak ayrılmıştır ve belleğe işaretçi her dizi nesnesinde saklanır.

```
/* Bu program "güvenli" dizi sınıfı oluşturur. Dizi için alan ayrıldığından,
bir dizi nesnesi diğerini hazırlamak için kullanıldığından yer ayırmak için
kopya yapılandırıcı sağlanır
*/
#include <iostream>
#include <cstdlib>
```

```

using namespace std;

class array {
    int *p;
    int size;
public:
    array(int sz) { //yapılendirici
        p = new int[sz];
        if(!p) exit(1);
        size = sz;
        cout << "Using 'normal' constructor\n";
    }
    ~array() {delete [] p;}

    //kopya yapılandırıcı
    array(const array &a);

    void put(int i, int j) {
        if(i>=0 && i<size) p[i] = j;
    }

    int get(int i) {
        return p[i];
    }
};

/* Kopya yapılandırıcı

Aşağıdaki programda bellek özellikle kopya için ayrılmıştır ve adresi p'ye
atanmıştır. Bu sebeple p orijinal nesne gibi dinamik olarak ayrılmış aynı
belleği işaret etmez.

*/
array::array(const array &a) {
    int i;

    size = a.size;
    p = new int[a.size]; //kopya için bellekten yer ayır
    if(!p) exit(1);
    for(i=0; i<a.size; i++) p[i] = a.p[i]; // içeriği kopyala
    cout << "Using copy constructor\n";
}

int main()
{
    array num(10); // bu "normal" constructor'u çağırır
    int i;
    //diziye birkaç değer koyar
    for(i=0; i<10; i++) num.put(i, i);

    // num'ı görüntüle
    for(i=9; i>=0; i--) cout << num.get(i);
    cout << "\n";

    // başka bir dizi oluştur ve num ile hazırla
    array x = num; // burası kopya yapılandırıcısını uyarır

    // x'i görüntüle
    for(i=0; i<10; i++) cout << x.get(i);

    return 0;
}

```

X'e ilk değer atamak için **num** kullanıldığında, kopya yapılandırıcı çağrılmıştır. Yeni dizi için bellek ayrılmış **x.p** içinde saklanmıştır. **num**'un içeriği **x**'in dizisine kopyalanmıştır. Bu şekilde **x** ve **num** aynı değerlerde dizilere sahip olur, fakat her bir dizi ayrı ve belirgindir.(Böylece, **num.p** ve **x.p** aynı hafıza parçasını belirtmez.) Eğer kopya yapılandırıcı yaratılmamış ise, bit bit hazırlama **array x = num**, **x**'in ve **num**'un kendi dizileri için aynı belleği paylaşılması ile sonuçlanacaktır. (Bu, **num.p** ve **x.p**'nin muhtemelen aynı bölgeye işaret etmesi olayıdır.)

Kopya yapılandırıcı sadece hazırlama için çağrılr. Örnek olarak, aşağıdaki kodda önceki örnekte tanımlanmış olan kopya yapılandırıcı çağrılmaz.

```
array a(10);
array b(10);
b = a; // bütün yapılandırıcıyı kopyalamaz
```

Bu durumda, **b = a** ifadesi atama işlemini gerçekleştirmektedir.

2. Kopya yapılandırıcının, bazı nesne tiplerinin fonksiyona gönderilmesi ile ilişkili olan birtakım problemlerin engellenmesinin nasıl mümkün olduğunu görmek için, aşağıdaki (yanlış olan) programı inceleyiniz:

```
// Programda hata var.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
public:
    strtype(char *s);
    ~strtype() { delete [] p; }
    char *get() { return p; }
};

strtype::strtype(char *s)
{
    int l;

    l = strlen(s)+1;
    p = new char [l];
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }

    strcpy(p, s);
}

void show(strtype x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}
```

```
int main()
{
    strtype a("Hello"), b("There");

    show(a);
    show(b);

    return 0;
}
```

Bu programda, bir **strtype** nesnesi **show()**'a gönderildiğinde, bit bit kopyası (hiç bir kopya yapılandırıcı tanımlanmadığından) yapılmıştır ve **x** parametresinin içine konmuştur. Böylelikle, fonksiyon döndüğünde, **x** kapsam dışına çıkar ve yok edilir. Bu, elbette ki **x.p**'yi bırakan **x** yok-edicisinin çağrılmasına sebep olur. Ancak bırakılan bellek, fonksiyonu çağrıran nesne tarafından halen kullanılmaktadır. Bu da bir hatayı doğurur.

Ortaya çıkan problemin çözümü, **strtype** sınıfı için kopya yaratıldığında hafıza ayıran bir kopya yapılandırıcısı tanımlamaktır. Bu yaklaşım aşağıdaki, düzeltilmiş programda kullanılmıştır.

```
/* bu program strtype nesnelerinin fonksiyonlara gönderilmesini sağlamak
   için kopya yapılandırıcısı kullanır.
*/
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
public:
    strtype(char *s); // constructor
    strtype(const strtype &o); // kopya constructor
    ~strtype() { delete [] p; } // destructor
    char *get() { return p; }
};

// "Normal" constructor
strtype::strtype(char *s)
{
    int l;

    l = strlen(s)+1;
    p = new char [l];
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    strcpy(p, s);
}

//kopya constructor
strtype::strtype(const strtype &o)
{
    int l;
```

```

l = strlen(o.p)+1;

p = new char [l]; // yeni kopya için bellekten yer ayırm

if(!p) {
    cout << "Allocation error\n";
    exit(1);
}

strcpy(p, o.p); // kopyaya katarı kopyala
}

void show(strtype x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}

int main()
{
    strtype a("Hello"), b("There");

    show(a);
    show(b);

    return 0;
}

```

Şimdi `show()` bittiğinde ve `x` kapsam dışına çıktığında, `x.p` (boşaltılacak olan) tarafından işaret edilen bellek, fonksiyona gönderilen nesne tarafından halen kullanılmakta olan belle ile aynı değildir.

Alıştırmalar

1. Kopya yapılandırıcı aynı zamanda, fonksiyonun döndüreceği değer olarak kullanılan geçici nesne, fonksiyon tarafından oluşturulduğunda, uyarılmıştır. Bunu akınızda bulundurun. Sıradaki çıktı ile ilgilenelim:

```

Constructing normally
Constructing normally
Constructing copy

```

Bu çıktı aşağıdaki program tarafından üretilmiştir. Neden olduğunu açıklayın ve önceden tahmin ederek neler olduğunu tanımlayınız.

```

#include <iostream>
using namespace std;

class myclass {
public:
    myclass();
    myclass(const myclass &o);
    myclass f();
};

```

```

// Normal constructor
myclass::myclass()
{
    cout << "Constructing normally\n";
}

// Copy constructor
myclass::myclass(const myclass &o)
{
    cout << "Constructing copy\n";
}

// Bir nesne döndür.
myclass myclass::f()
{
    myclass temp;
    return temp;
}

int main()
{
    myclass obj;
    obj = obj.f();
    return 0;
}

```

2. Aşağıdaki programın hatasını bulup düzeltiniz.

```

// Programda hata var
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
    int *p;
public:
    myclass(int i);
    ~myclass() { delete p; }
    friend int getval(myclass o);
};

myclass::myclass(int i)
{
    p = new int;
    if(!p) {
        cout << "Allocation error\n";
        exit(1);
    }
    *p = i;
}

int getval(myclass o)
{
    return *o.p; // değeri al
}

int main()
{
    myclass a(1), b(2);
    cout << getval(a) << " " << getval(b);
}

```

```

cout << "\n";
cout << getval(a) << " " << getval(b);
return 0;
}

```

3. Kendi cümlelerinizle, bir kopya yapılandırıcının amacını açıklayın. Normal bir yapılandırıcıyla ne gibi farklılıklar gösterir.

5.3. Aşırı Yükleme Anakronizmi

C++ ilk zamanlarında, **overload** anahtar kelimesine, aşırı yüklenmiş fonksiyon yaratmak için ihtiyaç duyulmuştur. Her ne kadar **overload** şimdiki moda geçmiş ve artık C++ derleyicileri tarafından desteklenmemekte ise de, eski programlarda kullandığını görebilirsiniz. Bu yüzden nasıl uygulandıklarını anlamınız önemlidir.

Overload'un genel hali aşağıda gösterilmiştir:

```
overload func-name;
```

Burada **func-name** aşırı yüklenecek fonksiyonun ismidir. Bu ifade aşırı yüklenmiş fonksiyon bildirimlerini öngörmektedir. Örneğin, aşağıdaki ifade sizin derleyiciye **timer()** isimli bir fonksiyonu aşırı yükleyeceğinizi söyler.

```
overload timer;
```

HATIRLATMA **overload** eski bir yapıdır ve artık modern C++ derleyicileri tarafından desteklenmemektedir.

5.4. Varsayılan Argümanları Kullanma

C++'ın fonksiyon aşırı yükleme ile ilişkili bir özelliği vardır. Bu özellik, *varsayılan argüman* olarak adlandırılır. Fonksiyon çağrıduğunda, karşılık gelen argüman belirtilmemişse, parametre olarak varsayılan bir değer verebilmenizi sağlar. Birazdan göreccksiniz ki, varsayılan argümanlarının kullanılma, fonksiyon aşırı yüklemesinin kayda değer şekilde kısaltılmış birimidir.

Fonksiyon çağrılığında eğer karşılık gelen argüman yoksa varsayılan argümana parametre vermek için, o parametrenin yanına = yerleştirerek istediğiniz değeri yanına koyabilirsiniz. Örneğin, bu fonksiyon iki parametresine "0" varsayılan değerini vermektedir:

```
void f(int a=0, int b=0);
```

Bu notasyonun değişken hazırlamaya benzettiğine dikkat edin. Bu fonksiyonu artık 3 değişik şekilde çağrılmak mümkündür. Birincisi her iki argümanı belirtmiş biçimde çağrılmak. İkincisi sadece birinci argümanı belirtmiş olarak çağrılmak. Bu durumda b varsayılan olarak 0'a ayarlanır. **f()**'in argümansız olarak çağrılması, hem a'nın hem de b'nin 0 ayarlanması sebep olur. Bu durumda aşağıdaki örnekler geçerlidir:

```
f(); // a ve b varsayılan olarak 0'a ayarlı
f(10); // a 10, b normal 0
f(10, 99) // a 10, b 99
```

Bu örnekte şurası açık olarak görünmelidir ki, **a**'yı varsayılan alıp, **b**'yi belirtmenin yolu yoktur. Bir veya daha fazla varsayılan argümanlı fonksiyon yarattığınızda, bu argümanlar sadece bir kere belirtilmelidir: Ya fonksiyon prototipinde ya da (eğer ilk kullanıldığı yerden önce ise) fonksiyonun kendi tanımında... Varsayılanlar hem prototipte hem de tanımında belirtilemezler. Bu kural aynı varsayılanı belirtseniz bile geçerliliğini korur.

Tahmin edebileceğiniz gibi, tüm varsayılan parametreler, varsayılanları olmayan parametrelerin sağında olmalıdır. Ayrıca, eğer bir kez varsayılan parametreleri tanımlamaya başladınız mı, varsayılan olmayan herhangi bir parametreyi belirtmemesiniz. Varsayılan argümanlarla ilgili bir başka nokta da onların sabitler veya global değişkenler olması gereklidir. Yerel değişkenler veya başka parametre olamazlar.

Alıştırmalar

- Az önce anlatılan konu ile ilgili örnekleri gösteren bir program aşağıda verilmiştir.

```
// varsayılan argümanlar ile ilişkili ilk basit örnek
#include <iostream>
using namespace std;

void f(int a=0, int b=0)
{
    cout << "a: " << a << ", b: " << b;
    cout << '\n';
}
int main()
{
    f();
    f(10);
    f(10, 99);
    return 0;
}
```

Sizinde beklediğiz gibi, bu program bize şu çıktıyi verir.

```
a: 0, b: 0
a: 10, b: 0
a: 10, b: 99
```

Hatırlayacağınız gibi, bir kez ilk varsayılan argüman belirtildiğinde, takiben bütün parametreler aynı şekilde varsayılanlara sahip olmalıdır. Örneğin, **f ()**'in biraz farklı sürümü derleme hatası verir.

```
void f(int a=0, int b) //yanlış! b de varsayılan olmalı
{
    cout << "a: " << a << ", b: " << b;
    cout << '\n';
}
```

2. Varsayılan argümanların, fonksiyon aşırı yüklemesiyle nasıl ilişkili olduğunu anlamak için özellikle, `rect_area()` isimli fonksiyonu aşırı yükleyen bir sonraki problemi inceleyelim. Bu fonksiyon bir dikdörtgenin alanını döndürmekte.

```
// aşırı yüklenmiş fonksiyonlarla dikdörtgenin alanını bul
#include <iostream>
using namespace std;

//kare olmayan dörtgenin alanını döndür
double rect_area(double length, double width)
{
    return length * width;
}

//karenin alanını döndür.
double rect_area(double length)
{
    return length * length;
}

int main()
{
    cout << "10 x 5.8 rectangle has area: ";
    cout << rect_area(10.0, 5.8) << '\n';
    cout << "10 x 10 square has area: ";
    cout << rect_area(10.0) << '\n';
    return 0;
}
```

Bu programda, `rect_area()` iki şekilde aşırı yüklenir. Birincisinde, dikdörtgenin her iki boyutu fonksiyona gönderilir. Bu sürüm, dikdörtgen, kare olmadığından kullanılır. Bu arada dikdörtgen kare olduğunda sadece bir argümanın belirtilmesi yeterlidir, ve `rect_area()`'nın ikinci sürümü çağrılr.

Düşündüğümüzde, bu durumda iki farklı argümana gerçekten ihtiyacımız olmadığı açıkır. Onun yerine 2. parametre `rect_area()`'ya işaret olacak bir değere ayarlanabilir. Bu değer fonksiyon tarafından görüldüğünde, `length` parametresini ikinci kez kullanır. Bu yaklaşımı bir örnek verelim:

```
// Dikdörtgenin alanını varsayılan argümanlarla hesaplar
#include <iostream>
using namespace std;

// dikdörtgenin alanını döndürür
double rect_area(double length, double width = 0)
{
    if(!width) width = length;
    return length * width;
}

int main()
{
    cout << "10 x 5.8 rectangle has area: ";
    cout << rect_area(10.0, 5.8) << '\n';

    cout << "10 x 10 square has area: ";
```

```

cout << rect_area(10.0) << '\n';
return 0;
}

```

Burada `width`'in varsayılan değeri "0" dır. Hiçbir dikdörtgen "0" genişliğine sahip olamayacağından bu değer seçilmiştir. (Tam olarak, "0" genişlikli dikdörtgen bir çizgidir). Böylece, eğer bu varsayılan değer görültürse, `rect_area()` otomatik olarak `length`'teki değeri `width` değerinin yerine kullanır.

Bu örnekte de görüldüğü gibi, varsayılan parametreler sıkça, fonksiyon aşırı yüklemesine basit bir alternatif sağlar. (Tabii ki, halen daha fonksiyon aşırı yüklemesine ihtiyaç duyulan birçok durum söz konusudur.)

- Yapilandırıcı fonksiyonlara varsayılan argüman vermek sadece geçerli değil, oldukça da yaygındır. Bu bölümde daha önceden de gördüğünüz gibi, birçok seferinde bir yapılandırmacı, hazırlanmış, hem de hazırlanmamış nesnelerin yaratılmasını sağlamak için aşırı yüklenir. Birçok durumda, bir veya daha çok argüman vererek bir yapılandırmacıyı aşırı yüklemekten kaçınabilirsiniz. Örnek olarak aşağıdaki programı inceleyiniz:

```

#include <iostream>
using namespace std;

class myclass {
    int x;
public:
    /* myclass'ın yapılandıricısını aşırı yüklemek yerine
     * varsayılan argümanı kullan
     */
    myclass(int n = 0) { x = n; }
    int getx() { return x; }
};

int main()
{
    myclass o1(10); // ilk değer ile bildir
    myclass o2; // hazırlayıcı olmadan bildir

    cout << "o1: " << o1.getx() << '\n';
    cout << "o2: " << o2.getx() << '\n';

    return 0;
}

```

Örnekte de görüldüğü gibi, `n`'ye "0" varsayılan değerini vererek açık belirtili ilk değerlere sahip olan, varsayılan değerleri yeterli nesneler yaratmak mümkündür.

- Varsayılan argümanın iyi bir kullanımı da parametrenin aynı zamanda bir seçenek olarak işe yaramasıyla ortaya çıkar. Parametreye, fonksiyona daha önce seçileni kullanmasını söyleyen bir işaret olan varsayılan değeri yüklemek mümkündür. Örneğin, aşağıdaki programda, `print()` fonksiyonu ekranda bir katar gösterir.

Eğer **how** parametresi **ignore**'a ayarlandıysa, yazı olduğu gibi ekranada yazılır. Eğer **how upper**'a ayarlanmış ise yazı üst durumda görünür. Eğer **how lower**'a ayarlı ise, yazı aşağı durumda görülür. **how** belirtildiğinde, fonksiyona en son **how** değerini kullanmasını söyleyen, -1 değeri verilmiş sayılır.

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

const int ignore = 0;
const int upper = 1;
const int lower = 2;

void print(char *s, int how = -1);

int main()
{
    print("Hello There\n", ignore);
    print("Hello There\n", upper);
    print("Hello There\n"); // büyük harf
    print("Hello there\n", lower);
    print("That's all\n"); // küçük harf
    return 0;
}

/* Belirtilen şekilde basar. Belirtilmemişse son hali kullanır */
void print(char *s, int how)
{
    static int oldcase = ignore;

    // belirtilmediyse eskisini kullan
    if(how<0) how = oldcase;
    while(*s) {
        switch(how) {
            case upper: cout << (char) toupper(*s);
                break;
            case lower: cout << (char) tolower(*s);
                break;
            default: cout << *s;
        }
        s++;
    }
    oldcase = how;
}
```

Bu fonksiyon aşağıdaki çıktıyı görüntüler.

```
Hello There
HELLO THERE
HELLO THERE
hello there
that's all
```

5. Bu bölümün başlarında kopya yapılandırıcının genel formunu görmüştük. Bu genel formda sadece bir argüman vardı. Bu arada ek argümanlar varsayılan değerleri aldı-

lığı sürece, ek argümanlar alan kopya yapılandırıcılar yaratmak mümkündür. Örneğin aşağıdaki program da bir kopya yapılandırıcının geçerli bir formudur.

```
myclass(const myclass &obj, int x=0) {
    //yapılandırıcının içi
}
```

İlk argüman kopyalanan nesneye başvuru ve bütün diğer argümanlar varsayılan olduğunda fonksiyon kopya yapılandırıcısı olarak tanınır. Bu esneklik bize başka kullanımı da olan kopya yapılandırıcılar oluşturma imkanı verir.

6. Varsayılan argümanlar kuvvetli ve etkin oldukları halde yanlış kullanılabilirler. Doğru kullanıldığında sorun olmadığı gibi, varsayılan argümanlar bir fonksiyonun işini etkin ve kolay kullanılır şekilde yapmasını sağlar. Bu, bir parametreye varsayılan değer verilirken etkili olan tek durumdur. Örneğin, eğer argüman değerinin onundan dokuzu istenen değer ise bu fonksiyona bu etki için bir varsayılan argüman vermek iyi bir fikirdir. Bu arada, bir değerin, diğerinin yerine kullanılmasının uygun olmadığı zamanlar, ya da varsayılan değeri işaret olarak kullanmanın hiçbir faydasının olmadığı hallerde, bir varsayılan değer hakkında düşünmek gereksizdir. Aslında çağrılmayan bir varsayılan argüman sağlamak programınızın yapısını bozar öyle ki o fonksiyonu kullanmak zorunda olan herhangi birinin yanlış iş yapmasına sebebiyet verebilir.

İyi bir C++ programcısı olmanın bir yolu da fonksiyon aşırı yüklemesinde olduğu gibi, bir varsayılan argümanı da ne zaman kullanıp, ne zaman kullanmaması gerektiğini bilmektir.

Aliştırmalar

1. C++ kütüphanesinde bulunan **strol()** fonksiyonunun prototipi:

```
long strtol(const char *start, const **end, int base);
```

Fonksiyon **start** ile gösterilen sayısal katarı uzun tamsayiya çevirir. Sayısal katarın tabanı **base** ile belirtiliştir. Dönüşte, **end**, sayının hemen sonunu takip eden katardaki karakteri işaret eder. Sayısal katarın uzun tamayı eşdeğeri döndürülür. **base** 2 ile 38 aralığında olmalıdır. Genelde taban 10 kullanılır.

Şimdi **mystrol()** isimli bir fonksiyon yaratın. İşlevi **strtol()** ile aynı olsun ancak **base**'e varsayılan değer olarak 10 verilsin. (bu çevrimi tam olarak sağlamak için **strtol()**'u dilediğiniz kadar kullanabilirsiniz. **<ctsdlib>** başlığınına ihtiyaç duyar.) Sürümünüzün düzgün çalıştığını gösteriniz.

2. Aşağıdaki fonksiyon prototipinin yanlışı nedir?

```
char *f(char *p, int x = 0, char *q);
```

3. Birçok C++ derleyicisi, imleci hareket ettirmek vs. gibi bazı standart olmayan fonksiyonları destekler. Eğer derleyiciniz böyle fonksiyonları destekliyorsa, imlecin bulunduğu noktadan o satırın sonuna kadar silme işlemi yapan, `mycreol()` isimli bir fonksiyon yaratınız. Bu arada, bu fonksiyona silinecek karakter pozisyonlarının sayısını belirten bir parametre veriniz. Eğer parametre belirtilmemişse, otomatik olarak geçerli satırı silsin. Aksi takdirde, sadece parametre ile belirlenmiş karakter pozisyonlarının sayısını kadar silsin.
4. Bir varsayılan argüman kullanan, aşağıda verilmiş olan prototipin hataları nedir.

```
int f(int count, int max = count);
```

5.5. Aşırı Yükleme ve Belirsizlik

Fonksiyonları aşırı yüklerken, programınızın içine belirsizlik tanımlamanız da mümkündür. Belirsizliğe neden olarak, aşırı yükleme, tip değiştirme, referans parametreler ve varsayılan argümanlar sebep gösterilebilir. Ayrıca, bazı belirsizlik tipleri aşırı yüklenmiş fonksiyonların kendilerinden kaynaklanır. Diğer tipleri, aşırı yüklenmiş fonksiyonların çağrıldığı durumlarda meydana gelir. Programınızın hatasız derlenebilmesi için önce belirsizliğin ortadan kaldırılması gereklidir.

Örnekler

1. Belirsizliğin en genel tiplerinden biri, C++'ın otomatik tip değiştirme kurallarından kaynaklanır. Bildiğiniz gibi, gönderilen parametre ile uyumlu (fakat aynısı değil) tipten argüman ile bir fonksiyon çağrıldığında argümanın tipi otomatik olarak hedef tipe çevrilir. `putchar()` fonksiyonunun argümanı `int` olarak belirtildiği halde, bir karakter ile çağrılabilmesine olanak vermesi gibi gerçekleşen kısa bir çevirimi tipidir. Ancak, bazı hallerde bu otomatik tip çevirimi, bir fonksiyon aşırı yüklenliğinde, belirsiz bir duruma sebep olabilir. Nasıl olduğunu anlamak için şu programı inceleyin.

```
// Programda belirsizlik hatası var
#include <iostream>
using namespace std;

float f(float i)
{
    return i / 2.0;
}

double f(double i)
{
    return i / 3.0;
}

int main()
{
    float x = 10.09;
```

```

double y = 10.09;

cout << f(x); // belirsiz değil - f(float)'ı kullan
cout << f(y); // belirsiz değil - f(double)'ı kullan

cout << f(10); // belirsiz, 10'u double'a mı yoksa float'a mı dönüştür??
return 0;
}

```

main()'deki açıklamalarında gösterdiği gibi, derleyici **float** veya **double** değişkenlerinden biri ile çağrıldığında **f()**'in doğru sürümünü seçebilmektedir. Ancak, tam sayı ile çağrıldığında ne olur? Derleyici **f(float)**'umu yoksa **f(double)**'ını çağırır. (Her ikisi de geçerli çevrimlerdir!) Her iki durumda da, bir tam sayıyı bir **float** veya **double**'a çevirmek geçerlidir. Böylece, belirsiz durum ortaya çıkar.

Bu örnek aynı zamanda aşırı yüklenmiş bir fonksiyonun çağrılmaması yoluyla da belirsizliğin meydana gelebileceğini göstermektedir. **f()**'in aşırı yüklenmiş sürümünün her biri belirsiz olmayan argümanlar ile çağrıldığı sürece tabii ki belirsizlik yoktur.

2. Burada, içinde ve kendinde belirsizlik olmayan bir fonksiyon aşırı yüklenmiş örneği bulunmaktadır. Ancak, bu fonksiyon yanlış argüman tipi ile çağrıldığında, C++'ın otomatik çevirim kuralları bir belirsiz duruma sebep olur.

```

//Program belirsizdir.
#include <iostream>
using namespace std;

void f(unsigned char c)
{
    cout << c;
}
void f(char c)
{
    cout << c;
}

int main()
{
    f('c');
    f(86); // Hangi f() çağrılacak??
    return 0;
}

```

Burada **f()**, 86 nümerik sabiti ile çağrıldığında, derleyici **f(unsigned char)**'ı mı yoksa **f(char)**'ı mı çağıracağımı bilemez. Her iki çevirimde aynı seviyede geçerlidir, böylece belirsizlik yine ortaya çıkar.

3. Tek farkları, birinin referans parametresi kullanması, diğerinin ise varsayılan değer ile çağrılan parametre kullanması olan fonksiyonların aşırı yüklemeye kalkarsınız bir tip belirsizliğine yol açarsınız. C++'ın verilen kurallı notasyonunda derleyicinin hangi fonksiyonu çağıracağının bilmesinin bir yolu yoktur. Unutmayın, değer pa-

arametresi alan bir fonksiyonu çağrırmakla, referans parametresi alan bir fonksiyonu çağrımanın arasında notasyon açısından bir fark yoktur. Örneğin:

```
// Belirsiz bir program
#include <iostream>
using namespace std;

int f(int a, int b)
{
    return a+b;
}

//belirsizlik
int f(int a, int &b)
{
    return a-b;
}

int main()
{
    int x=1, y=2;

    cout << f(x, y); //f()'in hangi sürümü çağrılacak

    return 0;
}
```

Burada, **f(x, y)** belirsizlidir çünkü fonksiyonun her iki sürümü de çağrılmıyor olabilir. Aslında tabiatıyla belirsiz olan iki fonksiyonun aşırı yüklenmesinden ve bunlara referans kararlaştırılmayacağından dolayı derleyici ifadeden önce hatayı verecektir.

4. Bir başka belirsizlikte, içinde bir yada daha çok aşırı yüklenmiş fonksiyonun, bir varsayılan argüman kullanması halinde meydana gelecektir. Şu programı inceleyiniz.

```
// Varsayılan argümanlarla belirsizlik ve aşırı yükleme
#include <iostream>
using namespace std;

int f(int a)
{
    return a*a;
}

int f(int a, int b = 0)
{
    return a*b;
}

int main()
{
    cout << f(10, 2); // f(int, int)'i çağırır
    cout << f(10); // belirsiz- f(int)'i mi f(int, int)'i i çağırıracak???

    return 0;
}
```

Burada **f(10, 2)** ifadesi tamamen geçerli ve belirlidir. Ancak yinede derleyicinin **f(10)** ifadesinin **f()**'in ilk sürümü mü yoksa **b** varsayıları ile ikinci sürümü mü olduğunu bilmesinin yolu yoktur.

Aliştırma

1. Az önce verdığım belirsiz programları derlemeyi deneyin. Meydana getirdikleri hata mesajlarının tipleri hakkında zihinsel notlar alınız. Bu ileride programlarınızda meyda-na gelecek belirsizlik hatalarını algılamanızda size yardımcı olacaktır.

5.5. Aşırı Yüklenmiş Fonksiyonun Adresini Bulma

Bu bölümü bitirmek için, aşırı yüklenmiş bir fonksiyonun adresini nasıl bulacağınızı öğreneceksiniz. C'de olduğu gibi, bir fonksiyonun adresini (giriş noktasıdır) bir işaretçiye atayabilirsiniz ve fonksiyon bu işaretçi ile çalıştırılır. Bir fonksiyonun adresi, atama ifadesinin sağ tarafına parantez ve argüman kullanmaksızın ismini koyarak elde edilir. Örneğin, eğer **zap()** geçerli bildirilmiş bir fonksiyon ise **p'ye zap()**'in adresini atamak mümkündür:

```
P=zap;
```

C'de işaretçinin herhangi bir tipi bir fonksiyonu işaret etmekte kullanılabilir çünkü işaret edebileceği sadece bir fonksiyon vardır. Ancak, C++'da durum biraz daha karmaşıktır çünkü bir fonksiyon aşırı yüklenebilir. Öyle ki hangi fonksiyonun adresinin elde edildiğini belirleyen bir mekanizma olmalıdır.

Çözüm hem zarif hem de etkileyicidir. Aşırı yüklenmiş bir fonksiyonun adresi elde edildiğinde, hangi aşırı yüklenmiş fonksiyonun adresinin elde edileceğini belirleyen *İşaretçinin bildirim şeklidir*. Alında, işaretçinin bildirimi bu aşırı yüklenmiş fonksiyonlara karşı eşleştirilir. Bildirimi eşleşen fonksiyon adresi kullanılacak fonksiyondur.

Örnek

1. İşte size **space()** isimli bir fonksiyonun iki sürümünü içeren bir program veriyorum. İlk sürüm ekrana **count**'un sayısı kadar boşluk görüntüler. İkinci sürüm **count**'un sayısı kadar **ch()**'ye gönderilen karakteri ekrana görüntüler. **main()**'de iki fonksiyon işaretçisi bildirilmiştir. İlk sadece bir tamsayı parametreye sahip olan fonksiyonu işaret edecek şekilde belirtilmiştir. İkincisi ise iki parametre alan bir fonksiyonu işaret edecek şekilde bildirilmiştir.

```
/* Fonksiyon işaretçilerini aşırı yüklenmiş fonksiyonlara atamak */
#include <iostream>
using namespace std;

// count sayısını kadar boşluk ver
void space(int count)
{
    for (int i = 0; i < count; i++)
        cout << " ";
}

// ch karakterini count kez ekrana yaz
void space(char ch, int count)
{
    for (int i = 0; i < count; i++)
        cout << ch;
}
```

```
for( ; count; count--) cout << ' ';
}

// count sayısız kadarchs gönder
void space(int count, char ch)
{
    for( ; count; count--) cout << ch;
}

int main()
{
    /* void fonksiyonuna int parametresi ile işaretçi oluştur */
    void (*fp1)(int);

    /* void fonksiyonuna bir int parametresi bir de karakter parametresi ile
       işaretçi oluştur */
    void (*fp2)(int, char);

    fp1 = space; // space(int)'in adresini alır
    fp2 = space; // space(int, char)'in adresini alır

    fp1(22); // 22 boşluk çıkarır
    cout << "|\n";
    fp2(30, 'x'); // 30 x basar
    cout << "|\n";

    return 0;
}
```

fp1 ve **fp2**'nin bildirilişine göre derleyici hangi aşırı yüklenmiş fonksiyonun adresini elde edeceğini saptayabilir.

Konuyu özetlersek: Aşırı yüklenmiş bir fonksiyonun adresini bir fonksiyon işaretçisine atarsanız, hangi fonksiyonun adresinin atanacağını belirleyen işaretçinin bildirim şeklidir. Ayrıca, fonksiyon işaretçisinin bildirimi tamamen aşırı yüklenmiş fonksiyonlardan birine uymamıştır. Eğer uymazsa belirsizlik ortaya çıkar ve derleme hatası meydana gelir.

Aliştırma

1. Aşağıda iki aşırı yüklenmiş fonksiyon verilmiştir. Her birinin adresinin nasıl elde edileceğini gösteriniz.

```
int dif(int a, int b)
{
    return a-b;
}

float dif(float a, float b)
{
    return a-b;
}
```

Pekiştirme Testi

Bu noktada aşağıdaki soruları cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

1. Konu 5.1, Örnek 3'teki `date()` yapılandırıcısını, `time_t` tipinin bir parametresi olarak kabul edecek şekilde aşırı yükleyiniz. (`time_t`'nin C++ derleyicinizin kütüphanesinde bulunan standart zaman ve tarih fonksiyonları ile tanımlanan bir tip olduğunu hatırlayınız.)
2. Aşağıdaki kodda neyin yanlış olduğunu tespit ediniz.

```
class samp {
    int a;
public:
    samp(int i) { a = i; }
    // ...
};

// ...

int main()
{
    samp x, y(10);
    // ...
}
```

3. Bir sınıfın yapılandırıcısına niye ihtiyaç duyabileceğinize dair iki neden gösteriniz.
4. Bir kopya yapılandırıcının en genel şekli nedir?
5. Ne tip işlemler kopya yapılandırıcının istenmesine sebep olur.
6. Kısaca `overload` anahtar kelimesinin ne yaptığını ve artık niye kullanılmadığını açıklayınız.
7. Kısaca varsayılan argümanı tanımlayınız.
8. İki parametresi olan `reverse()` isimli bir fonksiyon yaratınız. `str` isimli ilk parametresi, fonksiyondan döndürülenec göre ters çevrilecek bir katarı işaret eder. `count` isimli ikinci parametresi, `str`'nin kaç tane karakterinin çevrileceğini belirler. `count`'a, var olduğunda `reverse()`'e girilen katarı çevirmesini söyleyen bir varsayılan değer veriniz.
9. Aşağıdaki prototipin neyi yanlıştur?

```
char *wordwrap(char *str, int size=0, char ch);
```

10. Fonksiyon aşırı yüklemesi sırasında karşılaşabileceğiniz birkaç belirsizliği açıklayınız.
11. Aşağıdaki kodda ne yanlıştır?

```
void compute(double *num, int divisor=1);
void compute(double *num);
// ...
compute(&x);
```

12. Aşırı yüklenmiş bir fonksiyonun adresini bir işaretleyiciye atadığınızda, fonksiyonun hangi versiyonunun kullanıldığını tespit eden nedir?

Bütünleştirme Testi

- İki adet tam sayı referans parametresi alan **order()** adında bir fonksiyon oluşturun. Eğer birinci argüman diğerinden büyükse yerlerini değiştirin. Değilse bir iş yapmayın. Yani **order()**'ı çağırarak, ilk argüman diğerinden az olacak şekilde sıralayın. Örneğin size,

```
int x=1, y=0;  
order(x, y);
```

verilmiştir. **order** çağrıldıktan sonra **x** 0 ve **y** 1 olacaktır.

- Neden bu iki aşırı yüklenmiş fonksiyonlar tabii olarak belirsizdir?

```
int f(int a);  
int f(int sa);
```

- Varsayılan argüman kullanmanın fonksiyon aşırı yüklemesi ile neden ilişkili olduğunu açıklayınız.
- Aşağıda verilmiş olan kısmi sınıfa, **main()**'deki her iki bildirimin geçerli olması için, gerekli olan yapılandırıcıyı ekleyiniz.(ipucu: **samp()**'ı iki kez aşırı yüklemelisiniz.)

```
class samp {  
    int a;  
public:  
    // yapılandırıcı fonksiyonları ekleyin  
    int get_a() { return a; }  
};  
  
int main()  
{  
    samp ob(88); // ob'nin a'sına 88 yerleştir  
    samp obarray[10]; // hazırlanmamış 10-elemanlı dizi  
    // ...  
}
```

- Kısaca kopya yapılandırıcılara neden ihtiyaç duyulduğunu açıklayınız.

BÖLÜM 6

Operatörleri Aşırı Yüklemek

Operatörleri Aşırı Yüklemenin Temelleri

İkili Operatörlerin Aşırı Yüklenmesi

**Karşılaştırma ve Mantık Operatörlerini Aşırı
Yüklemek**

Birli Operatörleri Aşırı Yüklemek

Arkadaş Operatör Fonksiyonlarının Kullanımı

Atama Operatörünü Yakından İnceleyelim

[] İndis Operatörünü Aşırı Yüklemek

Bu bölümde C++'ın bir diğer önemli özelliğini anlatacağız: Operatörlerin aşırı-yüklenmesi. Bu özellik sayesinde, tanımladığınız sınıflar için C++ operatörlerinin ne manaya geldiğini tanımlayabilirsiniz. Operatörleri aşırı-yükleyerek, programınıza yeni veri tipleri ekleyebilirsiniz.

Gözden Geçirme Testi

Daha ileri konulara geçmeden önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

1. Başlatılmamış (uninitialized) nesnelerin de oluşturulabileceği şekilde yapıcının aşağıdaki sınıf için nasıl aşırı-yükleneceğini gösterin. Başlatılmamış nesneler yaratırken **x** ve **y**'ye 0 değerini verin.

```
class myclass {
    int x, y;
public:
    myclass(int i, int j) { x=i; y=j; }
    // ...
};
```

2. İlk sorudaki sınıfı kullanarak, varsayılan argümanlarla **myclass()**'ı aşırı-yüklemekten nasıl kaçınabileceğinizi gösterin.
3. Aşağıdaki bildirimde yanlış olan nedir?

```
int f(int a=0, double balance);
```

4. Aşağıdaki aşırı-yüklenmiş iki fonksiyonda ne gibi bir yanlışlık vardır?

```
void f(int a);
void f(int &a);
```

5. Varsayılan argümanların kullanılması ne zaman uygundur, ne zaman değildir?
6. Size aşağıdaki sınıf tanımı veriliyor, bu nesnelerden oluşan bir dizi için dinamik olarak bellekte yer ayırmak mümkün müdür?

```
class test {
    char *p;
    int *q;
    int count;
public:
    test(char *x, int *y, int c) {
        p = x;
        q = y;
        count = c;
    }
    // ...
};
```

7. Kopya yapıcı nedir ve hangi şartlar altında çağrılır?

6.1. Operatör Aşırı-Yüklemenin Temelleri

Operatör aşırı-yükleme, tipki fonksiyon aşırı-yüklemeye benzer. Hatta operatör aşırı-yükleme aslında bir tür fonksiyon aşırı-yüklemedir, ama bazı ek kuralları bulunmaktadır. Örneğin, operatörler her zaman, sınıf gibi, kullanıcı-tanımlı bir tiple ilişkili olarak aşırı-yüklenebilir. Aralarındaki farklılıklardan ise sırası geldikçe bahsedeceğiz.

Operatörler aşırı-yüklendiklerinde orijinal anımlarını kaybetmeyip tanımladıkları sınıf'a göre ek anımlar kazanırlar. Operatörleri aşırı-yüklemek için önce bir *operatör fonksiyonunu* oluşturmanız gereklidir. Operatör fonksiyonları çoğunlukla tanımladıkları sınıf'a ya üye-dirler ya da arkadaşırlar. Fakat üye operatör fonksiyonları ile arkadaş operatör fonksiyonları arasında ince bir nüans vardır. Bu bölümün başında üye operatör fonksiyonlarının nasıl oluşturulduğunu öğreneceğiz. Sonra arkadaş operatör fonksiyonlarından bahsedeceğiz.

Üye operatör fonksiyonlarının genel kullanım şeklini aşağıda inceleyelim :

```
dönüş-tipi sınıf-adı::operator#(argüman-listesi)
{
    // gerçekleştirilecek işlem
}
```

Operatör fonksiyonlarının döndürme tipi, genelde fonksiyon hangi sınıf için tanımlandıysa o sınıfı ifade eder. Fakat operatör fonksiyonları herhangi bir tip döndürmekte serbesttir. Aşırı-yüklenen operatör '#'ın yerini alır. Örneğin eğer + operatörünü aşırı-yüklüyor ise fonksiyonun adı **operator+** olur. *argüman-listesi*'nin içeriği, operatör fonksiyonunun tanımlanma şecline ve aşırı-yüklenebileceği operatörün tipine göre değişir.

Operatörleri aşırı-yüklerken hatırlamamız gereken iki önemli kısıtlama bulunmaktadır: Bunlardan ilki, operatörün önceliğinin değiştirilememesidir. İkincisi ise, operatörlerin terim sayısının değiştirilememesidir. Örneğin / operatörünü tek bir terim alacak şekilde aşırı-yüklemeyeziz.

C++ operatörlerinin çoğu aşırı-yüklenebilir. Aşırı-yüklenemeyen operatörler, aşağıda gösterilen operatörlerdir:

```
. :: * ?
```

Ön-işlemci operatörlerini de aşırı-yüklemeyezsiniz. .* operatörü çok özel bir operatördür ve bu kitabın sahasının dışına çıkmaktadır.

C++, [] indis operatörleri, () fonksiyon çağrı operatörleri, new ve delete, . (nokta) ve -> (ok) operatörleri de dahil olmak üzere çok geniş bir yelpazede operatörleri tanımlar, bunu unutmayın. Fakat bu bölümde, en sık karşılaşığınız operatörlerin aşırı-yüklenebilmesine ağırlık vereceğiz.

= operatörü hariç, diğer operatör fonksiyonları herhangi bir türetilmiş sınıf tarafından miras alınabilir. Fakat türetilmiş sınıflar kendilerine bağlı operatörler içerisindeki seçikleri herhangi bir operatörü (temel sınıflar tarafından aşırı-yüklenenler de dahil olmak üzere) aşırı-yüklemeye serbesttir.

Şimdiye kadar aşırı-yüklendiği iki operatör kullandık: << ve >>. Bu operatörler I/O işlemlerini gerçekleyeceğimiz şekilde aşırı-yüklendiğimizdir. Daha önce de bahsettiğimiz gibi bu operatörlerin I/O işlemleri için aşırı-yüklendiği olması, onları kendi asıl görevleri olan sol kaydırma (left shift) ve sağ kaydırmayı (right shift) gerçekleştirmekten alikoymaz.

Sizin için operatör fonksiyonunun *bir işe* yaraması önem taşıyor, bu iş operatörün asıl görevi ile ister ilgili olsun ister olmasın. Halbuki operatörleri aşırı-yüklerken yapacağı işlerin bu operatörün asıl kullanımının fazla dışına çıkmaması her zaman en doğrusudur. Bu prensipten saparsanız ve operatörleri orijinal görevleriyle alakasız işleri yapacak şekilde aşırı-yüklerseniz programınızı mahvetme riskini de almış olursunuz. Örneğin / operatörünün bir dosyaya 300 kere "Ben C++'ı seviyorum" cümlesini yazacak şekilde aşırı-yüklediğinizi varsayılm. Bu, kafa karıştırıcı ve tamamen yanlış bir kullanım şeviden başka bir şey değildir!

Yukarıda söylediğimizde rağmen operatörleri asıl kullanımları ile ilgili olmayan şeviderde aşırı-yüklemeniz gereken zamanlar da olacaktır. Bunun en iyi iki örneği << ve >> operatörleridir. Bu operatörler I/O işlemleri için aşırı-yüklendiğimizdir. Fakat bu iki operatör için bile sol ve sağ ok operatörleri görsel bir "ipucu" sağlayarak manalarını ortaya koymaktadır. Sonuç olarak operatörü standart olmayan bir şekilde aşırı-yüklemeniz gerektiğinde en uygun operatörü kullanmak için elinizden geleni yapmalısınız.

Son bir şey daha belirtelim, operatör fonksiyonlarının varsayılan argümanları olamaz.

6.2. İkili Operatörlerin Aşırı Yüklenmesi

Bir üye operatör fonksiyonu bir ikili operatörü aşırı-yüklediğinde, bu fonksiyonun sadece tek parametresi olacaktır. Bu parametre operatörün sağ tarafındaki nesneyi alacaktır. Sol taraftaki nesne ise operatör fonksiyonunu çağırılan nesnedir ve üstü kapalı olarak **this** tarafından gönderilir.

Operatör fonksiyonlarını çeşitli şekillerde yazabiliriz. Burada verdığımız örnekler pek ayrıntılı değiller, ama sık karşılaşacağınız çeşitli teknikleri göstermektedir.

Örnekler

- Aşağıdaki programda **coord** sınıfıyla ilişkili olan + operatörünü aşırı-yüklenmektedir. Bu sınıfı X,Y koordinatlarını saklamak için kullanıyoruz.

```
// coord sınıfı içerisinde + operatörünü aşırı-yükleniyor
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
```

```

    coord operator+(coord ob2);
}

// + operatörü coord sınıfı için aşırı-yüklenmektedir.
coord coord::operator+(coord ob2)
{
    coord temp;
    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // İki nesne toplanmaktadır, bu deyim operator+()'yı çağırır.
    o3.get_xy(x, y);
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

Bu program size aşağıdaki sonucu verir :

(o1+o2) X: 15, Y: 13

Programı daha yakından inceleyelim. **operator+()** fonksiyonu, terimlerinin X koordinatlarının toplamını x'te, Y koordinatlarının toplamını da y'de bulunduran **coord** tipinde bir nesne döndürür. **operator+()** içerisinde bulunan **temp** adında geçici bir nesneyi, sonucu saklamak için kullanıyoruz. Terimlerden hiçbirinde bir değişiklik meydana gelmemekte, buna dikkat edin. **Temp** nesnesinin kullanılma nedeni basittir. Verdiğimiz örnekte + operatörü normal aritmetik kullanım amacına benzer bir şekilde aşırı-yükleniyor. Sonuç olarak terimlerden hiçbirinin değişmemesini sağlıyoruz. Örneğin, 10'a 4 eklediğimizde sonuç 14'dür, fakat ne 10 ne de 4'de bir değişiklik olmaz. Bu yüzden sonucun saklanması için geçici bir nesne kullanılması gereklidir.

operator+() fonksiyonumuz **coord** tipinde bir nesne döndürüyor, bunun nedeni **coord** tipindeki nesnelerin toplanmasından elde ettiğimiz sonucu daha geniş ifadelerde kullanabilmemizdir. Örneğin,

o3 = o1 + o2;

deyiminin geçerli olmasının tek nedeni **o1+o2**'nin sonucunun **o3**'e atanabilen bir **coord** nesnesi olmasıdır. Farklı bir tip döndürseydik bu deyim doğru sayılماçaktı. Üstelik **coord** tipinde bir nesne döndürerek toplam operatörünün ikiden fazla terimi toplamasını da sağlarız. Örneğin şu geçerli bir deyimdir:

o3 = o1 + o2 + o1 + o3;

Operatör fonksiyonunun tanımlandığı nesneden farklı bir nesne döndürmesini isteyeceğimiz bazı durumlar olacaktır, fakat oluşturduğumuz fonksiyonlar genellikle kendi sınıflarından bir nesne döndüreceklerdir. Bu kurallı bozan en büyük istisna, karşılaştırma ve mantık operatörlerinin aşırı-yüklenmesi durumudur. Bunu Konu 6.3'de inceleyeceğiz.

Bu örnek için son bir noktaya daha değinelim. coord tipinde nesne döndürülmesi sayesinde aşağıdaki deyim de tamamen geçerlidir:

```
(o1+o2).get_xy(x, y);
```

Burada **operator+()** tarafından döndürülen geçici nesne doğrudan kullanılmaktadır. Elbette bu deyim işletildikten sonra geçici nesne yok ediliyor.

2. Yukarıdaki programı aşağıda başka bir şekliyle ele alalım. Bu programda - ve = operatörleri coord sınıfı için aşırı-yüklenmektedir.

```
// coord sınıfıyla ilişkili olarak +, - ve = operatörleri aşırı-
// yüklenmektedir.
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator+(coord ob2);
    coord operator-(coord ob2);
    coord operator=(coord ob2);
};

// + operatörü coord sınıfı için aşırı-yüklenmektedir.
coord coord::operator+(coord ob2)
{
    coord temp;

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}

// - operatörü coord sınıfı için aşırı-yüklenmektedir.
coord coord::operator-(coord ob2)
{
    coord temp;

    temp.x = x - ob2.x;
    temp.y = y - ob2.y;

    return temp;
}

// = operatörü coord sınıfı için aşırı-yüklenmektedir.
coord coord::operator=(coord ob2)
```

```

    {
        x = ob2.x;
        y = ob2.y;

        return *this; // atanmış nesne döndürülüyor
    }

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // iki nesne toplanıyor - bu deyim, operator+()'yı çağırır.
    o3.get_xy(x, y);
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 - o2; // birinci nesneden ikinci çıkarılıyor
    o3.get_xy(x, y);
    cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1; // bir nesne diğerine atanıyor
    o3.get_xy(x, y);
    cout << "(o3=o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

operator-() fonksiyonu **operator+()**'ya benzer bir şekilde oluşturulur. Fakat yukarıdaki örnekte, terimlerin sırasının önemli olduğu bir operatörü aşırı-yüklerken dikkat etmemiz gereken ciddi bir noktaya deníniliyor. **operator+()** fonksiyonu oluşturulurken terimlerin sırası önemli değildi. Yani, A+B, B+A ile aynı şeydi. Fakat çıkarma işleminde sıra önemlidir. Bu yüzden çıkarma operatörünü doğru bir şekilde aşırı-yüklememiz için soldaki terimden sağdaki terimin çıkarılmasını sağlamalıyız. Çünkü soldaki terim **operator-()**'yi çağrıır ve çıkarma şu sırayla yapılmalıdır:

x = ob2.x;

HATIRLATMA İkili operatörlerin aşın-yüklenmesi sırasında, soldaki terim fonksiyona argüman olarak değil, üstü kapali olarak gönderilir. Sağdaki terim de argüman olarak gönderilir.

Şimdi atama operatörü fonksiyonuna bakalım. İlk olarak dikkatinizi çekmesi gereken şey soldaki terimin (yani değer atanmış nesnenin) işlem sonunda değişikliğe uğramasıdır. Bu, atama işleminin normal anlamına uygundur. Dikkatinizi çekecek ikinci şey ise fonksiyonun ***this**'i döndürmesidir. Yani **operator=()** fonksiyonu atama yapılan nesneyi döndürür. Bu sayede ardışık atama yapılmasını sağlarız. Sizin de bilmeniz gerektiği gibi C++'da aşağıdaki deyim geçerlidir (ve hatta çok sık kullanılır):

a = b = c = d = 0;

***this'in döndürülmesiyle aşırı-yüklenen atama operatörü, coord tipinde nesnelerin benzer şekilde kullanılmasına imkan verir. Örneğin aşağıdaki deyim tamamen geçerlidir:**

```
o3 = o2 = o1;
```

Şunu aklınızda tutun: Aşırı-yüklendi bir atama fonksiyonunun atama yapılan nesneyi döndürmesini gerektiren bir kural yoktur. Fakat eğer aşırı-yüklendi = operatörünün kendi sınıfı için hazır tiplere davranışının gibi davranışmasını istiyorsanız bu operatör ***this** döndürmelidir.

3. Bir operatörü sağ taraftaki terim, operatör fonksiyonunun sınıfına ait olmak yerine, tamsayı gibi mevcut tiplerde bir nesne olacak şekilde aşırı-yüklememiz mümkündür. Örneğin burada + operatörünü, bir tamsayı değerini coord tipinde bir nesneyle toplayacak şekilde aşırı-yüklüyoruz.

```
// + operatörü, ob + ob için olduğu gibi ob + int için aşırı-yüklüyor.
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator+(coord ob2); // ob + ob
    coord operator+(int i); // ob + int
};

// + operatörü coord sınıfı için aşırı-yüklüyor.
coord coord::operator+(coord ob2)
{
    coord temp;

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}

// + operatörü ob + int için aşırı-yüklüyor
coord coord::operator+(int i)
{
    coord temp;

    temp.x = x + i;
    temp.y = y + i;

    return temp;
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;
```

```

o3 = o1 + o2; // iki nesne toplanıyor - operator+(coord) çağrılıyor
o3.get_xy(x, y);
cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";
o3 = o1 + 100; // nesne + int toplanıyor - operator+(int) çağrılıyor
o3.get_xy(x, y);
cout << "(o1+100) X: " << x << ", Y: " << y << "\n";

return 0;
}

```

Nesnelerin diğer mevcut tiplerle işleme tabi tutulabilmesi için üye operatör fonksiyonlarını aşırı-yüklüyororsak mevcut tipin operatörünün sağ tarafında bulunmasına dikkat etmeliyiz. Bunun nedeni basittir: Operatör fonksiyonu soldaki nesneyi çağırır. Örneğin, derleyici aşağıdaki satırı gördüğünde ne olur?

```
o3 = 19 + o1; // int + ob
```

Bir tamsayı ile bir nesneyi toplamak için tanımlanmış mevcut bir işlem yoktur. Aşırı-yüklenmiş **operator+(int i)** fonksiyonu sadece nesne soldayken çalışır. Sonuç olarak, bu satırda bir derleme hatasıyla karşılaşız. Çok kısa bir süre sonra bu kısıtlamayı aşmanın yolunu öğreneceğiz.

4. Operatör fonksiyonlarında bir referans parametresi kullanabiliriz. Örneğin aşağıdaki yönteme + operatörünü **coord** sınıfı için aşırı-yüklemek mümkündür:

```
// + operatörü coord sınıfı için referans kullanılarak aşırı-yükleniyor.
coord coord::operator+(coord &ob2)
{
    coord temp;
    temp.x = x + ob2.x;
    temp.y = y + ob2.y;
    return temp;
}
```

Operatör fonksiyonlarında referans parametrelerini kullanmadığımız nedenlerinden biri verimliliği sağlamaktır. Nesnelerin parametre olarak fonksiyonlara gönderilmesi çok maliyetlidir ve önemli miktarda CPU çevrimi tüketir. Fakat nesnelerin adreslerinin gönderilmesi daima daha hızlı ve verimli bir yoldur. Eğer operatörü sık kullanacaksanız, referans parametresini kullanarak genellikle performansı önemli bir şekilde artıtabiliriz.

Referans parametrelerinin kullanılmasının bir diğer nedeni de, terimlere ait kopyalarının yok edilmesi sonucu ortaya çıkan sorunları engellemektir. Önceki bölgülerden bildiğimiz gibi, değeri gönderilen argümanın kopyası oluşturulur. Eğer bu nesneye ait bir yıkıcı fonksiyonu varsa, fonksiyon sona erdiğinde kopyanın yıkıcısı çağrıılır. Bazı durumlarda yıkıcının çağrılan nesne tarafından gerek duyulan bazı şeyleri yok etmesi mümkün değildir. Eğer durum böyleyse, değer parametresi yerine referans parametresini kullanarak problemi kolay (ve etkili) bir şekilde halledebiliyoruz. Tabii bu problemi her durumda engelleyecek bir kopya yapıcı da tanımlayabiliyoruz.

Aliştırmalar

1. Coord sınıfı için * ve / operatörlerini aşırı-yükleyn. Bu operatörlerin çalışıp çalışmadıklarını kontrol edin.
2. Aşağıdaki örnek neden aşırı-yüklenmiş operatörlerc uygundan olmayan bir uygulamadır?

```
coord coord::operator*(coord ob)
{
    double i;
    cout << "Bir sayı girin: ";
    cin >> i;
    cout << i << " nin karesi ";
    cout << sqr(i);
}
```

3. Kendi başınıza, operatör fonksiyonlarının döndürme tiplerini coord haricinde bir tipe dönüştürmeyi deneyin. Ne gibi hataların ortaya çıktığını dikkat edin.

6.3. Karşılaştırma ve Mantık Operatörlerini Aşırı-Yüklemek

Karşılaştırma ve mantık operatörlerinin aşırı-yüklenmesi mümkündür. Karşılaştırma ve mantık operatörlerini normal işlevlerine benzer bir şekilde aşırı-yüklediğinizde operatör fonksiyonlarının tanımlandıkları sınıfı ait bir nesne döndürmesini istemeyeceksiniz. Bunun yerine true veya false değerlerini gösterecek bir tam sayı döndüreceklerdir. Bunun tek yaptığı bu operatör fonksiyonlarının true/false değeri döndürmelerini sağlamak değildir, aynı zamanda operatörlerin başka veri tiplerini de içeren daha geniş karşılaştırma ve mantık ifadelerle tümləstirilmelerine de izin verir.

NOT

Eğer modern bir C++ derleyicisi kullanıyoysanız, aşırı-yüklenmiş karşılaştırma ve mantık operatör fonksiyonlarının bool tipinde değer döndürmesini sağlayabilirsiniz, fakat bunun pek bir avantajı yoktur. Bölüm 1'de anlattığımız gibi, bool tipi için sadece iki değer tanımıdır: true ve false (doğru ve yanlış). Bu değerler, otomatik olarak sıfır-değil ve sıfır değerlerine dönüştürülür. Sıfır-değil ve sıfır tam sayı değerleri de otomatik olarak true ve false'a dönüştürülür.

Örnek

1. Aşağıdaki programda == ve && operatörleri aşırı-yüklenmiştir:

```
// coord sınıfı için == ve && aşırı-yükleniyor.
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
```

```

coord() { x=0; y=0; }
coord(int i, int j) { x=i; y=j; }
void get_xy(int &i, int &j) { i=x; j=y; }
int operator==(coord ob2);
int operator&&(coord ob2);
}

// coord sınıfı için == operatörü aşırı-yükleniyor.
int coord::operator==(coord ob2)
{
    return x==ob2.x && y==ob2.y;
}

// coord sınıfı için && operatörü aşırı-yükleniyor.
int coord::operator&&(coord ob2)
{
    return (x && ob2.x) && (y && ob2.y);
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3(10, 10), o4(0, 0);
    if(o1==o2) cout << "o1, o2 ile aynı\n";
    else cout << "o1 ve o2 farklı\n";
    if(o1==o3) cout << "o1, o3 ile aynı\n";
    else cout << "o1 ve o3 farklı\n";
    if(o1&&o2) cout << "o1 && o2 true\n";
    else cout << "o1 && o2 false\n";
    if(o1&&o4) cout << "o1 && o4 true\n";
    else cout << "o1 && o4 false\n";
    return 0;
}

```

Aliştırma

- coord sınıfı için < ve > operatörlerini aşırı-yükle.

6.4. Birli Operatörleri Aşırı-Yüklemek

Birli operatörlerinin aşırı-yüklenmesi ikili operatörlerin aşırı-yüklenmesine benzemektedir, aradaki tek fark ortada tek bir terimin bulunmasıdır. Birli operatörleri üye fonksiyonları kullanarak aşırı-yüklediğimizde fonksiyon parametre almaz. Sadece tek bir terim bulunduğuandan operatör fonksiyonunu çağrıran da bu terimdir. Başka bir parametreye gerek yoktur.

Örnekler

- Aşağıdaki programda coord sınıfı için ++ operatörü aşırı-yüklenmektedir:

```

// coord sınıfı için ++ operatörü aşırı-yükleniyor.
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri

```

```

public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator++():
};

// coord sınıfı ile ilişkili olarak ++ aşırı-yükleniyor.
coord coord::operator++()
{
    x++;
    y++;
    return *this;
}

int main()
{
    coord ol(10, 10);
    int x, y;

    ++ol; // nesne bir arttırılıyor
    ol.get_xy(x, y);
    cout << "(++ol) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

Arttırma operatörü, terimini 1 artttırmak için tasarlandığından aşırı-yüklenmiş olan **++** işlem yaptığı nesneyi değiştirir. Fonksiyon, arttırdığı nesneyi aynı zamanda döndürür. Bu sayede arttırma operatörünü aşağıdaki gibi daha geniş deyimlerde kullanabiliriz:

```
o2 = ++ol;
```

İkili operatörlerde olduğu gibi, birli operatörlerin de normal işlevini yerine getirmesi için aşırı-yüklenmesi gerektiğini söyleyen bir kuralımız yok. Fakat, çoğu zaman bunu yapmak isteyeciksınız.

2. C++'ın eski sürümlerinde, artturma veya eksiltme operatörü aşırı-yüklendiğinde, aşırı-yüklenmiş **++** veya **--**'nin terimden önce mi sonra mı geldiğine karar vermek imkansızdır. Yani bir önceki programı düşünecek olursak bu iki deyim birbirinin aynı olurdu:

```
ol++;
++ol;
```

Fakat, C++'ın modern özellikleri sayesinde derleyici bu iki deyimi birbirinden ayırt edebilmektedir. Bunu gerçekleştirmek için iki farklı **operator++()** fonksiyonu oluşturmamız gerektir. Bu fonksiyonlardan ilkini yukarıdaki örnekle aynı şekilde tanımladık. İkincisini ise aşağıdaki gibi declare ettik:

```
coord coord::operator++(int notused);
```

Eğer `++` terimden önceyse `operator++()` fonksiyonu çağrılır. Fakat eğer `++` terimden sonraya `operator++(int notused)` fonksiyonu kullanılır. Bu durumda `notused`'a her zaman 0 değeri gönderilecektir. Sonuç olarak eğer `++`'nın terimin başında veya sonunda olması sınıfımızın nesneleri için önemliyse her iki operatör fonksiyonunu da kullanmamız gereklidir.

3. Bildiğiniz gibi C++'da eksi operatörünü hem ikili hem de birli operatör olarak sayıyoruz. Bu operatörü her iki şekilde birden nasıl yükleyeceğini merak ediyor olabilirsiniz. Bunun cevabı aslında çok basit: Tek yapmanız gereken, bu operatörü iki kere aşırı-yüklemek (bir kere ikili operatör olarak, bir kere de birli operatör olarak). Aşağıdaki programı inceleyerek bunu nasıl yapacağımızı açıklığa kavuşturalım:

```
// coord sınıfı için '-'nin aşırı-yüklenmesi.  
#include <iostream>  
using namespace std;  
  
class coord {  
    int x, y; // koordinat değerleri  
public:  
    coord() { x=0; y=0; }  
    coord(int i, int j) { x=i; y=j; }  
    void get_xy(int &i, int &j) { i=x; j=y; }  
    coord operator-(coord ob2); // ikili eksı operatörü  
    coord operator-(); // birli eksı operatörü  
};  
  
// coord sınıfı için '-'nin aşırı-yüklenmesi.  
coord coord::operator-(coord ob2)  
{  
    coord temp;  
  
    temp.x = x - ob2.x;  
    temp.y = y - ob2.y;  
  
    return temp;  
}  
  
// birli '-'nin coord sınıfı için aşırı-yüklenmesi.  
coord coord::operator-()  
{  
    x = -x;  
    y = -y;  
    return *this;  
}  
  
int main()  
{  
    coord o1(10, 10), o2(5, 7);  
    int x, y;  
  
    o1 = o1 - o2; // çıkarma  
    o1.get_xy(x, y);  
    cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";  
  
    o1 = -o1; // negatif yapılmıyor  
    o1.get_xy(x, y);
```

```

    cout << "(-ol) X: " << x << ", Y: " << y << "\n";
    return 0;
}

```

Gördüğünüz gibi eksi operatörünü ikili operatör olarak aşırı-yüklediğimizde bu operatör bir parametre alır. Birli operatör olarak aşırı-yüklediğimizde ise hiç parametre almaz. Eksiyi her iki işlem için ayrı ayrı aşırı-yükleyemememize imkan veren şey parametre sayısındaki bu faktür. Programdan da göreceğiniz gibi eksi operatörü ikili operatörü olarak kullanıldığında **operator-(coord ob2)** fonksiyonu çağrılır. Birli eksi olarak kullanıldığında ise **operator-()** fonksiyonu çağrılır.

Alıştırmalar

1. **-** operatörünü **coord** sınıfı için aşırı-yükleyin. Operatörün hem terimden önce hem de terimden sonra bulunması durumlarını da dikkate alın.
2. **+** operatörünü **coord** sınıfı için hem ikili operatör olarak (daha önce gösterildiği gibi) hem de birli operatör olarak aşırı-yükleyin. Birli operatör olarak kullanıldığında, **+** herhangi bir negatif koordinatı pozitif yapın.

6.5. Arkadaş Operatör Fonksiyonlarının Kullanımı

Bölümün başında da belirttiğimiz gibi, operatörleri arkadaş fonksiyonlarını kullanarak da aşırı-yüklememiz mümkün değildir. Arkadaş fonksiyonlarının **this** işaretçileri olmadığını daha önce konuşmuştu. İkili operatörlerle uğraşıyorsak bu, arkadaş operatör fonksiyonuna her iki terimin de açık bir şekilde gönderildiği anlamına gelir. Birli operatörler için ise tek bir terim gönderilir. Üye fonksiyon yerine arkadaş fonksiyonunun kullanmamızın tek bir önemli nedeni vardır, bu da Örneklerde anlattığımız nedendir.

HATIRLATMA Bir arkadaş fonksiyonunu, otama operatörünü aşırı-yüklemek için kullanamazsınız. Atama operatörü sadece üye operatör fonksiyonu tarafından aşırı-yüklenebilir.

Örnekler

1. Burada **operator+()** fonksiyonu ile **coord** sınıfı bir arkadaş fonksiyonu kullanılarak aşırı-yüklenmiştir:

```

// Arkadaş fonksiyonu kullanılarak + operatörü coord sınıfı için aşırı-
yukleniyor.
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }

```

```

coord(int i, int j) { x=i; y=j; }
void get_xy(int &i, int &j) { i=x; j=y; }
friend coord operator+(coord ob1, coord ob2);
};

// Bir arkadaş fonksiyonu kullanılarak + operatörü aşırı-yükleniyor.
coord operator+(coord ob1, coord ob2)
{
    coord temp;

    temp.x = ob1.x + ob2.x;
    temp.y = ob1.y + ob2.y;

    return temp;
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // iki nesne toplanıyor - bu deyim, operator+()'yı çağırıyor
    o3.get_xy(x, y);
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

Sol terimi ilk parametreye, sağ terimi ise ikinci parametreye gönderiyoruz, dikkat edin.

2. Arkadaş fonksiyonu ile operatörü aşırı-yüklediğimizde bu bize üye fonksiyonların sağlamadığı çok önemli bir özellik sağlar. Arkadaş operatör fonksiyonu kullanarak tamsayı gibi mevcut tipleri operatörün sol tarafına da koyabiliriz, bu şekilde nesnelerimizi bu tür işlemlere de sokabiliriz. İçeren işlemlerde kullanılmasını sağlayabiliyoruz. Daha önce sol terimin nesne, sağ terimin mevcut tiplerden olduğu ikili üye operatör fonksiyonunu aşırı-yüklemiştik. Fakat mevcut tiplere sahip değişkenlerin operatörün sol tarafında bulunması için üye fonksiyon kullanmamız mümkün değildir. Örneğin, aşırı-yüklenmiş bir üye operatör fonksiyonunu düşünelim, verilen ilk deyim kurallara uygundur, fakat ikincisi değildir:

```

ob1 = ob2 + 10; // kurala uygun
ob1 = 10 + ob2; // kurala uygun değil

```

Programımızda ilk deyime benzeyen deyimleri kullanabiliriz, fakat her seferinde nesnenin terimin sol tarafında ve diğer tipin terimin sağ tarafında olduğundan emin olmak zorunda olmamız bir engel teşkil edebilir. Çözüm, aşırı-yüklenmiş operatör fonksiyonlarını arkadaş haline getirmek ve her iki olası durumu da tanımlamaktır.

Bildiğiniz gibi, *her iki* terim de açık bir şekilde arkadaş operatör fonksiyonlarına gönderilir. Böylece, aşırı-yüklenmiş arkadaş fonksiyonunu sol terim nesne ve sağ terim de diğer tipte olacak şekilde tanımlamak mümkündür. O halde operatörü yi-

ne sol terim hazır tipte ve sağ terim nesne olacak şekilde de aşırı-yükleyebiliriz. Aşağıdaki programda bu metodu inceleyelim:

```
// Programa esneklik katmak için arkadaş operatör fonksiyonları kullanılıyor.
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int si, int sj) { i=x; j=y; }
    friend coord operator+(coord ob1, int i);
    friend coord operator+(int i, coord ob1);
};

// ob + int için aşırı-yükleniyor.
coord operator+(coord ob1, int i)
{
    coord temp;

    temp.x = ob1.x + i;
    temp.y = ob1.y + i;
    return temp;
}

// int + ob için aşırı-yükleniyor.
coord operator+(int i, coord ob1)
{
    coord temp;

    temp.x = ob1.x + i;
    temp.y = ob1.y + i;

    return temp;
}

int main()
{
    coord ob1(10, 10);
    int x, y;

    ob1 = ob1 + 10; // nesne + tamsayı
    ob1.get_xy(x, y);
    cout << "(ob1+10) X: " << x << ", Y: " << y << "\n";

    ob1 = 99 + ob1; // tamsayı + nesne
    ob1.get_xy(x, y);
    cout << "(99+ob1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

Arkadaş operatör fonksiyonlarını her iki durum için de aşırı-yüklememiz sayesinde bu deyimlerin her ikisi de geçerlidir :

```
ob1 = ob1 + 10;
ob1 = 99 + ob1;
```

Eğer arkadaş operatör fonksiyonlarını `++` veya `--` birli operatörlerinden birini aşırı-yüklemek için kullanmak istiyorsanız, terimi fonksiyona bir referans parametresi olarak göndermeniz gereklidir. Bunun nedeni arkadaş fonksiyonlarının `this` işaretçilerinin olmamasıdır. Artırma ve eksiltme operatörlerinin terimde değişiklik yapacağını unutmayın. Fakat eğer bu operatörleri değer parametresi olan bir arkadaş fonksiyonu kullanarak aşırı-yüklerseniz, arkadaş operatör fonksiyonunun içerisindeki parametreye yapılan değişiklikler çağrıma işlemini yapan nesneyi etkilemeyecektir. Nesneye herhangi bir işaretçi üstü kapalı olarak gönderilmediği sürece (yani `this` işaretçisi yokken) arkadaş kullanıldığında terimi etkileyebilecek bir artırma veya eksiltme yapılması mümkün değildir.

Fakat eğer terimi arkadaşa bir referans parametresi olarak gönderirseniz, arkadaş fonksiyonunun içerisinde meydana gelen değişiklikler çağrıyı yapan nesneyi etkiler. Örneğin aşağıdaki program `++` operatörünü bir arkadaş fonksiyonu kullanarak aşırı-yüklemektedir:

```
//++, bir arkadaş fonksiyonu kullanılarak aşırı-yükleniyor.
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    friend coord operator++(coord &ob);
};

//++, bir arkadaş fonksiyonu kullanılarak aşırı-yükleniyor.
coord operator++(coord &ob) // referans parametre kullanılıyor
{
    ob.x++;
    ob.y++;

    return ob; // çağrıyı yapan nesne döndürülüyor
}

int main()
{
    coord ol(10, 10);
    int x, y;

    ++ol; // ol referans tarafından gönderiliyor
    ol.get_xy(x, y);
    cout << "(++ol) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

Eğer modern bir derleyici kullanıyorsanız, tipki üye fonksiyonlar gibi arkadaş fonksiyonlarını kullanırken de artırma ve eksiltme operatörlerinin terimin başında veya sonunda bulunduğuunu anlayabilirsiniz. Nesne operatörün sağındaysa bu du-

rumu belirtmek için bir tamsayı parametre eklersiniz. Aşağıda örnekte **coord** sınıfı için artırma operatörünün nesnenin solda ve sağda bulunma durumları için prototipi verilmektedir:

```
coord operator++(coord &ob); // başında
coord operator++(coord &ob, int notused); // sonunda
```

Eğer **++** terimden önce geliyorsa, **operator++(coord &ob)** fonksiyonu çağrılır. Fakat, sonra geliyorsa, **operator++(coord &ob, int notused)** fonksiyonu kullanılır. Bu durumda **notused**'a 0 değeri gönderilecektir.

Alıştırmalar

- ve / operatörlerini arkadaş fonksiyonlarını kullanarak **coord** sınıfı için aşırı-yükle Yin.
- coord** sınıfını, her bir koordinat için tamsayı değerlerinin çarpılabileceği işlemlerde **coord** nesnelerini kullanabilecek şekilde aşırı-yükle Yin. İşlemleri her iki durumda da çalışacak şekilde düzenleyin: **ob * int** ya da **int * ob**.
- Alıştırma 2'nin çözümünün neden arkadaş operatör fonksiyonlarının kullanımını gerektirdiğini açıklayın.
- Bir arkadaş fonksiyonu kullanarak, **coord** sınıfı için –'nın nasıl aşırı-yükleneceğini gösterin. Terimin hem operatörün başında hem de sonunda olma durumlarını tanımlayın.

6.6. Atama Operatörünü Yakından İnceleyelim

Daha önce görmüş olduğumuz gibi, atama operatörlerini bir sınıf için aşırı-yüklememiz mümkün değildir. Normalde atama operatörü bir nesneye uygulandığında sağıdaki nesnenin bitler halinde bir kopyası soldaki nesneye konur. Eğer istediğiniz buysa kendi **operator=()** fonksiyonunuzu oluşturmanız gerekmek. Fakat bitler halindeki bu kopyayı istemeyeceğiniz durumlar da vardır. Bölüm 3'de nesnelerin bellekte yer kapladığı buna ilişkin birtakım örnekler görmüştük. Bu tür durumlarda özel bir atama işlemi yapacağız.

Örnekler

- Burada önceki bölümlerde çeşitli şekillerde görmüş olduğunuz **strtype** sınıfını biraz değiştirerek bir daha ele alalım. Fakat burada bir atama işlemi yaparak = operatörünü **p** işaretçisinin üzerine başka değer gelmesini engelleyerek aşırı-yükleyeceğiz.

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

```

class strtype {
    char *p;
    int len;
public:
    strtype(char *s);
    ~strtype() {
        cout << (unsigned) p << "Serbest bırakılıyor " << '\n';
        delete [] p;
    }
    char *get() { return p; }
    strtype &operator=(strtype &ob);
};

strtype::strtype(char *s)
{
    int l;

    l = strlen(s)+1;
    p = new char [l];
    if(!p) {
        cout << "Bellekte yer ayırma hatası\n";
        exit(1);
    }

    len = l;
    strcpy(p, s);
}

// Nesne atanıyor.
strtype &strtype::operator=(strtype &ob)
{
    // daha fazla belleğe gerek var mı kontrol ediliyor
    if(len < ob.len) { // bellekte daha fazla yer ayrılması gerekiyor
        delete [] p;
        p = new char [ob.len];
        if(!p) {
            cout << " Bellekte yer ayırma hataları\n";
            exit(1);
        }
    }
    len = ob.len;
    strcpy(p, ob.p);
    return *this;
}

int main()
{
    strtype a("Hello"), b("re");

    cout << a.get() << '\n';
    cout << b.get() << '\n';

    a = b; // şimdi p'nin üzerine yazılmıyor

    cout << a.get() << '\n';
    cout << b.get() << '\n';

    return 0;
}

```

Sizin de görebileceğiniz gibi, aşırı-yükleme atama operatörleri p'nin üzerine yazılmasını engeller. İlk önce soldaki nesne kendisine atanmakta olan katara yetecek kadar bellekte yer ayırmış mı kontrol edilir. Eğer ayırmamışsa bu bellek serbest bırakılır ve başka bir bölge ayrılır. Sonra katar bu belleğe ve katarın uzunluğu da len'e kopyalanır.

operator=() fonksiyonunun iki önemli özelliği daha vardır. Bunlardan ilki, bu fonksiyonun bir referans parametresi almasıdır. Böylece atamanın sağ tarafındaki nesnenin kopyasının yapılması engellenir. Daha önce de anlattığımız gibi fonksiyona gönderilen nesnenin kopyası yapılır ve fonksiyon sona erdiğinde bu kopya yok edilir. Bu durumda kopya yok edilseydi yıkıcı fonksiyonu çağırılacaktı, bu fonksiyon da p'yi serbest bırakacaktır. Fakat bu p, argüman olarak kullanılan nesne tarafından hala gerekli olan aynı p'dir. Bir referans parametresi kullanılmasıyla bu problemi önleriz.

operator=() fonksiyonunun ikinci önemli özelliği nesne yerine referans döndürmesidir. Bunun nedenini merak ediyorsunuzdur, çok basit. Neden referans parametresi kullanırsak yine aynı sebeple referans döndürüyoruz. Fonksiyon nesne döndürüyorsa geçici bir nesne oluşturulur ve bu nesne döndürme işleminin sona ermesiyle yok edilir. Fakat bunun bir sakincası vardır, geçici nesnenin yıkıcıyı çağrılır ve bu fonksiyon p'yi serbest bırakır. Fakat p (ve onun işaret ettiği bellek) değer atanmakta olan nesne için hala gereklidir. Sonuç olarak referans döndürülürse geçici nesnenin oluşturulmasını ve p'nin serbest bırakılmasını engelleriz.

NOT

Bölüm 5'te öğrendiğimiz, kopya için yıkıcı fonksiyonu oluşturarak her iki problemi de engelleyebiliyoruz. Fakat bu yöntem referans parametre ve referans döndürme tipi kullanmamız kadar etkili bir çözüm olmayıpabilir. Çünkü referans kullanırsak her durum için nesnenin kopyasının yapılması sonucunda ortaya çıkan problemi engelleriz. Burada da gördüğümüz gibi C++'da aynı sonuç ulaşmak için çeşitli metodlarımız mevcut. Hangisini kullanacağımız ise C++'daki ustalığımıza kalmıyor.

Alıştırmalar

- Size aşağıdaki sınıf bildirimini veriliyor, dinamik bir dizi tipi oluşturacak şekilde bu bildirimizi siz tamamlayın. Yani p'deki bu belleğe bir işaretçi koyarak bellekte dizi için yer ayırin. Dizini büyülüüğünü byte'lar halinde size'in içine koyun. **put()** belirtilen elemana bir referans döndürsün, **get()** ise belirtilen elemanın değerini döndürsün. Dizinin sınırlarının aşılmamasına dikkat edin. Aynı zamanda atama operatörünü öyle bir aşırı-yükleyin ki, her bir diziye ayrılan bellek dizilerin atanması sırasında kazayla yok edilmesin. Bir sonraki konuda bu alıştırmada kullanabileceğiniz bir başka yöntem öğreneceğiz.

```
class dynarray {
    int *p;
    int size;
```

```

public:
    dynarray(int s); // s'nin içindeki dizinin boyutunu gönderir
    int &put(int i); // i elemanına referans döndürür
    int get(int i); // i elemanın değerini döndürür
    // operator[]( ) fonksiyonunu oluşturun
};

```

6.7. [] İndis Operatörünü Aşırı-Yüklemek

Aşırı-yükleyeceğimiz son operatör, [] indis operatöründür. C++'da [] operatörü aşırı-yükleneceği zaman ikili operatör olarak kabul edilir. [] sadece tek bir üye fonksiyon tarafından aşırı-yüklenebilir. Sonuç olarak **operator[]()** üye fonksiyonunun genel şekli aşağıda gösterdiğimiz gibidir:

```

tip sınıf-adı::operator[ ](int indis)
{
    // ...
}

```

Teknik açıdan parametre **int** tipinde olmak zorunda değildir, fakat **operator[]()** fonksiyonları genelde dizilerin eleman sayılarının belirlenmesi için kullanılır ve bunun için de bir tamsayı değeri kullanılır.

[] operatörünün nasıl çalıştığını anlamak için **O** isminde bir nesnenin aşağıdaki gibi indekslendiğini varsayıalım:

O[9]

Bu indis, **operator[]()** fonksiyonuna aşağıda yapılan çağrıya dönüştürülecektir:

O.operator[](9)

Yani indis operatörünün içerisindeki ifadenin değeri **operator[]()** fonksiyonunun açık parametresine gönderilecektir. **this** işaretçisi çağrıyı yapan **O** nesnesini işaret edecektir.

Örnekler

1. Aşağıdaki programda, **arraytype** beş tamsayıdan oluşan bir diziyi bildirmektedir. Bu tipe ait yapıçı fonksiyonumuz ise dizinin her bir üyesi için gerekli hazırlığı yapmaktadır. Aşırı-yüklenmiş **operator[]()** fonksiyonu, parametresi ile belirlenen elemanın değerini döndürür.

```

#include <iostream>
using namespace std;

const int SIZE = 5;

class arraytype {
    int a(SIZE);
public:
    arraytype() {

```

```

        int i;
        for(i=0; i<SIZE; i++) a[i] = i;
    }
    int operator[](int i) { return a[i]; }
};

int main()
{
    arraytype ob;
    int i;

    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";

    return 0;
}

```

Bu program bize aşağıdaki sonucu verir :

0 1 2 3 4

Bu ve aşağıdaki diğer programlardaki yapıcı fonksiyonları tarafından a dizisi için yapılan hazırlığı sadece konuyu daha iyi anlamak için yapıyoruz, bu gerekli bir hazırlık değildir.

2. **operator[]()** fonksiyonunu, [] operatörü atama deyiminin hem sağında hem de solunda kullanılabilecek şekilde tasarlamamız mümkündür. Bunu yapmak için aşağıdaki programda indekslenmekte olan elemana bir referans döndürüyoruz:

```

#include <iostream>
using namespace std;

const int SIZE = 5;

class arraytype {
    int a[SIZE];
public:
    arraytype() {
        int i;
        for(i=0; i<SIZE; i++) a[i] = i;
    }
    int &operator[](int i) { return a[i]; }
};

int main()
{
    arraytype ob;
    int i;

    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";

    cout << "\n";

    // dizideki her elemana 10 ekleniyor
    for(i=0; i<SIZE; i++)
        ob[i] = ob[i]+10; // []'in solunda

```

BÖLÜM 6: Operatörleri Aşırı Yüklemek

```

    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";
    return 0;
}

```

Bu program aşağıdaki sonucu verir:

```

0 1 2 3 4
10 11 12 13 14

```

Şimdi **operator[]()** fonksiyonumuz indisi *i* olan dizi elemanına referans döndürüyor, bu sayede bu fonksiyon atamanın sol tarafında kullanılabilir ve dizinin elemanlarını değiştirebilir. Tabii, aynı zamanda sağ tarafında da kullanılabilir. Siz de fark edeceksiniz bu şekilde **arraytype** tipinde nesneler normal diziler gibi hareket edebilir.

3. **[]** operatörünü aşırı-yüklememizin bir avantajı ise güvenli dizi indeksleme işlemini iyileştirmesidir. Dizinin elemanlarına erişirken önceden **get()** ve **put()** gibi fonksiyonları kullanarak güvenli dizi oluşturma işlemini basit bir şekilde gerçekleştiriyorduk. Burada aşırı-yüklediğimiz **[]** operatörünü de kullanarak güvenli dizi oluşturmanın daha iyi bir yöntemini öğreneceğiz. Güvenli dizinin, sınır denetimi yapan bir sınıf içerisinde yerleştirilmiş dizi olduğunu hatırlayın. Bu yöntemle dizinin sınırlarının aşılması engelleriz. **[]** operatörünü aşırı-yüklemeyecek bu tür bir diziye normal bir dizi gibi erişebiliriz.

Güvenli dizi oluşturmak için **operator[]()** fonksiyonuna sınır denetimi ekleyin. **operator[]()** fonksiyonumuz indekslenen elemana referans döndürmelidir. Örneğin, yukarıdaki programa sınır kontrolü ekleyerek aşağıdaki programı oluşturduk. Bu program sınır aşıldığında bir hata vererek çalıştığını ispatlamaktadır:

```

// Güvenli dizilere bir örnek.
#include <iostream>
#include <cstdlib>
using namespace std;
const int SIZE = 5;

class arraytype {
    int a[SIZE];
public:
    arraytype() {
        int i;
        for(i=0; i<SIZE; i++) a[i] = i;
    }
    int operator[](int i);
};

// arraytype için sınır denetimi sağlanıyor.
int arraytype::operator[](int i)
{
    if(i<0 || i> SIZE-1) {
        cout << "\n İndeks değeri ";
        cout << i << " sınırı aşmaktadır.\n";
        exit(1);
    }
}

```

```

    }
    return a[i];
}
int main()
{
    arraytype ob;
    int i;
    // bu is OK
    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";
    /* Burada bir hata meydana gelir, çünkü SIZE+100 sınırın dışına çıkarıyor */
    ob[SIZE+100] = 99; // hata!
    return 0;
}

```

Bu programda,

```
ob[SIZE+100] = 99;
```

deyimi işletildiğinde, sınır hatası **operator[]()** tarafından engellenir ve program herhangi bir zarar meydana gelmeden sonlandırılır.

[] operatörünü aşırı-yüklememiz, tamamen normal diziler gibi görünen ve davranış gerekliliklerini sağlayan güvenli diziler oluşturmak için yeterlidir. Fakat dikkatli olun. Güvenli diziler bazı sorumlular da neden olacaktır. Hatta bu nedenle C++'da dizilere ilk başta sınır denetlemesi konulmamıştır. Fakat sınır hatası meydana gelmeyeceğinden emin olduğunuz uygulamalarda güvenli diziler çok işinize yarayacak.

Aliştırmalar

1. Konu 6.6'daki Örnek 1'i **strtype**, [] operatörü aşırı-yüklenenek şekilde değiştirein. Bu operatör belirlenen indeksteki karakteri döndürsün. Aynı zamanda [] operatörünün atama deyiminin sol tarafında da kullanılmasını sağlayın. Bu operatörü kullanarak çalışıp çalışmadığını kontrol edin.
2. Konu 6.6'daki Alistırma 1'e verdığınız cevabı, []'ı dinamik diziyi indekslemek için kullanacak şekilde değiştirein. Yani **get()** ve **put()** fonksiyonlarını [] operatörü ile değiştirin.

Pekiştirme Testi

Bu noktada aşağıdaki alıştırmaları yapabiliyor ve soruları cevaplayabiliyor olmalısınız.

1. **coord** sınıfı için **>>** ve **<<** kaydırma operatörlerini aşağıdaki tipte işlemleri gerçekleştirebilecek şekilde aşırı-yükleyin:

```
ob << tamsayı
ob >> tamsayı
```

Operatörleriniz **x** ve **y** değerlerini belirlenen miktarda kaydırsın.

2. Size aşağıdaki sınıf veriliyor:

```
class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
    {
        x = i; y = j; z = k;
    }
    three_d() { x=0; y=0; z=0; }
    void get(int &i, int &j, int &k)
    {
        i = x; j = y; k = z;
    }
};
```

`+, -, ++` ve `--` operatörlerini bu sınıf için aşırı-yükleyin. (Arttırma ve eksiltme operatörlerini sadece terimin operatörün önüne geldiği durum için aşırı-yükleyin.)

3. Soru 2'yi operatör fonksiyonlarına değer parametreleri yerine referans parametreleri kullanarak yeniden cevaplayın. İpucu: arttırma ve eksiltme operatörleri için arkadaş fonksiyonlarını kullanmanız gerekecek.
4. Arkadaş operatör fonksiyonlarının üye operatör fonksiyonlarından farkı nedir?
5. Atama operatörünü neden aşırı-yüklemek zorunda kalabileceğinizi açıklayın.
6. `operator=()` bir arkadaş fonksiyonu olabilir mi?
7. `+` operatörünü Soru 2'deki `three_d` sınıfı için aşağıdaki tipte işlemleri gerçekleştirilecek şekilde aşırı-yükleyin:

```
ob + int;
int + ob;
```

8. `==`, `!=`, ve `||` operatörlerini Soru 2'deki `three_d` sınıfı için aşırı-yükleyin.
9. `[]` operatörünün aşırı-yüklenmesinin ana nedenini açıklayın.

Bütünleştirme Testi

Burada bir önceki bölümle bu bölümde öğrendiklerinizi ne kadar iyi bir araya getirebildiğiniz kontrol edilmektedir.

1. Aşağıdaki tipte işlemlerin gerçekleştirilebileceği bir `strtype` sınıfı oluşturun:

- `+` operatörlerinin kullanılmasıyla katar birleştirme işlemi
- `=` operatörünün kullanılmasıyla katar ataması işlemi
- `<, >` ve `==` operatörlerinin kullanılmasıyla katar karşılaştırılması işlemi

Sabit uzunlukta katarlar kullanmakta serbestsiniz. Bu zor bir atamadır, fakat biraz düşünerek (ve deneyerek) yapabilirsiniz.

www.gerokku.com

BÖLÜM 7

Miras

Temel Sınıflara Erişim Denetimi

Protected Üyelerin Kullanımı

Yapıcılar, Yok-Ediciler ve Miras

Çoklu Miras

Sanal Temel Sınıflar

Miras kavramıyla daha önce tanışmışsınız. Şimdi onu biraz daha yakından inceleyelim. Miras, OOP'un üç ilkesinden biridir ve tabii böyle olunca bu kavram C++'ın da önemli bir özelliği olarak karşımıza çıkmaktadır. Miras, sadece hiyerarşik sınıflandırma kavramının desteklenmesini sağlamakla kalmaz, aynı zamanda Bölüm 10'da öğreneceğiniz gibi OOP'un bir diğer ilkesini, çok-bağımlılığı de destekler. Bu bölümde şu konuları ele alacağız: Temel sınıflara erişim kontrolü ve **protected** erişim belirticisi, çoklu temel sınıfları miras almak, temel sınıf yapıcılarına argüman gönderilmesi ve sanal temel sınıflar.

Gözden Geçirme Testi

Bu bölümme başlamadan önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

1. Operatörler aşırı yükleniklerinde işlevlerinden herhangi birini kaybederler mi?
2. Operatörler de sınıflar gibi, kullanıcı tarafından aşırı yüklenmiş tipler için aşırı yüklenmeli midir?
3. Aşırı yüklenmiş bir operatörün önceliği ve terim sayısı değiştirilebilir mi?
4. Size aşağıdaki program veriliyor, gerekli operatör fonksiyonlarını tamamlayın:

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

class array {
    int nums[10];
public:
    array();
    void set(int n[10]);
    void show();
    array operator+(array ob2);
    array operator-(array ob2);
    int operator==(array ob2);
};

array::array()
{
    int i;
    for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
    int i;

    for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
    int i;

    for(i=0; i<10; i++)
        cout << nums[i] << ' ';
```

```

    cout << "\n";
}

// Operatör fonksiyonlarını tamamlayın.

int main()
{
    array o1, o2, o3;

    int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    o1.set(i);
    o2.set(i);

    o3 = o1 + o2;
    o3.show();

    o3 = o1 - o3;
    o3.show();

    if(o1==o2) cout << "o1, o2'ye eşit\n";
    else cout << "o1, o2'ye eşit değil\n";

    if(o1==o3) cout << "o1, o3'e eşit\n";
    else cout << "o1, o3'e eşit değil\n";

    return 0;
}

```

Aşırı yüklenmiş + operatörü, her iki terimin elemanlarını toplasın. Aşırı yüklenmiş - operatörü, sol terimden sağ terimi çıkarsın. Aşırı yüklenmiş == operatörü ise terimlerin elemanları eşitse true, değilse false döndürsün.

5. Aşılıtmı 4'e verdığınız cevabı, arkadaş fonksiyonlarını kullanarak operatörleri aşırı yükleyecek şekilde değiştirin.
6. Aşılıtmı 4'deki sınıf ve fonksiyonları kullanarak ++ operatörünü bir üye fonksiyon, -- operatörünü ise bir arkadaş fonksiyonu ile aşırı yükleyin. ++ ve -- operatörleri sadece terimin operatörden önce geldiği durum için aşırı yüklenecek.
7. Atama operatörleri arkadaş fonksiyonları kullanılarak aşırı yüklenebilir mi?

7.1. Temel Sınıflara Erişim Kontrolü

Bir sınıf, diğer bir sınıfın mirasla oluşturulurken şu genel notasyon kullanılır :

```

class türetilmiş-sınıf-adı : erişim-turu temel-sınıf-adı {
    // ...
}

```

Burada *erişim-türü* yerine şu üç anahtar kelimedenden bir tanesini yazacağımız: **public**, **private** veya **protected**. **protected** erişim belirticisini bir sonraki konuda öğreneceğiz. Şimdi de diğer ikisini inceleyelim.

Erişim belirticisi ile, temel sınıfın elemanlarının türetilmiş sınıf tarafından ne şekilde alınacağını belirleriz. Mirasla özellikleri alınacak olan temel sınıfın erişim belirticisi **public** ise, temel sınıfın tüm üyeleri türetilmiş sınıfın da **public** (genel) üyeleridir. Erişim belirticisi **private** ise temel sınıfın tüm **public** üyeleri türetilmiş sınıfın **private** (özel) üyeleri olur. Her iki durumda da temel sınıfın tüm **private** üyeleri ona **private** kalır ve bu üyeler türetilmiş sınıf tarafından erişilemez.

Burada önemli nokta şu: Erişim belirticisi **private** ise temel sınıfın **public** üyeleri, türetilmiş sınıfın **private** üyeleri haline gelir ve bu üyeler, türetilmiş sınıfın üye fonksiyonları tarafından erişilebilir.

Teknik açıdan, *erişim-türüni*n yazılması sizin seçimminize bağlıdır. Eğer belirticiyi yazmazsanız ve türetilmiş sınıf bir **class** ise *erişim-türü* varsayılan olarak **private**'tir. Türetilmiş sınıf **struct** (yapı) ise ve yine erişim belirticisi yoksa varsayılan erişim türü **public**'tir. Aslında programcı olarak karışıklık çökmesin diye *erişim-türüni* kesin bir şekilde belirtmemizde yarar var.

Örnekler

1. Burada bir temel sınıf ve bu sınıfın mirasla oluşturulmuş (public olarak) bir türetilmiş sınıf verilmektedir:

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

class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};

// Public türü ile miras.
class derived : public base {
    int y;
public:
    void sety(int n) { y = n; }
    void showy() { cout << y << '\n'; }
};

int main()
{
    derived ob;

    ob.setx(10); // temel sınıfın üyelerine erişiyor
    ob.sety(20); // türetilmiş sınıfın üyelerine erişiyor

    ob.showx(); // temel sınıfın üyelerine erişiyor
    ob.showy(); // türetilmiş sınıfın üyelerine erişiyor

    return 0;
}
```

Bu programda gördük ki, **base** temel sınıfının özelliklerini mirasla public türü ile aldık ve bu nedenle **base-setx()** ve **showx()**’in public üyeleri, **derived** sınıfının da public üyeleridir. Böylece programın herhangi bir bölümünden onlara erişebiliyoruz. Bu üyeleri, **main()** fonksiyonunun içerisinde kurala uygun şekilde çağrırlılar.

2. Diyelim ki türetilmiş sınıf, temel sınıfın üyelerini mirasla public erişim-türü ile alınsın. Fakat bu durum, türetilmiş sınıf temel sınıfın private üyelerine erişebilir anlamına gelmez. Örneğin, yukarıdaki **derived** sınıfına yaptığımız eklemeler yanlışır:

```
class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};

// Public türünde mirasla alınıyor, burada bir hata var!
class derived : public base {
    int y;
public:
    void sety(int n) { y = n; }
    /* Temel sınıfın private üyelerine erişilemez. X, temel sınıfın private bir
       üyesidir ve türetilmiş sınıf içerisinde bu değişkene erişilemez.*/
    void show_sum() { cout << x+y << '\n'; } // Hata!

    void showy() { cout << y << '\n'; }
};
```

Bu örnekte **derived** sınıfı, **base**’in private üyesi olan **x**’e erişmeye teşebbüs etmektedir. Bu bir hatadır, çünkü temel sınıfa ait private üyeleri, *miras ne şekilde yapılsın o sınıfa özel kahr*.

3. Burada Örnek 1’deki programı biraz değiştirerek bir daha inceleyelim, bu defa **derived** sınıfını **base** sınıfından private olarak türetiyoruz. Bu değişiklik yüzünden program içerisindeki açıklamalarda da belirtildiği gibi programımız hatalı hale gelmiştir.

```
// Bu program bir hata içermektedir.
#include <iostream>
using namespace std;

class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};

// Temel sınıftan private olarak base sınıfı türetiliyor.
class derived : private base {
    int y;
public:
    void sety(int n) { y = n; }
    void showy() { cout << y << '\n'; }
};
```

```

int main()
{
    derived ob;
    ob.setx(10); // HATA - şimdi türetilmiş sınıf'a private
    ob.sety(20); // türetilmiş sınıfın ıyesine erişiyor - OK
    ob.showx(); // HATA - şimdi türetilmiş sınıf'a private
    ob.showy(); // türetilmiş sınıfın ıyesine erişiyor - OK
    return 0;
}

```

Hatalı programdaki açıklamalardan da görebileceğimiz gibi **showx()** ve **setx()** fonksiyonları **derived** sınıfına private'tır ve bu fonksiyonlara bu sınıfın dışından erişilemez. Türetilmiş sınıflar tarafından özellikleri ne şekilde alınırsa alınsın **showx()** ve **setx()**, **base** sınıfı içerisinde hala public'tır, buna dikkat edin. Bu da, **base** tipinde nesneler bu fonksiyonlara her yerden erişebilir demektir. Fakat bu nesneler **derived** tipine ait nesnelere göre private'tırlar. Aşağıdaki örneği inceleyelim:

```

base base_ob;
base_ob.setx(1); // Bu, kurala uygun, çünkü base_ob, base tipinde
setx()'in çağrılması kurala uygun, çünkü setx(), base içerisinde public'tır.

```

4. Önceden belirttiğimiz gibi temel sınıfların public üyeleri, **private** belirticisi kullanılarak miras gerçekleştirildiğinde, türetilmiş sınıfların private üyeleri olur, buna rağmen bu üyelere türetilmiş sınıf içerisinde hala erişilebilir. Yukarıdaki programın "hatası nasıl düzeltilmiş", inceleyelim:

```

// Bu programın hatası düzeltilmiştir.
#include <iostream>
using namespace std;

class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};

// Temel sınıfın özellikleri private olarak alınmıştır.
class derived : private base {
    int y;
public:
    // setx'e derived sınıfı içerisinde erişilebilir
    void setxy(int n, int m) { setx(n); y = m; }
    // showx'e derived sınıfı içerisinde erişilebilir
    void showxy() { showx(); cout << y << '\n'; }
};

int main()
{
    derived ob;
    ob.setxy(10, 20);
    ob.showxy();
    return 0;
}

```

Bu durumda, `setx()` ve `showx()` fonksiyonlarına türetilmiş sınıf içerisinde erişebiliriz. Bu tamamen kurallara uygundur; çünkü bu fonksiyonlar sınıfın private üyeleriidir.

Aliştırmalar

1. Aşağıdaki programı inceleyelim:

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

class mybase {
    int a, b;
public:
    int c;
    void setab(int i, int j) { a = i; b = j; }
    void getab(int &i, int &j) { i = a; j = b; }
};

class derived1 : public mybase {
    // ...
};

class derived2 : private mybase {
    // ...
};

int main()
{
    derived1 o1;
    derived2 o2;
    int i, j;
    // ...
}
```

Sizce `main()` içerisindeki bildirilerden hangisi doğru?

- A. `o1.getab(i, j);`
 - B. `o2.getab(i, j);`
 - C. `o1.c = 10;`
 - D. `o2.c = 10;`
2. Public üyeleri, public olarak mirasla alınırsa ne olur, private olarak alınırsa ne olur?
 3. Eğer henüz yapmadıysanız, konu içerisinde verilen tüm örnekleri deneyin. Erişim belirticileri ile ilgili çeşitli denemeler yapın ve sonuçları gözlemleyin.

7.2. Protected Üyelerin Kullanımı

Bir önceki konuda da görmüştük, türetilmiş sınıf temel sınıfın private üyelerine erişemez. Yani, türetilmiş sınıfın temel sınıf'a ait bazı üyelerle erişmesini istiyorsak bu üye public olmak zorundadır. Fakat temel sınıf'a private kalmasını istediğiniz bir üyenin türetilmiş sınıf tarafından erişilmesini istediğimiz zamanlar olacak. Bunu sağlamamız için C++ bize `protected` erişim belirticisini sunmaktadır.

Protected erişim belirticisi, **private** belirticisi ile eşdeğerdir, tek bir farkla, temel sınıfa ait protected üyelerle, bu sınıfın türetilmiş sınıflara ait üyeleri erişebilir. Temel sınıfın veya türetilmiş sınıfların dışından bu protected üyelerle erişilemez. **protected** erişim belirticisi sınıf bildirimini içerisinde herhangi bir yerde bulunabilir. Fakat genelde private üyelerden sonra ve public üyelerden önce verilir. Sınıf bildirimini şu şekilde yapacağız:

```
class sınıf-adı {
    // private üyeleri
    protected: // opsiyonel
    // protected üyeleri
public:
    // public üyeleri
};
```

Temel sınıfa ait protected üyeleri türetilmiş sınıf tarafından public olarak mirasla alınmışlarsa, bu üyeleri türetilmiş sınıfın da protected üyeleriidir. Eğer protected üyeleri temel sınıfın public olarak türetilmişlerse bu üyeleri türetilmiş sınıfın private üyesi olur.

Türetilmiş sınıfı temel sınıfın protected olarak da türetebiliriz. Eğer durum böyleyse temel sınıfın public ve protected üyeleri türetilmiş sınıfın da protected üyeleri olur. Tabii, temel sınıfın private üyeleri ona özel kalır bu üyelerle türetilmiş sınıfın erişemeyiz. **protected** erişim belirticini yapılarla çalışırken de kullanabiliriz.

Örnekler

1. Bu programda sınıfı ait public, private ve protected üyelerine nasıl erişileceğini görelim:

```
#include <iostream>
using namespace std;
class samp {
    // private by default
    int a;
protected: // hala samp için private
    int b;
public:
    int c;

    samp(int n, int m) { a = n; b = m; }
    int geta() { return a; }
    int getb() { return b; }
};

int main()
{
    samp ob(10, 20);

    // ob.b = 99; Hatalı! B, protected olup burada private'dır
    ob.c = 30; // OK, çünkü c, public
    cout << ob.geta() << ' ';
    cout << ob.getb() << ' ' << ob.c << '\n';

    return 0;
}
```

Gördüğümüz gibi `main()`'deki açıklamanın içine yazılmış olan satır hatalıdır, çünkü **b** protected'tir ve bu yüzden hala **samp'e private'tır.**

- Protected üyeleri public olarak türettiğimizde neler olduğunu aşağıdaki örnekte inceleyelim:

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

class base {
protected: // base sınıfına private
    int a, b; // ama türetilmiş sınıf bu değişkenlere hala erişebilir
public:
    void setab(int n, int m) { a = n; b = m; }
};

class derived : public base {
    int c;
public:
    void setc(int n) { c = n; }

    // bu fonksiyon temel sınıf içerisinde a ve b'ye erişebilir.
    void showabc() {
        cout << a << ' ' << b << ' ' << c << '\n';
    }
};

int main()
{
    derived ob;

    /* a ve b'ye buradan erişilemez, çünkü y, hem temel sınıfa hem de türetilmiş
       sınıf'a private'dır. */

    ob.setab(1, 2);
    ob.setc(3);

    ob.showabc();

    return 0;
}
```

a ve **b**, **base**'in içerisinde protected olduğundan ve **derived** tarafından public olarak alındıklarından **derived** sınıfına ait üye fonksiyonlarda kullanabiliriz. Fakat bu iki sınıfın dışında, **a** ve **b** kesinlikle private'tır ve onlara erişilemez.

- Daha önce de belirtmiştık, türetilmiş sınıfı protected olarak oluşturmuşsak temel sınıf'a ait public ve protected üyeleri türetilmiş sınıfın protected üyeleri haline gelir. Örneğin, yukarıdaki programda public olarak oluşturduğumuz türetilmiş sınıfı burada protected olarak oluşturuyoruz:

```
// Bu program derlenmeyecek.
#include <iostream>
using namespace std;

class base {
protected: // base sınıfına private
```

```

int a, b; // fakat derived sınıfı hala bu değişkenlere erişebilir
public:
    void setab(int n, int m) { a = n; b = m; }

class derived : protected base { // özellikleri protected olarak alınıyor
    int c;
public:
    void setc(int n) { c = n; }

    // bu fonksiyon base'in içerisinde a ve b'ye erişebilir
    void showabc() {
        cout << a << ' ' << b << ' ' << c << '\n';
    }
}

int main()
{
    derived ob;

    // HATA: şimdi setab(), temel sınıfın protected üyesi.
    ob.setab(1, 2); // setab()'a buradan erişemeyiz.
    ob.setc(3);
    ob.showabc();

    return 0;
}

```

Tıpkı açıklamalarda gördüğümüz gibi, türetilmiş sınıfı **base**'den protected olarak oluşturduğumuzdan dolayı, temel sınıfın public ve protected elemanları **derived** sınıfının protected üyeleri olur ve bu yüzden bu elemanlara **main()**'in içerisinde erişilemez.

Alıştırmalar

- Protected üyeleri mirasla public olarak, private olarak ve protected olarak alırsak ne olur?
- Neden protected kategorisine gerek duyuyoruz, açıklayın.
- Konu 7.1 Alıştırma 1'deki **myclass** sınıfının içerisindeki **a** ve **b** değişkenleri private yerine protected olsaydı bu alıştırmaya verdığınız cevaplar değişim miydi? Cevabınızı değiştirmesi gerekiyorsa nasıl değiştirdi?

7.3. Yapıcılar, Yıkıcılar ve Miras

Temel sınıflar, türetilmiş sınıflar veya her ikisinin bir yapıçı ve/ya yıkıcı fonksiyonu olması mümkün değildir. Burada bu durumla ilgili çeşitli konuları inceleyeceğiz. Temel sınıfın ve türetilmiş sınıfın yapıçı ve yıkıcı fonksiyonları varsa bu yapıçı fonksiyonları sınıfın oluşturulma sırasına göre işletilir. Yıkıcı fonksiyonları ise bunun tersi sırada işletilir. Yani temel sınıfa ait yapıçı, türetilmiş sınıfa ait yapıçıdan önce çalışır. Yıkıcı fonksiyonları için ise bunun tersi doğrudur, türetilmiş sınıfın yıkıcısı temel sınıfın yıkıcısından önce çalışır.

Eğer üzerinde biraz düşünecek olursak yapıcı fonksiyonlarının oluşum sırasına göre çalıştırılmasının nedenini anlayabiliriz. Temel sınıf kendisinden oluşturulan türetilmiş sınıfların hiçbirini hakkında bilgiye sahip değildir. Bu nedenle temel sınıfın yaptığı hazırlık işlemleri türetilmiş sınıfın hazırlık işlemlerinden hem apayrıdır, hem de türetilmiş sınıfın hazırlıklarının yapılması için temel sınıfın hazırlıklarının yapılması gereklidir. Sonuçta ilk olarak çalıştırılmalıdır.

Öte yandan türetilmiş sınıfın yıkıcısı temel sınıfın yıkıcısından çalışır. Çünkü temel sınıf türetilmiş sınıfın esasıdır. Eğer temel sınıfın yıkıcısı ilk çalıştırılıyorsa olsaydı, bu fonksiyon türetilmiş sınıfın yok edilmesini gerektirecekti. Bu nedenle, türetilmiş sınıfın yıkıcısı nesne yok edilmenden önce çağrılmalıdır.

Şimdiye kadar incelediğimiz örneklerin hiçbirinde herhangi bir türetilmiş veya temel sınıf ait yapıcı fonksiyonuna argüman göndermedik. Fakat bunu yapabiliyoruz. Sadece türetilmiş sınıf için hazırlık yapıyorsak argümanları türetilmiş sınıfın yapıcısına normal bir şekilde göndeririz. Fakat temel sınıfın yapıcısına argüman göndermemiz gerekiyorsa biraz daha fazla uğraşmamız gerekecektir. Bu iş için öncelikle bir argüman gönderme zinciri kurmalıyız. İlk olarak gerek temel sınıf'a gerekse türetilmiş sınıf'a göndermek durumunda olduğumuz tüm argümanları türetilmiş sınıfın yapıcısına göndeririz. Türetilmiş sınıf'a ait yapıcı bildirimini daha geniş bir amaçla kullanınız ve sonra temel sınıf'a ona göndereceğimiz argümanları göndeririz. Türetilmiş sınıf'tan temel sınıf'a argüman gönderirken kullanmanız gereken genel notasyonu inceleyelim:

```
türetilmiş-yapıcı (argüman-listesi) : temel-sınıf(argüman-listesi) {
    // türetilmiş sınıfın yapıcısı
}
```

Burada *temel-sınıf* yerine temel sınıfın adını yazacağımız. Türetilmiş sınıfta temel sınıfın aynı argümanı kullanmasında bir sakınca yoktur. Aynı zamanda türetilmiş sınıfın tüm argümanları görmezden gelmesi ve onları doğrudan temel sınıf'a göndermesi de mümkündür.

Örnekler

- Aşağıdaki örneği inceleyerek temel sınıfın ve türetilmiş sınıfın yapıcı ve yıkıcı fonksiyonlarının ne zaman çalıştırıldığıni görelim:

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

class base {
public:
    base() { cout << "Temel sınıf oluşturuluyor\n"; }
    ~base() { cout << "Temel sınıf yok ediliyor\n"; }
};
class derived : public base {
public:
    derived() { cout << "Türetilmiş sınıf oluşturuluyor\n"; }
    ~derived() { cout << "Türetilmiş sınıf yok ediliyor\n"; }
};
```

```
int main()
{
    derived o;
    return 0;
}
```

Bu program bize aşağıdaki sonucu verecek:

```
Temel sınıf oluşturuluyor
Temel sınıf yok ediliyor
Türetilmiş sınıf oluşturuluyor
Türetilmiş sınıf yok ediliyor
```

Gördüğünüz gibi, yapıcı fonksiyonları sınıfın oluşturulmuş sırasına göre yıkıcı fonksiyonları ise bunun tam tersi sıradır çalıştırılırlar.

2. Burada türetilmiş sınıfının yapıcısına nasıl argüman göndereceğimizi göreceğiz:

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

class base {
public:
    base() { cout << "Temel sınıf oluşturuluyor\n"; }
    ~base() { cout << "Temel sınıf yok ediliyor\n"; }
};

class derived : public base {
    int j;
public:
    derived(int n) {
        cout << "Türetilmiş sınıf oluşturuluyor\n";
        j = n;
    }
    ~derived() { cout << "Türetilmiş sınıf yok ediliyor\n"; }
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10);
    o.showj();
    return 0;
}
```

Argümanı türetilmiş sınıfın yapıcısına normal bir şekilde gönderiyoruz, buna dikkat edin.

3. Aşağıdaki örnekte, hem türetilmiş sınıfın hem de temel sınıfın yapıcları argüman alıyor. Bu özel durum içerisinde her iki sınıf da aynı argümanı kullanıyor ve türetilmiş sınıf argümanı doğrudan temel sınıf'a gönderiyor.

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

class base {
    int i;
public:
```

```

base(int n) {
    cout << "Temel sınıf oluşturuluyor\n";
    i = n;
}
~base() { cout << "Temel sınıf yok ediliyor\n"; }
void showi() { cout << i << '\n'; }
};

class derived : public base {
    int j;
public:
    derived(int n) : base(n) { //argüman temel sınıf'a gönderilir
        cout << "Türetilmiş sınıf oluşturuluyor\n";
        j = n;
    }
    ~derived() { cout << "Türetilmiş sınıf yok ediliyor\n"; }
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10);

    o.showi();
    o.showj();

    return 0;
}

```

derived sınıfının yapıcı fonksiyonuna özellikle dikkat edelim. Hazırlık için kullanılan argümanı alan **n** parametresini hem **derived()** içerisinde kullanıyoruz hem de onu **base()** fonksiyonuna gönderiyoruz.

4. Çoğu durumda temel sınıf için ve türetilmiş sınıflar için oluşturduğumuz yapıcı fonksiyonları aynı argümanı *kullanmayacaktır*. Durum böyleyse ve her bir fonksiyona bir veya daha fazla argüman göndermemiz gerekiyorsa *hem* türetilmiş sınıf ve *hem de* temel sınıf tarafından kullanılan argümanların *tümünü* türetilmiş sınıfın yapıcısına göndereceğiz. Daha sonra türetilmiş sınıf, temel sınıf tarafından kullanılan argümanları ona aktaracaktır. Aşağıdaki örneğimizi inceleyelim ve türetilmiş sınıfın yapıcısı ile temel sınıfın yapıcısına nasıl argüman gönderebileceğimizi daha iyi anlamaya çalışalım:

```

#include <iostream>
using namespace std;

class base {
    int i;
public:
    base(int n) {
        cout << " Temel sınıf oluşturuluyor\n";
        i = n;
    }
    ~base() { cout << "Temel sınıf yok ediliyor\n"; }
    void showi() { cout << i << '\n'; }
};

```

```

class derived : public base {
    int j;
public:
    derived(int n, int m) : base(m) { // argüman temel sınıf'a aktarılıyor
        cout << "Türetilmiş sınıf oluşturuluyor\n";
        j = n;
    }
    ~derived() { cout << "Türetilmiş sınıf yok ediliyor\n"; }
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10, 20);

    o.showi();
    o.showj();

    return 0;
}

```

5. Türetilmiş sınıfa ait yapıcının temel sınıf'a göndereceği argümanı gerçekten kullanması gerekmekz. Eğer türetilmiş sınıfımızın argümanına ihtiyacı yoksa kendisine gönderilen argümanı görmezden gelir ve onu doğrudan temel sınıf'a aktarır. Örneğin, aşağıdaki programın parametresi `derived()` fonksiyonumuz tarafından kullanılmıyor ve bu nedenle de doğrudan `base()` fonksiyonuna aktarılıyor:

```

class base {
    int i;
public:
    base(int n) {
        cout << "Temel sınıf oluşturuluyor\n";
        i = n;
    }
    ~base() { cout << "Temel sınıf yok ediliyor\n"; }
    void showi() { cout << i << '\n'; }
};

class derived : public base {
    int j;
public:
    derived(int n) : base(n) { // argüman temel sınıf'a aktarılıyor
        cout << "Türetilmiş sınıf oluşturuluyor\n";
        j = 0; //Then not used here
    }
    ~derived() { cout << "Türetilmiş sınıf yok ediliyor\n"; }
    void showj() { cout << j << '\n'; }
};

```

Alıştırmalar

1. Size aşağıdaki program veriliyor, buradaki `myderived` sınıfına ait yapıcı fonksiyonunu tamamlayın. Bu fonksiyon, `mybase`'nın hazırlık katarına bir işaretçi göndersin ve katarın uzunluğunu da `len` değişkenine atasın.

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

class mybase {
    char str[80];
public:
    mybase(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

class myderived : public mybase {
    int len;
public:
    // myderived() fonksiyonunu buraya ekleyin
    int getlen() { return len; }
    void show() { cout << get() << '\n'; }
};

int main()
{
    myderived ob("merhaba");

    ob.show();
    cout << ob.getlen() << '\n';

    return 0;
}
```

2. Size aşağı verilen programı kullanarak **car()** ve **truck()** yapıcı fonksiyonlarını düzgün bir şekilde oluşturun. Her iki fonksiyon da **vehicle**'a göndermesi gereken argümanları göndersin. Bunlara ek olarak **car()** fonksiyonu, nesne oluşturulduğu zaman **passengers** için gerekli hazırlıkları yapsın. **truck()** fonksiyonu ise **loadlimit** için aynı şekilde hazırlık yapsın.

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

// Çeşitli araç tipleri için bir temel sınıf.
class vehicle {
    int num_wheels;
    int range;
public:
    vehicle(int w, int r)
    {
        num_wheels = w; range = r;
    }
    void showv()
    {
        cout << "Tekerlekler: " << num_wheels << '\n';
        cout << "Menzil: " << range << '\n';
    }
};
class car : public vehicle {
    int passengers;
public:
    // car() yapıcısını buraya ekleyin
    void show()
    {
```

```

showv();
cout << "Yolcular: " << passengers << '\n';
}
}

class truck : public vehicle {
    int loadlimit;
public:
    // truck() yapıcısını buraya ekleyin
    void show()
    {
        showv();
        cout << "yolcu-kapasitesi " << loadlimit << '\n';
    }
}

int main()
{
    car c(5, 4, 500);
    truck t(30000, 12, 1200);

    cout << "Araba: \n";
    c.show();
    cout << "\nKamyon: \n";
    t.show();

    return 0;
}

```

car() ve **truck()** fonksiyonları nesneleri aşağıdaki gibi deklare etsin:

```

car ob(passengers, wheels, range);
truck ob(loadlimit, wheels, range);

```

7.4. Çoklu Miras

Türetilmiş sınıfımızın birden fazla temel sınıfın özelliklerini mirasla alabilmesi için iki yöntem var. Birinci yöntemde türetilmiş sınıfların bir diğer türetilmiş sınıfın oluşturulmasında temel sınıf gibi rol oynayabilmesinden yararlanacağız ve bu şekilde çok seviyeli bir sınıf hiyerarşisi oluşturmuş oluruz. Bu durumda, ilk temel sınıfa ikinci türetilmiş sınıfın *dolaylı* temel sınıfı denir. Sunu akılmızdan çıkarmamamız gerekiyor, her sınıf ne şekilde oluşturulmuş olursa olsun, bir temel sınıf gibi kullanılabilir. İkinci yöntemimizde ise türetilmiş sınıfın birden fazla temel sınıfından doğrudan oluşturulabilme özelliğinden faydalanaćagız. Bu yöntemde iki veya daha fazla sınıf, türetilmiş sınıfı oluşturmak üzere bir araya getirilir. Burada birden çok temel sınıfta ne şekilde çalışacağımızı hep birlikte öğreneceğiz.

Bir temel sınıfımız ve bu temel sınıfından türetilmiş bir sınıfımız var. Bu türetilmiş sınıfı da oluşturacağımız bir başka türetilmiş sınıf için temel sınıf olarak kullanacağız. Burada bu üç sınıfın yapıçı fonksiyonlarını sınıfların oluşturuluş sırasına göre çağıracağız. (Bu genel prensibi bir önceki konuda öğrenmiştık.) Yıkıcı fonksiyonları ise oluşturulmuş sırasının tersi sıradan çağrılacak. Böylelikle eğer *B1* sınıfı *D1*sinden türetilmişse ve *D1* de *D2* sınıfından türetilmişse, *B1*'in yapıcısı ilk olarak çağrıılır, daha sonra *D1*'in, en sonda da *D2*'nin yapıcısını çağrıriz. Yıkıcı fonksiyonlarımıza ise bunun tersi sıradan çağrıracagız.

Türetilmiş sınıfımızı doğrudan birden fazla temel sınıfından oluşturuyorsak aşağıdaki bildirimi kullanacağız:

```
class türetilmiş-sınıf-adi : erişim-türü temeli1, erişim-türü temeli2, ...
erişim-türü temeliN
{
    // ... sınıfın gövdesi
}
```

Burada *temel1* ile *temelN* arasına temel sınıfların isimleri ve *erişim-türü* yerine ise erişim belirticisini yazacağız. Erişim belirticisi her temel sınıf için ayrı ayrı olabilir. Bu yöntemde yapıcılar soldan sağa doğru temel sınıfların veriliş sırasına göre çalıştırırlar. Yıkıcılar ise bunun tersi sırada işletilir.

Eğer bu yapıcılar argüman kullanıyorlarsa türetilmiş sınıflar bu fonksiyonlara gerekli argümanları şu şekilde aktarırlar:

```
türetilmiş-yapıcı (argümanlar) : temeli1(argümanlar), temeli2(argümanlar), ...
temeliN(argümanlar)
{
    // türetilmiş sınıfa ait yapıcı fonksiyonunun gövdesi
}
```

Burada *temel1* ile *temelN* arasına yine temel sınıfların isimleri gelecek. Her türetilmiş sınıfın, bir önceki temel sınıfına bu temel sınıfın kullandığı argümanları göstermesi gereklidir.

Örnekler

1. Bu örnekte bir temel sınıfından oluşturulmuş türetilmiş sınıfımızdan bir türetilmiş sınıf daha oluşturuyoruz. Burada D2 ile B1 arasındaki zincirde argümanları nasıl aktarıldığımıza dikkat edeceğiz.

```
// Çoklu Miras
#include <iostream>
using namespace std;

class B1 {
    int a;
public:
    B1(int x) { a = x; }
    int geta() { return a; }
};

// Temel sınıfından doğrudan oluşturuluyor.
class D1 : public B1 {
    int b;
public:
    D1(int x, int y) : B1(y) // y, B1'e gönderiliyor
    {
        b = x;
    }
    int getb() { return b; }
};
```

```

// Bir türetilmiş sınıfın ve bir indirekt temel sınıfın sınıf oluşturuluyor.
class D2 : public D1 {
    int c;
public:
    D2(int x, int y, int z) : D1(y, z) // argümanlar D1'e aktarılıyor
    {
        c = x;
    }

    /* Temel sınıflardan miras public olarak gerçekleştirildiğinden, D2, hem
       D1'in hem de D1'in public elemanlarına erişebilir. */
    void show() {
        cout << geta() << ' ' << getb() << '\n';
        cout << c << '\n';
    }
};

int main()
{
    D2 ob(1, 2, 3);
    ob.show();
    // geta() ve getb() fonksiyonları burada hala public
    cout << ob.geta() << ' ' << ob.getb() << '\n';

    return 0;
}

```

ob.show() fonksiyonu bize 3 2 1 sonucunu verecek. Bu örnekte, **B1**, **D2**'nin doğrudan temel sınıfıdır. **D2**'nin hem **D1**'in hem de **B1**'in public üyelerine erişebildiğine dikkat edelim. Hatırlayacaksınız, temel sınıfa ait public üyeleri public olarak mirasla aldığımda, türetilmiş sınıfın da public üyesi haline geliyordu. Sonuç olarak **D1'i**, **B1'den** türettiğimizde **geta()** fonksiyonu **D1**'in public üyesi olur, **D1** ise **D2**'nin public üyesidir.

Programda da gördük ki, sınıf hiyerarşisi içerisindeki her sınıf, bir önceki temel sınıfı tarafından gerek duyulan tüm argümanları ona göndermek zorundadır. Bu yapılmazsa bir derleme hatası ile karşı karşıya kalırız.

Programda oluşturduğumuz sınıf hiyerarşisini hep birlikte kullanalım:



Devam etmeden önce C++ miras grafiklerini nasıl çizeceğimizi öğrenelim. Yukarıdaki grafikte okların aşağı yerine yukarıyı gösterdiğine dikkat edelim. Normalde C++ programcılar genellikle miras grafiklerini çizerken okların yönü türetilmiş sınıfından temel sınıf'a doğrudur. C++'la yeni tanısanlar bazen bunun ters olduğunu düşünebilirler. Fakat yine de miras grafikleri C++'da bu şekilde gösterilir.

2. Burada yukarıda verdiğimiz programda biraz değişiklik yaptık. Türetilmiş sınıfımızı doğrudan iki temel sınıfından oluşturuyoruz:

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

// İlk temel sınıfı oluşturuyoruz.
class B1 {
    int a;
public:
    B1(int x) { a = x; }
    int geta() { return a; }
};

// İkinci temel sınıfı oluşturuyoruz.
class B2 {
    int b;
public:
    B2(int x)
    {
        b = x;
    }
    int getb() { return b; }
};

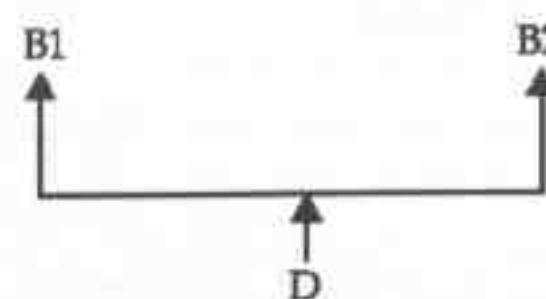
// Türetilmiş sınıfımızı bu iki temel sınıfından doğrudan oluşturuyoruz.
class D : public B1, public B2 {
    int c;
public:
    // Burada x ve y doğrudan B1 ve B2'ye aktarılıyor
    D(int x, int y, int z) : B1(z), B2(y)
    {
        c = x;
    }
    /* Miras temel sınıflardan public olarak gerçekleştirildiğinden, D, hem
       B1'in hem de B2'nin public elemlarına erişebilir. */
    void show()
    {
        cout << geta() << ' ' << getb() << ' ';
        cout << c << '\n';
    }
};

int main()
{
    D ob(1, 2, 3);

    ob.show();

    return 0;
}
```

Bu uyarlamada argümanlarımız **D** tarafından **B1** ve **B2**'ye tek tek gönderiliyor. Bu program, aşağıdaki gibi görünen bir sınıf yaratacaktır:



3. Bu uygulamada, türetilmiş sınıfımız birden çok temel sınıfın doğrudan oluşturulduğunda, yapıcı ve yıkıcı fonksiyonlarının hangi sırada çağrıldığını göreceğiz:

```

#include <iostream>
using namespace std;

class B1 {
public:
    B1() { cout << "B1 oluşturuluyor\n"; }
    ~B1() { cout << "B1 yok ediliyor\n"; }
};

class B2 {
    int b;
public:
    B2() { cout << "B2 oluşturuluyor\n"; }
    ~B2() { cout << "B2 yok ediliyor\n"; }
};

// İki temel sınıfın miras gerçekleştiriliyor.
class D : public B1, public B2 {
public:
    D() { cout << "D oluşturuluyor\n"; }
    ~D() { cout << "D yok ediliyor\n"; }
};

int main()
{
    D ob;
    return 0;
}
  
```

Programımız şu sonucu verecek:

```

B1 oluşturuluyor
B2 oluşturuluyor
D oluşturuluyor
D yok ediliyor
B2 yok ediliyor
B1 yok ediliyor
  
```

Öğrenmişti ki, birden çok temel sınıfın doğrudan miras gerçekleştirdiğimizde yapıcılar soldan sağa sınıfların miras listesinde veriliş sırasına göre çağrırlar. Yıkıcılar ise ters sırada çağrırlar.

Aliştırmalar

1. Aşağıdaki program bize nasıl bir sonuç verecektir? Bunu programı çalıştırmadan tahmin etmeye deneyin.

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

class A {
public:
    A() { cout << " A oluşturuluyor\n"; }
    ~A() { cout << " A yok ediliyor\n"; }
};

class B {
public:
    B() { cout << " B oluşturuluyor\n"; }
    ~B() { cout << " B yok ediliyor\n"; }
};

class C : public A, public B {
public:
    C() { cout << " C oluşturuluyor\n"; }
    ~C() { cout << " C yok ediliyor\n"; }
};

int main()
{
    C cb;
    return 0;
}
```

2. Aşağıdaki sınıf hiyerarşisini kullanarak **C**'nin yapıcı fonksiyonunu oluşturun, bu fonksiyon **k** için gerekli hazırlıkları tamamlasın, **A()** ve **B()** fonksiyonlarına argümanlarını göndersin.

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

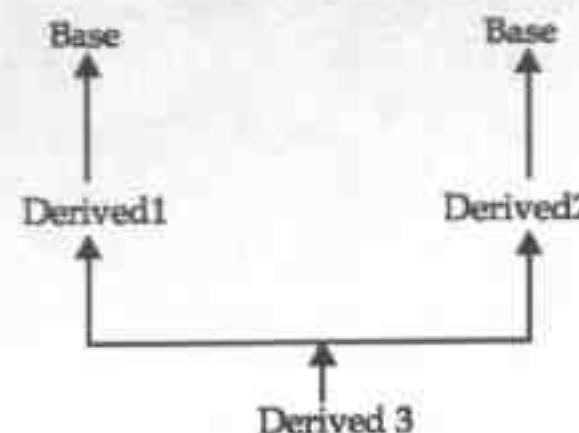
class A {
    int i;
public:
    A(int a) { i = a; }
};

class B {
    int j;
public:
    B(int a) { j = a; }
};

class C : public A, public B {
    int k;
public:
    /* C() fonksiyonu k için gerekli hazırlıkları yapacak, A() ve B()'ye
       argümanlar gönderecek şekilde oluşturuluyor */
};
```

7.5. Sanal Temel Sınıflar

Türetilmiş sınıfımızı birden çok temel sınıfından doğrudan türettiğimizde potansiyel bir problem ortaya çıkmaktadır. İsterseniz bu problemin ne olduğunu anlamak için aşağıdaki sınıf hiyerarşisine bir göz atalım:



Burada *Base* temel sınıfımızdan *Derived1* ve *Derived2* türetilmiş sınıflarımızı oluşturuyoruz. *Derived3* sınıfını ise doğrudan *Derived1* ve *Derived2* türetilmiş sınıflarından türetiliyoruz. Fakat, aslında *Base*'in özellikleri *Derived3* tarafından iki kere mirasla alınıyor, ilk olarak *Derived1* üzerinden ikinci olarak da *Derived2* üzerinden. *Base*'e ait üyeler *Derived3* tarafından kullanıldığında bu durum bir belirsizliğe neden olur. *Derived3*, *Base* sınıfına ait iki kopya içerdiği sürece, *Base*'e ait üyeye yaptığıımız gönderme acaba *Derived1* üzerinden miras alınan *Base*'e mi, yoksa *Derived2* üzerinden miras alınan *Base*'c mi yapılıyor? *Derived3* tarafından *Base*'in sadece tek bir kopyasının içeri alınması sağlanan mekanizma sayesinde bu belirsizliği açıklığa kavuşturabiliriz. Bu özelliği *sanal temel sınıf* olarak adlandırıyoruz.

Türetilmiş sınıfın, bir temel sınıfın özelliklerini dolaylı bir şekilde birden fazla kere alması durumunda türetilmiş nesnede bu temel sınıf'a ait iki adet kopyanın bulunmasını engellememiz mümkün. Türetilmiş sınıflarımızı bu temel sınıfından sanal olarak oluştururuz ve sorunu hallederiz. Bu sayede temel sınıfından dolaylı olarak türettiğimiz türetilmiş sınıflarda temel sınıfımızın iki veya daha fazla kopyası bulunmaz. **virtual** (sanal) anahtar kelimesi temel sınıfın erişim belirticisinden önce yazılır.

Örnekler

1. Burada *derived3* içerisinde *base*'e ait iki kopyanın bulunmasını engellemek için sanal temel sınıf kullanıyoruz.

```

// Bu programda sanal temel sınıf kullanılıyor.
#include <iostream>
using namespace std;

class base {
public:

```

```
int i;
};

// Temel sınıfından miras sanal olarak gerçekleştiriliyor.
class derived1 : virtual public base {
public:
    int j;
};

// Burada da miras temel sınıfından sanal olarak gerçekleştiriliyor.
class derived2 : virtual public base {
public:
    int k;
};

/* Burada, derived3 hem derived1'den hem de derived2'den oluşturuluyor. Fakat,
   ortada base'e ait tek bir kopya bulunmakta.
*/
class derived3 : public derived1, public derived2 {
public:
    int product() { return i * j * k; }
};

int main()
{
    derived3 ob;

    ob.i = 10; // belirsizlik yok, çünkü tek bir kopya mevcut
    ob.j = 3;
    ob.k = 5;

    cout << "Çarpım sonucu " << ob.product() << '\n';

    return 0;
}
```

Eğer **derived1** ve **derived2** sınıfları **base**'den sanal olarak türetilmemiş olsaydı,

```
ob.i = 10;
```

bildirisinde belirsizlik olacaktı ve bir derleme hatası ile karşılaşacaktık. Aşağıdaki Alistırma 1'i inceleyin.

2. Türetilmiş sınıf, temel sınıfından sanal olarak türetildiğinde bu temel sınıf yine de türetilmiş sınıfın içerisinde mevcuttur, bunu unutmayalım. Örneğin, aşağıda verdiğimiz satırların yukarıdaki programda kullanılmasında hiçbir sakınca yoktur:

```
derived1 ob;
ob.i = 100;
```

Normal bir temel sınıf ile sanal bir temel sınıf arasındaki tek bir fark vardır ve bu fark sadece bir nesne aynı temel sınıf kökenli iki türetilmiş sınıfın özelliklerini aldığımda ortaya çıkar. Sanal temel sınıfları kullanırsak nesne içerisinde sadece tek bir temel sınıf bulunur. Aksi halde nesne içerisinde temel sınıfın birden fazla kopyası bulunacaktır.

Aliştırmalar

1. Örnek 1'deki programı kullanın ve **virtual** anahtar kelimesini silin. Sonra programı derlemeyi deneyin ve ne tür hataların ortaya çıktığını dikkat edin.
2. Sanal temel sınıfların kullanılmasının neden gerekli olabileceğini açıklayın.

Pekiştirme Testi

Daha ileri konulara geçmeden önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

1. **building** isminde bir sosyal temel sınıf oluşturun. Bu sınıf, bir binadaki kat sayısını, oda sayısını ve toplam alanını saklasın. **house** isminde bir türetilmiş sınıf oluşturun ve bu sınıf **building** sınıfının özelliklerini mirasla alınsın. Aynı zamanda yatak odası sayısını ve banyo sayısını da saklasın. Daha sonra **office** isminde bir türetilmiş sınıf oluşturun, bu sınıf yine **building** sınıfına ait özellikleri mirasla alınsın ve bu özelliklerin yanı sıra yanın söndürücülerin ve telefonlarının da sayısını saklasın. Not: Sizin cevabınız kitabın sonunda verilen cevaptan farklı olabilir. Fakat, fonksiyonel olarak aynıysa onu doğru sayın.
2. Bir türetilmiş sınıf temel sınıfından public olarak türetilirse temel sınıfın public üyelerine ve private üyelerine ne olur? Türetilmiş sınıf private olarak oluşturulmuşsa yine temel sınıfın public ve private üyelerine ne olur?
3. **protected** ne anlama geliyor açıklayınız. (Kelimenin hem sınıf'a ait üyeler'e gönderme yaparken ki anlamını hem de miras erişim belirticisi olarak kullanırken ki anlamını açıklayın.)
4. Bir sınıfı diğerinden mirasla oluşturduğumuzda sınıflara ait yapıcı ve yıkıcı fonksiyonları ne zaman çağrılır?
5. Size aşağıdaki program iskeleti veriliyor, açıklamalarda gösterildiği şekilde programı tamamlayın :

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

class planet {
protected:
    double distance; // günested mil cinsinden uzaklığı
    int revolve; // gün cinsinden uzaklığı
public:
    planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
    double circumference; // yörüğünün çevresi
public:
    /* earth(double d, int r) fonksiyonunu oluşturun. Bu fonksiyon dönüş mesafesini ve süresini planet'e geri göndersin. Bu fonksiyon da yörüğe çevresini hesap etsin. (İpucu : çevre = 2r*3.1416.) */
}
```

```
*/  
/* show() isimli bir fonksiyon oluşturun, bu fonksiyon sonucu ekranda  
göstersin. */  
}  
  
int main()  
{  
    earth ob(93000000, 365);  
    ob.show();  
    return 0;  
}
```

6. Aşağıdaki programda bulunan hatayı düzeltin:

```
/* Araç hiyerarşisini gösteren programı biraz değiştirdik.  
Fakat bu program bir hata içermektedir. Bu hatayı düzeltin.  
İpucu: programı derlemeyi deneyin ve çıkan hata mesajlarını gözlemlayın.  
*/  
#include <iostream>  
using namespace std;  
  
// Çeşitli araç türleri için oluşturulmuş bir temel sınıf.  
class vehicle {  
    int num_wheels;  
    int range;  
public:  
    vehicle(int w, int r)  
    {  
        num_wheels = w; range = r;  
    }  
    void showv()  
    {  
        cout << "Tekerlekler: " << num_wheels << '\n';  
        cout << "Menzil: " << range << '\n';  
    }  
};  
  
enum motor {gaz, elektrik, dizel};  
  
class motorized : public vehicle {  
    enum motor mtr;  
public:  
    motorized(enum motor m, int w, int r) : vehicle(w, r)  
    {  
        mtr = m;  
    }  
    void showm() {  
        cout << "Motor: ";  
        switch(mtr) {  
            case gas : cout << "Gaz\n";  
                break;  
            case electric : cout << "Elektrik\n";  
                break;  
            case diesel : cout << "Dizel\n";  
                break;  
        }  
    }  
};  
class road_use : public vehicle {  
    int passengers;
```

```
public:  
    road_use(int p, int w, int r) : vehicle(w, r)  
    {  
        passengers = p;  
    }  
    void showr()  
    {  
        cout << "Yolcular: " << passengers << '\n';  
    }  
  
    enum steering { power, rack_pinion, manual };  
  
class car : public motorized, public road_use {  
    enum steering strng;  
public:  
    car(enum steering s, enum motor m, int w, int r, int p) :  
        road_use(p, w, r), motorized(m, w, r), vehicle(w, r)  
    {  
        strng = s;  
    }  
    void show() {  
        showv(); showr(); showm();  
        cout << "Direksiyon : ";  
        switch(strng) {  
            case power : cout << "Power\n";  
            break;  
            case rack_pinion : cout << "Rack ve Pinion\n";  
            break;  
            case manual : cout << "Manual\n";  
            break;  
        }  
    }  
};  
  
int main()  
{  
    car c{power, gas, 4, 500, 5};  
    c.show();  
  
    return 0;  
}
```

Bütünleştirme Testi

Burada bir önceki bölümde bu bölümde öğrendiklerinizi ne kadar iyi bir araya getirebildiğiniz kontrol edilmektedir.

1. Yukarıdaki Pekiştirme Testinin 6. sorusunda **car** ve **motorized**'ın içerisindeki **switch** bildirisini ile ilgili bir hata mesajı görmüş olabilirsiniz. Sizce bunun nedeni nedir?
2. Bir önceki bölümden hatırlayacağınız gibi, temel sınıf içerisinde aşırı yüklenmiş bulunan operatörlerin çoğu türetilmiş sınıfın içerisinde de kullanılabilir. Hangisi ya da hangileri kullanılamaz? Bu duruma bir sebep gösterebilir misiniz?

3. Bir önceki bölümde ele aldığımız coord sınıfı üzerinde biraz değişiklik yaptık. Bu sınıf burada quad isminde bir başka sınıfın temel sınıfı olarak kullanılıyor. Programı çalıştırın ve verdiği sonucu anlamayı deneyin.

```
/* +, -, ve = operatorleri coord sınıfı için aşırı yükleniyor
   ve sonra coord sınıfını quad için temel sınıf olarak kullanılıyor. */
#include <iostream>
using namespace std;

class coord {
public:
    int x, y; // koordinat değerleri
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator+(coord ob2);
    coord operator-(coord ob2);
    coord operator=(coord ob2);
};

// + operatörü coord sınıfı için aşırı yükleniyor.
coord coord::operator+(coord ob2)
{
    coord temp;

    cout << "operator+() kullanılıyor\n";

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}

// - operatörü coord sınıfı için aşırı yükleniyor.
coord coord::operator-(coord ob2)
{
    coord temp;

    cout << "operator-() kullanılıyor\n";

    temp.x = x - ob2.x;
    temp.y = y - ob2.y;

    return temp;
}

// = operatörü coord sınıfı için aşırı yükleniyor.
coord coord::operator=(coord ob2)
{
    cout << "Using coord operator=\n";

    x = ob2.x;
    y = ob2.y;

    return *this; // Kendisine atama yapılan nesne döndürülüyor
}

class quad : public coord {
    int quadrant;
```

```

public:
    quad() { x = 0; y = 0; quadrant = 0; }
    quad(int x, int y) : coord(x, y)
    {
        if(x>=0 && y>=0) quadrant = 1;
        else if(x<0 && y>=0) quadrant = 2;
        else if(x<0 && y<0) quadrant = 3;
        else quadrant = 4;
    }
    void showq()
    {
        cout << "Point in Quadrant: " << quadrant << '\n';
    }
    quad operator=(coord ob2);
};

quad quad::operator=(coord ob2)
{
    cout << "Using quad operator=( )\n";

    x = ob2.x;
    y = ob2.y;
    if(x>=0 && y>=0) quadrant = 1;
    else if(x<0 && y>=0) quadrant = 2;
    else if(x<0 && y<0) quadrant = 3;
    else quadrant = 4;

    return *this;
}

int main()
{
    quad o1(10, 10), o2(15, 3), o3;
    int x, y;

    o3 = o1 + o2; // iki nesne toplanıyor - bu, operator+() fonksiyonunu çağırır
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 - o2; // iki nesne birbirinden çıkarılıyor
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1; // nesne atanıyor
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o3=o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

4. Ahşirma 3'te verilen programı arkadaş operatör fonksiyonlarını kullanacak şekilde değiştirin.

BÖLÜM 8

C++'in I/O Sistemine Giriş

C++'da Bazı I/O Temelleri
Biçimlendirilmiş I/O
width(), precision() ve fill() Kullanımı
I/O Manipülatörlerinin Kullanımı
Kendi Inserter'larınızı Oluşturun
Extractor Yaratmak

Kitabın başından beri C++'ın yeni I/O'sunu kullanıyoruz, şimdi bu konuyu detaylı bir biçimde ele almanın tam zamanı. Tıpkı atası C gibi C++ da esnek, güçlü ve zengin bir I/O sistemine sahiptir. C++, C'nin I/O sistemini de tümüyle desteklemektedir, bunu unutmamak lazım. Fakat, kendine özgü nesnel I/O rutinleri vardır. Yeni I/O sisteminin en büyük avantajı, sizin oluşturduğunuz sınıflar için aşırı yüklenememesidir ve bu sistem oluşturduğunuz yeni tipleri bir araya getirmenize imkan verir.

Tıpkı C'nin I/O sistemi gibi, C++'in nesnel I/O sistemi, konsol ve dosya I/O'su arasında fark gözetmez. Dosya ve konsol I/O'su gerçekten de aynı mekanizmaya farklı iki bakış açısıdır. Bu bölümdeki Aşıtırmalarda konsol I/O'su kullanıyoruz, fakat verdigimiz bilgileri dosya I/O'su için de kullanabilirsiniz. (Dosya I/O'sunu Bölüm 9'da detaylarıyla inceleyeceğiz.)

Bu kitabın hazırlamıştı sırasında iki ayrı I/O kütüphanesi kullanılıyordu: C++'in ilk özelliklerine dayanan eski kütüphane ve Standart C++ tarafından tanımlanmış olan yeni kütüphane. Bu kütüphaneler programcılara büyük ölçüde aynı gibi gözükür. Çünkü yeni I/O kütüphanesi eskisinin güncelleştirilmesi ve geliştirilmesiyle oluşturulmuştur. Aslında, ikisi arasındaki en büyük farklılık kullanım aşamasında ortaya çıkmaz, görünmeyen kısımda kalır. Programcıların bakış açısına göre en büyük fark, yeni I/O kütüphanesinin yeni özellikler içermesi ve yeni veri tipleri tanımlamasıdır. Sonuç olarak yeni I/O kütüphanesi eskisinin daha kapsamlı hale getirilmesiyle oluşturulmuştur. Eski kütüphane için yazılmış olan programların neredeyse tamamı, pek bir değişiklikle gerek duyulmadan yeni kütüphane ile derlenebilecektir. Eski kütüphane artık kullanılmadığından kitabımızda sadece Standart C++ tarafından tanımlanmış olan yeni I/O kütüphanesi ele alacağız. Fakat verdigimiz bilgilerin çoğunu eski I/O kütüphanesi ile de kullanabilirsiniz.

Bu bölümde C++'in I/O sistemini çeşitli açılardan inceleyeceğiz. Biçimlendirilmiş I/O, I/O manipülatörleri, kendi I/O inserter'larınızı ve extractor'larınızı oluşturma gibi konuları ele alacağız. Siz de göreceksiniz ki, C++'in I/O sistemi, C'nin I/O sistemi ile pek çok ortak özelliği sahiptir.

Gözden Geçirme Testi

Daha ileri konulara geçmeden önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

1. Hava taşıtları hakkında bilgi saklayan bir sınıf hiyerarşisi oluşturun. İşe **airship** isminde genel bir temel sınıf oluşturarak başlayın. Bu sınıf taşınabilecek yolcu sayısı ve kargo miktarnı (kilogram cinsinden) saklasın. Daha sonra **airship** temel sınıfından **airplane** ve **balloon** isimlerinde iki türetilmiş sınıf oluşturun. **airplane** sınıfı, kullanılan motor tipini (pervaneli veya jet) ve mil cinsinden menzili saklasın. **balloon** sınıfı ise balonu kaldırınmak için kullanılan gaz tipini (hidrojen veya heliyum) ve maksimum seyir yüksekliğini saklasın (fit cinsinden). Bu sınıf hiyerarşisini gösteren kısa bir program oluşturun. (Şüphesiz ki vereceğiniz cevap kitabın ar-

kasındaki çözümden biraz farklı olacaktır. Eğer işlevsel olarak benziyorsa cevabınızı doğru sayabilirsiniz.)

2. **protected**'ı ne amaçla kullanırsınız?
3. Size aşağıda bir sınıf hiyerarşisi veriliyor. Constructor ve destructor fonksiyonları hangi sırayla çağrılır?

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

class A {
public:
    A() { cout << "A oluşturuluyor \n"; }
    ~A() { cout << "A yok ediliyor \n"; }
};

class B : public A {
public:
    B() { cout << "B oluşturuluyor\n"; }
    ~B() { cout << "B yok ediliyor\n"; }
};

class C : public B {
public:
    C() { cout << "C oluşturuluyor\n"; }
    ~C() { cout << "C yok ediliyor\n"; }
};

int main()
{
    C ob;
    return 0;
}
```

4. Size aşağıdaki program veriliyor, constructor fonksiyonları hangi sırayla çağrılır?

```
class myclass : public A, public B, public C { ... }
```

5. Aşağıdaki programda constructor fonksiyonlarını tamamlayın:

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

class base {
    int i, j;
public:
    // need constructor
    void showij() { cout << i << ' ' << j << '\n'; }
};

class derived : public base {
    int k;
public:
    // need constructor
    void show() { cout << k << ' '; showij(); }
};

int main()
```

```

    derived ob{1, 2, 3};
    ob.show();
    return 0;
}

```

6. Genelde, bir sınıf hiyerarşisi tanımlarken, işe en _____ sınıf ile başlarsınız ve en _____ sınıf'a doğru hareket edersiniz. (Boşlukları doldurun.)

8.1. C++'da Bazı I/O Temelleri

C++'in I/O'sunu inceleyeceğiz, fakat daha önce bazı açıklamalar yapmamız gerekecek. C++'ın I/O sistemi, tipki C'nin I/O sistemi gibi, *akışlar (stream)* üzerinde hareket eder. C'deki tecrübelerinizle zaten akımın ne olduğunu biliyor olmalısınız, fakat yine de özet şeklinde açıklayalım. Akım, bilgi üretten veya bilgiyi alan sanal bir aygittır. Fiziksel aygıtlara C++'ın I/O sistemi ile bağlanır. Bağlı oldukları fiziksel aygıtlar farklı da olsa bütün akımlar aynı şekilde davranışırlar. Bütün akımların aynı şekilde çalışması nedeniyle, I/O sistemi, farklı yeteneklere sahip aygıtlar üzerinde çalıştığı halde programcı aynı arabirimini kullanır. Örneğin ekrana bilgi göndermek için kullandığınız fonksiyonu aynı zamanda disk ve yazıcı erişimi için de kullanabilirsiniz.

Bildiğiniz gibi, bir C programı çalışmaya başladığında önceden tanımlanmış üç akım kendiliğinden açılır: Bu akımlar **stdin**, **stdout** ve **stderr**. Aynı şey C++ programı için de geçerlidir. C++ programı çalışmaya başladığında şu dört akım kendiliğinden açılır:

Akım	Anlamı	Varsayılan Aygit
cin	Standart giriş	Klavye
cout	Standart çıkış	Ekrان
cerr	Standart hata	Ekrان
clog	Ön-bellekli cerr	Ekrان

Sizin de tahmin ettiğiniz gibi C++'daki **cin**, **cout** ve **cerr** akımları C'nin **stdin**, **stdout** ve **stderr** akımlarına karşılık gelir. **cin** ve **cout**'u zaten kullanıyorduk. **clog** akımı ise **cerr**'in ön-belleklisidir. Standart C++, bu akımların **wcin**, **wcout**, **wcerr**, ve **wclog** adı verilen 16 bitlik sürümlerini de kullanır, fakat biz onları kitabımızda kullanmayacağız. 16 bitlik bu geniş karakter akımları, Çince gibi büyük karakter kümeleri kullanan dilleri desteklemek için oluşturulmuştur.

Normalde, standart akımlar konsol ile haberleşmek için kullanılmıştır. Fakat, I/O yönlendirme özelliğini destekleyen ortamlarda bu akımlar başka aygıtlara yönlendirilebilir.

Bölüm 1'de öğrenmişistik, C++, kendi I/O sistemine **<iostream>** başlık dosyası ile destek verir. Bu dosyada, I/O işlemlerini destekleyen çok daha karmaşık sınıf hiyerarşileri tanımlanmıştır. I/O sınıfları *şablon sınıflar* sistemi ile başlar. Sosyal sınıflar adını da verdiği şablon sınıflar konusuna Bölüm 11'de tekrar döneceğiz, şimdilik tanımını vermekle yetinelim. Şablon sınıf, işleyeceği verileri tam olarak belirlemeden sınıfın biçimini tanımlayan sınıfıdır. Şablon sınıfı tanımladığımızda ona ait belirli örnekler oluşturabiliriz. I/O kü-

tüphanesi ile ilişkili olan Standart C++, iki farklı I/O şablon sınıfı oluşturur, bunlardan biri 8 bitlik karakterler diğer ise geniş karakterler içindir. Bunlardan en çok kullanılan 8 bitlik olandır, bu nedenle kitap içerisinde sadece onlara yer vereceğiz.

C++'ın I/O sistemi, farklı fakat birbiriyle bağlantılı iki şablon sınıfı hiyerarşisi üzerine kuruldu. Bunlardan ilki, **basic_streambuf** ismindeki alt seviyeli I/O sınıfı tarafından türetilmiştir. Bu sınıf, temel alt seviyeli giriş-çıkış işlemlerini destekleyen ve tüm I/O sisteminin temelini oluşturan sınıfıdır. Gelişmiş I/O işlemleri kullanmıyorsak **basic_streambuf**'e doğrudan ihtiyacımız olmayacak. En sık kullanacağımız sınıf hiyerarşisi **basic_ios**'dır. Bu üst seviyeli bir I/O sınıfıdır, biçimlendirme, hata denetimi ve I/O akımı durum bilgisi sağlar. **basic_ios**, **basic_istream**, **basic_ostream** ve **basic_iostream**'i de içeren çeşitli türetilmiş sınıflar için bir temel olarak kullanılmaktadır. Bu sınıflar ise giriş, çıkış ve giriş/çıkış yapabilen akımlar oluşturmaktadır.

Daha önce açıklamıştık, I/O kütüphanesi iki farklı sınıf hiyerarşisi oluşturur. Bunlardan biri 8 bitlik karakterler için ve diğerinin geniş karakterler içindir. Aşağıdaki tabloda 8 bitlik karakterler için olanın şablon sınıflarını inceleyelim (bunlardan bazılarını Bölüm 9'da kullanacağız):

Şablon Sınıfı	8 Bitlik Karakter Tabanlı Sınıf
basic_streambuf	streambuf
basic_ios	ios
basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_fstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream

Programlarınızda kullanacağınız karakter tabanlı isimleri biz de kitap içerisinde kullanacağız. Aynı isimler eski I/O kütüphanesi tarafından da kullanılan isimlerdir. Eski ve yeni I/O kütüphanelerinin kaynak kodu seviyesinde uyumlu olmasının nedeni de budur.

Son bir noktaya daha değinelim, **ios** sınıfı, önemli akım işlemleri denetleyen ve izleyen çeşitli üye fonksiyonlar, değişkenler içerir. Oldukça da sık kullanılır. Sadece şunu hatırlamamız gerekiyor, programımız **<iostream>**'i içeriyorsa, bu önemli sınıfa erişim hakkımız olacaktır.

8.2. Biçimlendirilmiş I/O

Şimdiye kadar verdığımız tüm alıştırmalarda bilgileri ekran'a göndermek için C++'ın varsayılan formatlarını kullandık. Fakat, bilgiyi çok çeşitli şekillerde göndermemiz mümkündür. Gerçekte, C++'da tipki C'nin **printf()** fonksiyonunu kullandığımız şekilde verileri biçimlendirerek ekran'a gönderebiliriz. Bunun yanında veri girişinin nasıl yapılacağına da karar verebilirisiniz.

Her akım, bilginin biçimlendirilişini kontrol eden bir takım format bayrakları ile bağlantılıdır. **ios** sınıfı, içerisinde aşağıdaki değerlerin tanımlandığı **fmtflags** isminde bir bitmask deklare edilir:

adjustfield	floatfield	right	skipws
basefield	hex	scientific	unitbuf
boolalpha	internal	showbase	uppercase
dec	left	showpoint	
fixed	oct	showpos	

ios'un içerisinde tanımlanmış olan bu değerlerle format bayraklarına değer verilir ya da bu bayrakların değerleri silinir. Eğer daha eski ve Standart olmayan bir derleyici kullanıyorsanız derleyiciniz **fmtflags** enumeration (numaralandırılmış sabit) tipini tanımayabilir. Bu durumda format bayrakları bir long integer'in içine kodlanacaktır.

skipws bayrağına değer verildiğinde, başlıca boşluk karakterleri (boşluklar, sekmeler ve satırbaşları) akım içerisinde giriş yapılrken atılır. **skipws**'in içindeki değer silindiğinde ise boşluk karakterleri atulmaz.

left bayrağına değer verildiğinde, çıkış soldan hizalanır. **sağ** bayrağına değer verildiğinde ise sağdan hizalanır. **internal** bayrağı ise, nümerik bir değerin herhangi bir işaret ya da taban değerinin arasına boşluk ekleyerek bir alanı doldurmasını sağlar. Eğer bu bayraklardan hiçbirine değer vermezsek çıkış varsayılan olarak sağa hizalı verilecektir.

Normalde nümerik değerler çıkışa onluk düzende gönderilir. Fakat, sayı tabanını değiştirmemiz mümkün değildir. **oct** bayrağı ile çıkışımız sekizlik düzende, **hex** bayrağı ile de onaltılık düzende gösterilir. Sonra çıkışı tekrar onluk düzende vermek istersek **dec** bayrağını 1 yapmalıyız.

showbase bayrağı ile nümerik değerlerimizin tabanını da gösterebiliriz. Örneğin değiştirmeye tabanımız onaltı ise 1F değeri 0x1F şeklinde gösterilir.

Normalde bilimsel notasyon kullanıldığında e küçük harfle yazılır. Onaltılık düzende bir değer gösterilirken ise x küçük harfle yazılır. **uppercase** bayrağı 1 yapıldığında bu karakterler büyük harfle verilir.

showpos bayrağı ile pozitif değerlerin başına artı işaretini konulur.

showpoint bayrağı ise gereksin veya gerekmese tüm kayan nokta çıkışlarında nokta ve ondan sonra gelen sıfırların gösterilmesine neden olur.

scientific bayrağımızı 1 yaparsak kayan nokta nümerik değerlerimiz bilimsel notasyonla gösterilir. **fixed** bayrağı ile de kayan nokta değerlerimiz normal notasyonla verilir. Hangi bayrak 1 yapılmış olursa olsun derleyici doğru yönemi bulacaktır.

unitbuf bayrağı ile tampon belleğimiz her ekleme işleminin ardından taşar.

boolalpha bayrağına değer verildiğinde Boolean'lar **true** ve **false** anahtar kelimeleri kullanılarak giriş veya çıkış olabilir.

oct, **dec** ve **hex** alanları çok sık kullanılır, bu nedenle onlar **basefield** (temel alan) olarak da adlandırılır. Benzer şekilde **left**, **right** ve **internal** alanları da **adjustfield** şeklinde kullanılabilir. Son olarak da **scientific** ve **fixed** alanlarına **floatfield** şeklinde başvurulabilir.

Format bayrağına değer vermek için **setf()** fonksiyonunu kullanırız. Bu fonksiyon **ios**'a ait bir üyedir. En sık karşılaşacağınız kullanım şekli şudur :

```
fmtflags setf(fmtflags flags);
```

Bu fonksiyon format bayraklarının bir önceki değerini döndürür ve *flags* tarafından belirlenen bayrakların değerlerini on (açık) yapar. (Diğer bayrakların hiçbirini bundan etkilenmez.) Örneğin, **showpos** bayrağının değerini on (açık) yapmak için aşağıdaki deyīimi kullanabiliriz :

```
stream.setf(ios::showpos);
```

Burada *stream*, etkilemek istediğiniz akımdır. Kapsam çözümleme operatörünün nasıl kullanıldığına dikkat edelim. **showpos**, **ios** sınıfı içerisindeki numaralandırılmış (enumerated) bir sabittir. Sonuç olarak derleyiciye bu durumu **showpos**'un önüne sınıf ismini ve kapsam çözümleme operatörünü yazarak bildirmek gereklidir. Eğer bunu yapmazsak **showpos** sabiti görmezden gelinecektir.

setf(), **ios** sınıfına ait bir üye fonksiyondur ve bu sınıf tarafından oluşturulan akımları etkiler, bunu anlamamız önemli. Sonuçta **setf()** fonksiyonu, mutlaka belirli bir akıma ilişkin olarak çağrılr. **setf()**'i kendi kendine çağrımanın bir anlamı yoktur. Olaya başka bir açıdan bakacak olursak C++'da global format durumu kavramı da yoktur. Her akım, kendi özel format durum bilgisini korur.

setf() fonksiyonunun tek bir çağrılsında birden fazla bayrağın değerini değiştirmemiz mümkünür, bu fonksiyonu birkaç kere çağrırmamıza gerek yoktur. Örneğin aşağıdaki çağrı ile **showbase** ve **hex** bayraklarının her ikisinin de değerini **cout** için değiştirebiliriz :

```
cout.setf(ios::showbase | ios::hex);
```

HATIRLATMA Format bayraklarının **ios** sınıfı içinde tanımlanmış olması nedeniyle her iki değere de **ios** ve kapsam çözünürlük operatörünü kullanarak erişmeliyiz. Örneğin, **showbase** tek başına fork edilmeyecektir, **ios::showbase** şeklinde yazmamız gerektir.

setf() fonksiyonunun türmeyeni **unsetf()** fonksiyonudur. **ios**'a ait bu üye fonksiyon bir veya birden fazla format bayrağının değerini sıfırlar. En sık kullanılan prototip şu şekildedir :

```
void unsetf(fmtflags flags);
```

flags tarafından belirlenen bayrakların değeri silinir. (Diğer bayrakların hiçbirisi bundan etkilenmez.)

Bazı zamanlar o anki format değerlerini bilmemiz gerekecek, ama aynı zamanda bu değerleri korumak da isteyeceğiz. Bir veya daha fazla bayrağın değerini değiştiren **setf()** ve **unsetf()** fonksiyonlarını biliyoruz, fakat **ios**'a ait **flags()** isminde bir başka üye fonksiyon daha vardır ve bu fonksiyon, format bayraklarının o anki değerlerini döndürür. Prototipi ise şu şekildedir:

```
fmtflags flags();
```

flags() fonksiyonunun ikinci bir şekli daha vardır, fonksiyonun bu kullanımıyla, bir akımla ilişkili olan *tüm* format bayraklarının değerini değiştirebiliriz. Bu format bayrakları fonksiyonun argümanında belirlenir. **flags()** fonksiyonunun bu ikinci prototipi ise şöyledir:

```
fmtflags flags(fmtflags f);
```

Fonksiyonu bu şekilde kullandığımızda, *f*'in içerisindeki bit düzeni, akımla ilişkili format bayraklarını saklamakta kullanılan değişkene kopyalanır. Bu şekilde bayrakların önceki değerleri değiştirilmiş olur. Fonksiyon ise eski değerleri döndürür.

Alıştırmalar

1. Aşağıdaki örnekle çeşitli format bayraklarının değerlerini nasıl değiştirebileceğimizi görelim:

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

int main()
{
    // Çıkış varsayılan değerlerle gösterilecek.
    cout << 123.23 << " merhaba " << 100 << '\n';
    cout << 10 << ' ' << -10 << '\n';
    cout << 100.0 << "\n\n";

    // Şimdi formatlar değiştiriliyor
    cout.unsetf(ios::dec); // tüm derleyicilerde gerekmek.
    cout.setf(ios::hex | ios::scientific);
    cout << 123.23 << " merhaba " << 100 << '\n';

    cout.setf(ios::showpos);
    cout << 10 << ' ' << -10 << '\n';

    cout.setf(ios::showpoint | ios::fixed);
    cout << 100.0;

    return 0;
}
```

Program bize aşağıdaki çıkışı verecektir:

```
123.23 merhaba 100
10 -10
```

```

100
1.232300e+02 merhaba 64
a ffffff6
+100.000000

```

showpos bayrağı sadece onluk düzendeki çıkışları etkiler, buna dikkat edelim. Çıkış onaltılık düzendeyken 10 değerini etkilemez. Aynı zamanda **dec** bayrağını (varsayılan değeri on - açık) değiştirmek için **unsetf()** fonksiyonuna yapılan çağrıya da dikkat etmemiz gerekiyor. Her derleyicide bu çağrıya gerek bulunmayabilir. Fakat bazı derleyiciler **dec** bayrağı diğer bayrakların değerini değiştirebilir, bu nedenle **hex** veya **oct**'un değerini on yaparken bu bayrağı off (kapalı) hale getirmek gereklidir. Genelde, maksimum taşınabilirlik için sadece kullanmak istediğiniz sayı tabanını değiştirmek ve diğerlerini silmek en iyisidir.

2. Bu programda **uppercase** bayrağının gösterdiği etkiyi göreceğiz. İlk önce **uppercase**, **showbase** ve **hex** bayraklarına değer vereceğiz. Sonra onaltılık düzende 88 sonucunu ekranda göstereceğiz. Burada onaltılık düzende kullanılan X büyük harfle yazılacak. Sonra **unsetf()** fonksiyonu ile **uppercase** bayrağını sıfırlayacağız ve 88 sayısını onaltılık düzende tekrar ekrana göndereceğiz. Bu sefer x küçük harfle yazılacak.

```

#include <iostream>
using namespace std;

int main()
{
    cout.unsetf(ios::dec);
    cout.setf(ios::uppercase | ios::showbase | ios::hex);

    cout << 88 << '\n';

    cout.unsetf(ios::uppercase);

    cout << 88 << '\n';

    return 0;
}

```

3. Program, **flags()** fonksiyonunu kullanarak **cout**'la ilişkili olan format bayraklarının değerlerini ekranda gösterir. **showflags()** fonksiyonuna özellikle dikkat edelim. Bu fonksiyonu ileride yazacağınız programlarda kullanabilirsiniz.

```

#include <iostream>
using namespace std;

void showflags();

int main()
{
    // format bayraklarının varsayılan durumu gösteriliyor
    showflags();
}

```

```
cout.setf(ios::oct | ios::showbase | ios::fixed);
showflags();
return 0;
}

// Bu fonksiyon format bayraklarının durumunu gösteriyor.
void showflags()
{
    ios::fmtflags f;

    f = cout.flags(); // flag değerleri alınıyor

    if(f & ios::skipws) cout << "skipws on\n";
    else cout << "skipws off\n";

    if(f & ios::left) cout << "left on\n";
    else cout << "left off\n";

    if(f & ios::right) cout << "right on\n";
    else cout << "right off\n";

    if(f & ios::internal) cout << "internal on\n";
    else cout << "internal off\n";

    if(f & ios::dec) cout << "dec on\n";
    else cout << "dec off\n";

    if(f & ios::oct) cout << "oct on\n";
    else cout << "oct off\n";

    if(f & ios::hex) cout << "hex on\n";
    else cout << "hex off\n";

    if(f & ios::showbase) cout << "showbase on\n";
    else cout << "showbase off\n";

    if(f & ios::showpoint) cout << "showpoint on\n";
    else cout << "showpoint off\n";

    if(f & ios::showpos) cout << "showpos on\n";
    else cout << "showpos off\n";

    if(f & ios::uppercase) cout << "uppercase on\n";
    else cout << "uppercase off\n";

    if(f & ios::scientific) cout << "scientific on\n";
    else cout << "scientific off\n";

    if(f & ios::fixed) cout << "fixed on\n";
    else cout << "fixed off\n";

    if(f & ios::unitbuf) cout << "unitbuf on\n";
    else cout << "unitbuf off\n";

    if(f & ios::boolalpha) cout << "boolalpha on\n";
    else cout << "boolalpha off\n";

    cout << "\n";
}
```

`showflags()` fonksiyonunda, `f` yerel değişkeni `fmtflags` tipinde deklare ediliyor. Eğer derleyiciniz `fmtflags` tipini tanımiyorsa bu değişkeni `long` olarak deklare edebilirsiniz. Programımız ekranda şu sonucu verecektir :

```
skipws on
left off
right off
internal off
dec on
oct off
hex off
showbase off
showpoint off
showpos off
uppercase off
scientific off
fixed off
unitbuf off
boolalpha off
```

```
skipws on
left off
right off
internal off
dec on
oct on
hex off
showbase on
showpoint off
showpos off
uppercase off
scientific off
fixed on
unitbuf off
boolalpha off
```

4. Burada `flags()` fonksiyonunun ikinci kullanım şeklini göreceğiz. Bu program, ilk önce `showpos`, `showbase`, `oct` ve `right`'ın değerini on yapan bir bayrak maskesi oluşturur. Sonra `flags()` fonksiyonunu kullanarak `cout` ile ilişkili bayrak değişkenlerini bu değerlere getirir. `showflags()` fonksiyonu bayraklara değerlerinin doğru bir şekilde verildiğini doğrular. (Bir önceki programda kullanılan fonksiyon.)

```
#include <iostream>
using namespace std;
void showflags();
int main()
{
    // format bayraklarının varsayılan değerleri gösterilir
    showflags();
    // showpos, showbase, oct, left bayrakları on değerinde, diğerleri ise off
    ios::fmtflags f = ios::showpos | ios::showbase | ios::oct | ios::left;
    cout.flags(f); // bayraklara değerleri veriliyor
    showflags();
    return 0;
}
```

Aliştırmalar

1. cout'a ait bayrakların değerlerini pozitif tamsayıların önüne artı işaretini konacak şekilde değiştiren bir program yazın. Format bayraklarına doğru değerler verdığınızı gösterin.
2. cout'a ait bayrakların değerlerini kayan nokta değerleri gösterilirken her zaman noktayı da gösteren bir program yazın. Bunun yanında kayan nokta değerleri, bilimsel notasyonla gösterilsin ve büyük E harfi kullanılsın.
3. Format bayraklarının o anki durumunu saklayan, **showbase** ve **hex**'in değerlerini değiştiren ve 100 değerini gösteren bir program yazın.

8.3. width(), precision() ve fill()'in Kullanılması

Bayrakların biçimlendirilmesinde kullanılan fonksiyonlara ek olarak **ios** tarafından tanımlanan üç üye fonksiyon daha vardır. Bu fonksiyonlar şu format parametrelerin değerlerini değiştirirler: alan genişliği (field width), doğruluk (toplam basamak sayısı, precision) ve doldurma karakteri (fill character). Bu fonksiyonlar sırasıyla **width()**, **precision()** ve **fill()** fonksiyonlarıdır. Normalde çıkışa bir değer gönderdiğimizde bu değer, sadece karakter sayısı kadar yer kaplar. Fakat, **width()** fonksiyonunu kullanarak minimum bir alan belirleyebiliriz. Bunu aşağıdaki şekilde yapabiliriz:

```
streamszie width(streamsize w);
```

Burada *w* alan genişliğidir ve fonksiyon bir önceki alan genişliğini döndürür. **streamszie** tipi **<iostream>** tarafından bir tamsayı olarak tanımlanmaktadır. Bazı durumlarda çıkış işlemini her gerçekleştirdiğimizde alan genişliği varsayılan değerine gelebilir. Bu yüzden çıkış deyimlerinden önce minimum alan genişliği değerini belirlemek gerekli olabilir. Biz bu minimum alan genişliğini belirledikten sonra eğer değer belirlediğimizden daha az yer kullanıyorsa, boş kalan alan o anki doldurma karakteri (varsayılan değeri boşluk karakteridir) ile doldurulur. Fakat, şunu aklımızda tutalım, eğer çıkış değerinin büyütülüğü minimum alan genişliğinden daha büyükse alan tamamen dolacaktır. Değerde yuvarlatma yapılmaz.

Normalde basamak sayısı altıdır. Bu sayıyı **precision()** fonksiyonu ile değiştirebiliriz. Bu fonksiyon şu şekilde kullanılır:

```
streamszie precision(streamsize p);
```

Burada toplam basamak sayısına (precision) *p* değerini veriyoruz ve fonksiyon da eski değeri döndürür.

Normalde, alanın doldurulması gerekiyorsa, boş yerler boşluk karakteri ile doldurulur. Doldurma karakterini **fill()** fonksiyonunu kullanarak değiştirebiliriz. Bu fonksiyonun kullanımı şu şekildedir:

```
char fill(char ch);
```

`fill()`'in çağrılmamasından sonra, `ch` yeni doldurma karakteri olur ve eski değer döndürülür.

Alıştırmalar

1. Bu programda format fonksiyonlarını inceleyeceğiz:

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

int main()
{
    cout.width(10); // minimum alan genişliğine değer veriliyor
    cout << "merhaba" << '\n'; // varsayılan olarak sağa hizalı
    cout.fill('*'); // doldurma karakteri belirleniyor
    cout.width(10); // genişlik belirleniyor
    cout << "merhaba" << '\n'; // varsayılan olarak sağa hizalı
    cout.setf(ios::left); // sola hizalı
    cout.width(10); // genişlik belirleniyor
    cout << "merhaba" << '\n'; // çıkış sola hizalı

    cout.width(10); // genişlik belirleniyor
    cout.precision(10); // toplam basamak sayısı 10 yapılıyor
    cout << 123.234567 << '\n';
    cout.width(10); // genişlik belirleniyor
    cout.precision(6); // toplam basamak sayısı 6 yapılıyor
    cout << 123.234567 << '\n';

    return 0;
}
```

Program şu çıkışı verecek:

```
merhaba
*****merhaba
hello*****
123.234567
123.235***
```

Her çıkış deyiminden önce alan genişliğini yeniden belirlediğimize dikkat edelim.

2. Aşağıdaki programda hizalı bir numara tablosu oluşturmak için C++'ın I/O format fonksiyonlarını nasıl kullanacağımızı göreceğiz:

```
// Sayıların karekök ve karelerini veren bir tablo oluşturulacak.
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double x;
    cout.precision(4);
    cout << " x sqrt(x) x^2\n\n";
    for(x = 2.0; x <= 20.0; x++) {
        cout.width(7);
        cout << x << " ";
```

```

    cout.width(7);
    cout << sqrt(x) << " ";
    cout.width(7);
    cout << x*x << '\n';
}
return 0;
}

```

Program bize aşağıdaki tabloyu verecektir :

x	sqrt(x)	x^2
2	1.414	4
3	1.732	9
4	2	16
5	2.236	25
6	2.449	36
7	2.646	49
8	2.828	64
9	3	81
10	3.162	100
11	3.317	121
12	3.464	144
13	3.606	169
14	3.742	196
15	3.873	225
16	4	256
17	4.123	289
18	4.243	324
19	4.359	361
20	4.472	400

Alıştırmalar

1. 2 ile 100 arasındaki sayıların doğal logaritmasını ve 10 tabanındaki logaritmasını yazdırın bir program oluşturun. Tablodaki sayılar, 10 genişliğinde bir alan içerisinde sağa hizalı olacak ve onluk tabandaki bu sayıların toplam basamak sayısı da beş olacak şekilde tabloyu formatlayın.
2. `center()` isminde bir fonksiyon oluşturun, bu fonksiyon şu şekilde kullanılın:

```
void center(char *s);
```

Bu fonksiyon belirlenen katarı ekranın tam ortasına yazın. Bu fonksiyonu oluşturmak için `width()` fonksiyonunu kullanın. Ekranın 80 karakter genişliğinde olduğunu farz edin. (Problemi daha basit hale getirmek için hiçbir katarın 80 karakteri geçmediğini varsayıabilirsiniz.) Fonksiyonunuz çalıştığını da bir program yazarak gösterin.

3. Format bayrakları ve format fonksiyonları ile çeşitli denemeler yapın. C++'ın I/O sistemini uygulamalar yaparak daha yakından tanıdıkça çıkışları istediğiniz gibi formatlamakta hiç zorluk çekmeyeceksiniz.

8.4. I/O Manipülatörlerinin Kullanılması

C++'da bilgilerin formatlanması için bir başka yöntem daha kullanılabilir. Bu metotta *I/O manipülatörleri* adı verilen özel fonksiyonlar kullanılır. Siz de göreceksiniz, I/O manipülatörlerinin kullanılması bazı durumlarda **ios** format bayraklarını ve fonksiyonlarını kullanmaktan daha kolaydır.

I/O manipülatörleri, bir I/O deyimi içerisinde bulunabilen özel I/O format fonksiyonlarıdır. **ios** üye fonksiyonları gibi I/O deyimlerinden uzak olmak zorunda değildir. Tablo 8.1'de standart manipülatörleri görebiliriz. Sizin de görebileceğiniz gibi, I/O manipülatörlerinin çoğu **ios** sınıfına ait üye fonksiyonlara paraleldir. Tablo 8.1'de gösterilen manipülatörlerin çoğu Standart C++'a kısa bir süre önce eklenmişlerdir ve sadece modern derleyiciler tarafından desteklenecektir.

setw() gibi parametre alan manipülatörlerle erişmek için, programımıza **<iomanip>**'ı dahil etmemiz gereklidir. Eğer kullandığınız manipülatörün argümanına ihtiyaç yoksa bunu yapmamız gerekmeyebilir.

Yukarıda da değinmiş olduğumuz manipülatörler I/O işlemlerinin içerisinde bulunabilirler. Örneğin:

```
cout << oct << hex << 100;
cout << setw(10) << 100;
```

İlk deyim **cout**'a tamsayıları sekizlik düzende göstermesini söyler ve çıkışa 100 gönderir. Sonra akıma tamsayıları onaltılık düzende göstermesini söyler ve yine çıkışa onaltılık formatta 100 gönderir. İkinci deyimde ise alan genişliği 10 yapılır ve sonra çıkışa yine onaltılık formatta 100 gönderilir. Şuna dikkat edelim, eğer **oct**'ta olduğu gibi manipülatörün argümanı yoksa parantez kullanılmaz. Çünkü o, aşırı yüklenmiş **<<** operatöründe gönderdiğimiz manipülatörün adresidir.

I/O manipülatörlerinin sadece I/O ifadesinin bir parçası olan akımı etkilediğini aklimızda bulunduralım. I/O manipülatörleri o an kullanılmak üzere açılan akımların hepsini *etkilemez*. Yukarıdaki örneklerde gördüğümüz gibi, manipülatörlerin **ios** üye fonksiyonlarına göre avantajı, daha kolay kullanımları ve daha yoğun kod yazmamıza imkan vermelidir.

Eğer manipülatör kullanarak belirli format bayraklarını elle değiştirmek istiyorsak **setiosflags()** fonksiyonunu kullanacağız. Bu manipülatör **setf()** fonksiyonu ile aynı şekilde çalışır. Bayrakları off yapmak için **resetiosflags()** manipülatörünü kullanacağız. Bu manipülatör ise **unsetf()** fonksiyonuna karşılık gelir.

Manipülatör	Input/Output
boolalpha	Input/Output
dec	Input/Output
endl	Output
ends	Output
fixed	Output

flush	Output
hex	Input/Output
internal	Output
left	Output
noboolalpha	Input/Output
noshowbase	Output
noshowpoint	Output
noshowpos	Output
noskipws	Input
nounitbuf	Output
nouppercase	Output
oct	Input/Output
resetiosflags	Input/Output
right	Output
scientific	Output
setbase	Input/Output
setfill	Output
setiosflags	Input/Output
setprecision	Output
setw	Output
showbase	Output
showpoint	Output
showpos	Output
skipws	Input
unitbuf	Output
uppercase	Output
ws	Input

Tablo 8.1. Standart C++ Manipülatörleri

Aliştırmalar

1. Bu programda çeşitli I/O manipülatörlerini göreceğiz:

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

int main()
{
    cout << hex << 100 << endl;
    cout << oct << 10 << endl;
    cout << setfill('X') << setw(10);
    cout << 100 << " selam " << endl;
    return 0;
}
```

Bu program bize şu sonucu verecek:

```

64
12
XXXXXXX144 selam

```

2. 2 ile 20 arasındaki sayıların kareleri ve karekökleri tablosunu gösteren programı biraz değiştirdik. Üye fonksiyonlar ve format bayrakları yerine I/O manipülatörleri kullandık.

```

/* Bu programda kare ve karekök tablosunu göstermek için
   I/O manipülörlerini kullanıyoruz. */
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
    double x;
    cout << setprecision(4);
    cout << " x sqrt(x) x^2\n\n";
    for(x = 2.0; x <= 20.0; x++) {
        cout << setw(7) << x << " ";
        cout << setw(7) << sqrt(x) << " ";
        cout << setw(7) << x*x << '\n';
    }
    return 0;
}

```

3. Yeni I/O kütüphanesi tarafından eklenen bayraklarının en ilginci **boolalpha** bayrağıdır. Bu bayrağın değeri doğrudan ya da yeni **boolalpha** veya **noboolalpha** manipülatörleri kullanılarak değiştirilebilir. **boolalpha**'yı bu kadar ilginç yapan, onun değerini değiştirerek Boolean değerlerin **true** ve **false** anahtar kelimeleriyle giriş ve çıkışlarda kullanılmasını sağlayabilmemizdir. Normalde **true** (doğru) için 1, **false** (yanlış) için ise 0 girmeniz gereklidir. Konuyu daha iyi anlamak için aşağıdaki programı inceleyelim:

```

// boolalpha format bayrağı gösterilecek.
#include <iostream>
using namespace std;

int main()
{
    bool b;

    cout << " boolalpha bayrağına değer verilmeden önce : ";
    b = true;
    cout << b << " ";
    b = false;
    cout << b << endl;

    cout << " boolalpha bayrağına değer verildikten sonra : ";
    b = true;
    cout << boolalpha << b << " ";
    b = false;
    cout << b << endl;
}

```

```

cout << "Bir boolean değer girin : ";
cin >> boolalpha >> b; // true veya false değerlerini yazabilirsiniz
cout << b << "değerini girdiniz ";

return 0;
}

```

Çalıştırduğumızda şu sonucu göreceğiz:

```

boolalpha bayrağına değer verilmeden önce: 1 0
boolalpha bayrağına değer verildikten sonra: true false
Bir boolean değer girin: true
true değerini girdiniz

```

boolalpha kullanıldığında boolean değerlerin giriş ve çıkışları artık **true** veya **false** kelimeleri ile yapılabilir. **boolalpha** bayraklarına **cin** ve **cout** için ayrı ayrı değer vermemiz gerektiğine dikkat edelim. Tüm format bayraklarında olduğu gibi tek bir akım (stream) için **boolalpha** bayrağına değer vermek bir diğer akımı etkilemez.

Aliştırmalar

1. Konu 8.3'deki Alistırma 1 ve 2'yi tekrar yapın ve bu sefer üye fonksiyonlar ve format bayrakları yerine I/O manipülatörlerini kullanın.
2. 100 değerini onaltılık düzende ve tabanıyla birlikte (0x) gösteren I/O deyimini yazın. Bunun için **setiosflags()** manipülatörünü kullanın.
3. **boolalpha** bayrağının neyi etkilediğini açıklayın.

8.5. Kendi Inserter'larınızı Oluşturun

Daha önce de bahsetmiştık, C++'ın I/O sisteminin avantajlarında biri, oluşturduğumuz sınıflar için I/O operatörlerini aşırı yükleyebilmemizdi. Bu sayede kendi sınıflarımızı C++ programlarımıza mükemmel bir şekilde entegre edebiliyorduk. Burada C++'ın çıkış operatörü olan **<<** operatörünü nasıl aşırı yükleyeceğimizi öğreneceğiz.

C++'da çıkış işlemine *insertion* (ekleme) ve **<<** operatöründe *insertion operatorü* (ekleme operatörü) adı verilir. **<<** operatörünü çıkış için aşırı yüklediğimizde bir *inserter* fonksiyon (ekleme fonksiyonu) veya kısaca *inserter* (ekleyici) oluşturuyoruz demektir. Bu terimin mantığı çıkış operatörünün akıma (stream) bilgi *eklemesine* dayanmaktadır.

Ekleme fonksiyonlarının tümü şu şekilde kullanılır :

```

ostream &operator <<(ostream &stream, sınıf-ismi ob)
{
    // Ekleme fonksiyonunun gövdesi
    return stream;
}

```

İlk parametremiz **ostream** tipinde bir nesneyi işaret etmektedir. Bu da *stream* (akım) bir çıkış akımı olmalıdır anlamına geliyor. (**ostream**'in **ios** sınıfından türetildiğini hatırlatalım.)

İkinci parametremiz yerine çıkış olarak verilecek nesneyi koyacağınız. (Eğer uygulamamız için daha uygunsa bu bir referans parametresi de olabilir.) Ekleme fonksiyonumuzun *stream*'ı işaret eden **ostream** tipinde bir referans döndürdüğüne dikkat edelim. Aşırı yüklenmiş olan << operatörü aşağıdaki gibi bir takım I/O ifadelerinde kullanılacaksa bu gereklidir:

```
cout << ob1 << ob2 << ob3;
```

Bir ekleme fonksiyonu içerisinde herhangi bir prosedür gerçekleştirilebilirsiniz. Ekleme fonksiyonunun ne yapacağı tamamen size kalmış bir şey. Fakat, uygulamalarınızda ekleme fonksiyonunun tutarlı bir şekilde çalışmasını istiyorsanız, yapacağı işlemleri akıma bilgi göndermekle kısıtlamak zorundasınız.

Bu size başta garip gelebilir, fakat ekleme fonksiyonu çalışacağı sınıf'a ait bir üye *olamaz*. Bunun nedeni şudur: Operatör fonksiyonu bir sınıf'a ait olduğunda **this** işaretçisi üzerinden üstü kapalı olarak gönderilen soldaki terim, operatör fonksiyonunu çağırılan nesnedir. Bu da sol terim bu sınıf'a ait bir nesnedir demektir. Sonuç olarak eğer aşırı yüklenmiş operatör fonksiyonu bir sınıfın üyesi ise sol terim bu sınıf'a ait bir nesne olmazdır. Fakat, bir ekleme fonksiyonu oluşturduğumuzda sol terim bir akımdır ve sağ terim ise çıkış olarak vermek istediğiniz nesnedir. Bunun sonucu olarak da bir ekleme fonksiyonu, bir üye fonksiyon olamaz.

Ekleme fonksiyonunun üye fonksiyon olamaması C++'ın ciddi bir kusuru gibi görünebilir. Çünkü ekleme fonksiyon kullanılarak çıkışa verilecek sınıf üyesi verilerin, public olması gereklidir ve bu da depolama özelliğinin anahtar prensibini ihlal eder gibi görünüyor. Fakat, durum böyle değil. Her ne kadar ekleme fonksiyonları çalışacakları sınıflara ait olmasalar da bu sınıf arkadaşı (*friend*) olabilirler. Gerçekte çoğu zaman aşırı yüklenmiş bu ekleme fonksiyonlarını kendisi için oluşturdukları sınıfın *friend*'i kabul edeceğiz.

Alıştırmalar

- İlk örneğimizde **coord** sınıfı için ekleme fonksiyonu içeren bir program inceleyelim. **coord** sınıfını bir önceki bölümde ele almıştık:

```
// coord tipinde nesneler için arkadaş ekleme fonksiyonu kullanılıyor.
#include <iostream>
using namespace std;
class coord {
    int x, y;
public:
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
    friend ostream &operator<<(ostream &stream, coord ob);
};
ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}
```

```
int main()
{
    coord a(1, 1), b(10, 23);
    cout << a << b;
    return 0;
}
```

Bu program bize şu sonucu verecek:

```
1, 1
10, 23
```

Bu programdaki ekleme fonksiyonunu incelediğimizde kendi ekleme fonksiyonlarını oluştururken dikkat etmemiz gereken önemli bir noktayı fark ediyoruz: Ekleme fonksiyonlarını mümkün olduğunca genel yapmak zorundayız. Ekleme fonksiyonunun içerisindeki I/O deyimi **x** ve **y**'nin değerlerini **stream**'e gönderir. Burada **stream** fonksiyona gönderilen akımdır. Bir sonraki bölümde de göreceğiniz gibi, ekrana çıkış gönderen ekleme fonksiyonu doğru şekilde yazılırsa *herhangi bir* akıma da çıkış gönderebilir. Yeni başlayanlar, **coord** ekleme fonksiyonunu şu şekilde yazabilirler:

```
ostream &operator<<(ostream &stream, coord ob)
{
    cout << ob.x << ", " << ob.y << '\n';
    return stream;
}
```

Bu programda çıkış deyimi **cout**'a bağlı Standart çıkış aygıtına bilgi göndermek için zorlanmaktadır. Fakat, bu durum, ekleme fonksiyonunun diğer akımlar tarafından kullanılmasını engeller. Bu nedenle yapmamız gereken şu, ekleme fonksiyonlarını mümkün olduğu kadar genel hale getirmektir, çünkü bu yapmamızın hiçbir dezavantajı yoktur.

2. Yukarıdaki programı bir kez daha ele alıyoruz, fakat bu sefer ekleme fonksiyonu (*inserter*) **coord** sınıfına ait bir arkadaş *değil*. Çünkü ekleme fonksiyonunun, **coord** sınıfının private kısımlarına erişim hakkı yoktur, **x** ve **y** değişkenleri public yapılmalıdır.

```
/* coord tipinde nesneler için ekleme fonksiyonu oluşturun.
   Arkadaş olmayan bir ekleme fonksiyonu kullanın. */
#include <iostream>
using namespace std;
class coord {
public:
    int x, y; // public olmalı
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
};
// coord sınıfı için ekleme fonksiyonu.
ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}
```

```

    }
int main()
{
    coord a(1, 1), b(10, 23);
    cout << a << b;
    return 0;
}

```

3. Ekleme fonksiyonlarının yapacağı işler ekrana yazı göndermekten ibaret değildir. Ekleme fonksiyonu, belirli bir aygit veya durumun gerektirdiği formda bilgi çıkışını yapmak için gerekli herhangi bir işlem ve dönüştürme işlemini gerçekleştirebilir. Örneğin, plotter'a bilgi gönderen bir ekleme fonksiyonu oluşturabiliriz. Burada ekleme fonksiyonunun bilgiye ek olarak doğru plotter kodlarını da göndermesi gerekecektir. Ekleme fonksiyonlarının getirdiği bu rahatlıklarını görmek için aşağıdaki programı biraz inceleyelim. Bu programda **triangle** isminde bir sınıf oluşturuyoruz, bu sınıf bir eşkenar üçgenin alanını ve yüksekliğini saklıyor. Bu sınıfın ekleme fonksiyonu ise bu üçgeni ekranda gösteriyor.

```

// Bu program bir eşkenar üçgen çiziyor
#include <iostream>
using namespace std;

class triangle {
    int height, base;
public:
    triangle(int h, int b) { height = h; base = b; }
    friend ostream &operator<<(ostream &stream, triangle ob);
};

// Üçgen çiziliyor.
ostream &operator<<(ostream &stream, triangle ob)
{
    int i, j, h, k;

    i = j = ob.base-1;
    for(h=ob.height-1; h; h--) {
        for(k=i; k; k--)
            stream << ' ';
        stream << '**';

        if(j!=i) {
            for(k=j-i-1; k; k--)
                stream << ' ';
            stream << '**';
        }

        i--;
        stream << '\n';
    }
    for(k=0; k<ob.base; k++) stream << '**';
    stream << '\n';

    return stream;
}

int main()

```

```
    triangle t1(5, 5), t2(10, 10), t3(12, 12);  
  
    cout << t1;  
    cout << endl << t2 << endl << t3;  
  
    return 0;  
}
```

Bu programda iyi tasarlanmış bir ekleme fonksiyonunun "normal" bir I/O ifadesine nasıl entegre edilebileceğini görüyoruz. Bu program bize şu sonucu verecek:



Aliştırmalar

1. Size aşağıdaki program içerisinde **strtype** sınıfı veriliyor. Katar gösteren bir ekleme fonksiyonu oluşturun:

```
#include <iostream>  
#include <cstring>  
#include <cstdlib>  
using namespace std;  
  
class strtype {  
    char *p;  
    int len;  
public:  
    strtype(char *ptr);  
    ~strtype() (delete [] p);  
    friend ostream &operator<<(ostream &stream, strtype &obj);  
};
```

```

strtype::strtype(char *ptr)
{
    len = strlen(ptr)+1;
    p = new char [len];
    if(!p) {
        cout << "Bellekte yer ayırma hatası \n";
        exit(1);
    }
    strcpy(p, ptr);
}

// Burada << operatörünün ekleme fonksiyonunu oluşturun.
int main()
{
    strtype s1("Bu bir deneme."), s2("C'yi seviyorum.");
    cout << s1 << '\n' << s2;

    return 0;
}

```

2. Programın içerisindeki `show()` fonksiyonunu bir ekleme fonksiyonu ile değiştirin:

```

#include <iostream>
using namespace std;

class planet {
protected:
    double distance; // güneşe mil cinsinden uzaklığı
    int revolve; // gün cinsinden uzaklığı
public:
    planet(double d, int r) { distance = d; revolve = r; }

    class earth : public planet {
        double cevre; // yörünge yarıçapı
    public:
        earth(double d, int r) : planet(d, r) {
            cevre = 2*distance*3.1416;
        }

        /* Bilgiyi ekrana göndermek için ekleme fonksiyonu kullanacak şekilde
         programı değiştirin. */
        void show() {
            cout << "Güneşten uzaklığı : " << distance << '\n';
            cout << "Yörüngeyi kaç günde tamamladığı : " << revolve << '\n';
            cout << "Yörüğenin çevresi : " << cevre << '\n';
        }
    };
};

int main()
{
    earth ob(93000000, 365);
    cout << ob;
    return 0;
}

```

3. Ekleme fonksiyonlarının neden üye fonksiyon olamadığını açıklayın.

8.6. Extractor'ların Oluşturulması

Tipki << çıkış operatörünü aşırı yükleyebildiğimiz gibi, >> giriş operatörünü de aşırı yükleyebiliriz. C++'da, >> çıkarma operatörü olarak bilinir ve onu aşırı yükleyen fonksiyon ise *extractor* (*cıkarma*) fonksiyonudur. Bu terimi kullanmamızın amacı akımdan bilgi alınması işleminin sonucunda akımda bilgi kalmamasıdır.

Cıkarma fonksiyonunun genel kullanım şekli şudur:

```
istream &operator>>(istream &stream, class-name &ob)
{
    // çıkış fonksiyonunun gövdesi
    return stream;
}
```

Cıkarma fonksiyonları *istream*'i gösteren bir referans döndürürler, bu da bir giriş akımıdır. İlk parametrenin bir giriş akımını göstermesi gerekiyor. İkinci parametre ise girilen bilgiyi alan nesneyi göstermelidir.

Tipki ekleme fonksiyonunun üye fonksiyon olamaması gibi, çıkış fonksiyonu da aynı nedenden dolayı üye fonksiyon olamaz. Çıkarma fonksiyonu içerisinde her tür işlemi gerçekleştirebilirsiniz, fakat en iyisi bu fonksiyonu sadece giriş işlemleri için kullanmak.

Aliştırmalar

- Bu programda **coord** sınıfına bir de çıkış fonksiyonu ekliyoruz:

```
// coord tipinde nesneler için bir arkadaş çıkış fonksiyonu ekleyin.
#include <iostream>
using namespace std;

class coord {
    int x, y;
public:
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
    friend ostream &operator<<(ostream &stream, coord ob);
    friend istream &operator>>(istream &stream, coord &ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}

istream &operator>>(istream &stream, coord &ob)
{
    cout << "Koordinatları girin : ";
    stream >> ob.x >> ob.y;
    return stream;
}

int main()
{
```

```

coord a(1, 1), b(10, 23);
cout << a << b;

cin >> a;
cout << a;

return 0;
}

```

Çıkarma fonksiyonunun kullanıcıdan bilgi girmesini nasıl istedigine dikkat edelim. Çoğu kez buna ihtiyacımız olmayacaksa da (veya istenilmeyecekse de), bu fonksiyonu inceleyerek iyi düzenlenmiş bir çalışma fonksiyonunun kullanıcıya mesaj gönderme işlemini nasıl kolaylaştırdığını görebiliriz.

- Burada bir mal stoku sınıfı oluşturun. Sınıfınız malın ismini, elinizde bulunan miktarı ve malın fiyatını saklasın. Programda bu sınıfa ait bir ekleme fonksiyonu bir de çalışma fonksiyonu bulunuyor.

```

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

class inventory {
    char item[40]; // malın ismi
    int onhand; // elimizdeki miktar
    double cost; // malın fiyatı
public:
    inventory(char *i, int o, double c)
    {
        strcpy(item, i);
        onhand = o;
        cost = c;
    }
    friend ostream &operator<<(ostream &stream, inventory ob);
    friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
    stream << ob.item << ": elimizde " << ob.onhand;
    stream << " tane var ve fiyatı $" << ob.cost << '\n';
    return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
    cout << "Malın ismini girin: ";
    stream >> ob.item;
    cout << "Elinizde kaç tane var? ";
    stream >> ob.onhand;
    cout << "Malın fiyatını girin : ";
    stream >> ob.cost;

    return stream;
}

int main()

```

```

    inventory ob("cekic", 4, 12.55);
    cout << ob;
    cin >> ob;
    cout << ob;
    return 0;
}

```

Aliştırmalar

1. Bir önceki konunun 1. Aliştırmasındaki **strtype** sınıfına bir çıkarma fonksiyonu ekleyin.
2. Bir tamsayıyı ve bu tamsayının en küçük ortak bölenini saklayan bir sınıf oluşturun. Bu sınıf için bir ekleme fonksiyonu bir de çıkarma fonksiyonu oluşturun.

Pekştirme Testi

Bu noktada aşağıdaki alıştırmaları yapabiliyor ve soruları cevaplayabiliyor olmalısınız.

1. 100 sayısını onluk, onaltılık ve sekizlik düzende gösteren bir program yazın. (**ios** format bayraklarını kullanın.)
2. 1000.5364 sayısını 20 karakterlik bir alanda, sola hizalı, virgülden sonra iki rakam olacak şekilde ve boşluk karakteri de * olacak şekilde gösteren bir program yazın. (**ios** format bayraklarını kullanın.)
3. Aliştırma 1 ve 2'ye verdığınız cevapları I/O manipülatörlerini kullanacak şekilde tekrar yazın.
4. **cout** için format bayraklarının değerlerini nasıl saklayacağınızı ve nasıl tekrar düzenleyecelerini gösterin. Üye fonksiyonları ya da manipülatörleri kullanın.
5. Bu sınıf için bir ekleme fonksiyonu bir de çıkarma fonksiyonu oluşturun:

```

class pwr {
    int base;
    int exponent;
    double result; // üslü sayının tabanı
public:
    pwr(int b, int e);
};
pwr::pwr(int b, int e)
{
    base = b;
    exponent = e;

    result = 1;
    for( ; e; e--) result = result * base;
}

```

6. **box** isminde bir sınıf oluşturun. Bu sınıf karenin kenar uzunluklarını saklasın. Ekran da kare gösteren bir ekleme fonksiyonu oluşturun. (Kareyi göstermek için istediğiniz metodу kullanabilirsiniz.)

Bütünleştirme Testi

Burada bir önceki bölümle bu bölümde öğrendiklerinizi ne kadar iyi bir araya getirebildiğiniz kontrol edilmektedir.

1. Burada verilen **stack** sınıfını kullanarak yiğinin içeriğini gösteren bir ekleme fonksiyonu oluşturun. Ekleme fonksiyonunun çalıştığını örnek vererek gösterin.

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

#define SIZE 10

// karakterler için bir stack sınıfı deklare edin
class stack {
    char stck[SIZE]; // stack'ı (yığını) tutar
    int tos; // top-of-stack'in indeksi
public:
    stack();
    void push(char ch); // yiğine karakter iter
    char pop(); // yiğinden karakter çekter
};

// Initialize stack
stack::stack()
{
    tos = 0;
}

// Bir karakter itiliyor.
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Yığın dolu \n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Bir karakter çekiliyor.
char stack::pop()
{
    if(tos==0) {
        cout << "Yığın boş \n";
        return 0; // yiğin boşsa sıfır gönderir
    }
    tos--;
    return stck[tos];
}
```

2. **watch** isminde bir sınıf içeren programınızı oluşturun. Standart zaman fonksiyonlarını kullanarak bu sınıfın ait constructor fonksiyonunun sistem saatini okumasını ve saklamasını sağlayın. Zamanı gösteren bir eklem fonksiyonu oluşturun.
3. Fit'i içe çeviren aşağıdaki sınıfı kullanarak kullanıcıdan fit değerini girmesini isteyen çıkarma fonksiyonunu oluşturun. Aynı zamanda fit ve inç değerlerini ekranda gösteren bir ekleme fonksiyonu oluşturun. Ekleme ve çıkarma fonksiyonlarınızın çalıştığını gösteren bir de program yazın.

```
class ft_to_inches {  
    double feet;  
    double inches;  
public:  
    void set(double f) {  
        feet = f;  
        inches = f * 12;  
    }  
};
```

BÖLÜM 9

İleri Düzey C++ I/O'su

Kendi Manipülatörlerinizi Oluşturun
Dosya I/O'sunun Temelleri
Biçimlendirilmemiş, İkili I/O
Rasgele Erişim
I/O Durumunun Denetlenmesi
Özelleştirilmiş I/O ve Dosyalar

Bu bölümde C++'ın I/O sistemini incelemeye devam edeceğiz. Burada kendi manipülatörlerinizi oluşturmayı ve dosyalarla çalışmayı öğreneceğiz. Sunu aklımızdan çıkarmayalım: C++'ın I/O sistemi, zengin ve esnek olmanın yanı sıra pek çok özelliğe de sahiptir. Burada bu özelliklerin tamamını ele almamıza imkan olmadığından biz en önemli olanlarını öğreneceğiz.

NOT Bu bölümde anlattığımız C++'ın I/O sistemi Standart C++ tarafından tanımlanarı yansımaktadır ve belli başlı C++ derleyicileriyle de uyumludur. Eğer derleyiciniz eskiyse ve uyum sağlamıyorsa, I/O sistemi burada anlattıklarımızın hepsini gerçekleştiremeyecektir.

Gözden Geçirme Testi

Bu bölümde başlamadan önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

1. Doldurma karakteri iki nokta üst üste (:) olacak şekilde 40 karakter genişliğinde bir alana "C++ çok eğlenceli" cümlesini yazdıracak bir program yazın.
2. 10/3 işleminin sonucunu üç haneli olarak gösteren bir program yazın. Bunun için `ios` üye fonksiyonlarını kullanın.
3. Yukarıdaki program I/O manipülatörlerini kullanarak tekrar yazın.
4. Ekleme fonksiyonu ve çıkarma fonksiyonu nedir?
5. Aşağıdaki sınıf veriliyor, bu sınıf için bir ekleme bir de çıkarma fonksiyonu oluşturun.

```
class date {
    char d[9]; // tarihi katar şeklinde saklayın: gg/aa/yy
public:
    // buraya ekleme ve çıkarma fonksiyonu ekleyin.
};
```

6. Programınız, parametre alan I/O manipülatörlerini kullandığı hangi başlığı dahil etmeniz gerekiirdi?
7. C++ programı çalışmaya başladığında önceden tanımlanmış akımlardan hangileri oluşturulur?

9.1. Kendi Manipülatörlerinizi Oluşturun

C++'ın I/O sistemini özelleştirmek için ekleme ve çıkarma fonksiyonlarının aşırı yüklenmesine ek olarak kendi manipülatör fonksiyonlarınızı da oluşturabilirsiniz. Özel (custom) manipülatörlerin önemini olmasının iki sebebi var. Birincisi, bir manipülatörün ayrı durumda bulunan çeşitli I/O işlemlerini bir araya getirebilmesidir. Program içerisinde çok sık meydana gelen I/O işlemlerinin bulunması hiç de alışılmadık bir şey değil. Bu durumda işlemleri gerçekleştirmek için custom manipülatörünü kullanabilirsiniz. Bu da kaynak

kodunuzun basit hale gelmesini sağlar ve çeşitli hataların ortaya çıkmasını engeller. İkinci nedeni ise şu: custom manipülatörler standart olmayan aygıtlarla I/O işlemlerini gerçekleştirmeniz gerekiğinde de önemli olabilir. Örneğin, özel tipte bir yazıcıya veya bir optik tanıma sistemine kontrol kodları göndermek için manipülatör kullanabilirsiniz.

Custom manipülatörler, C++'in OOP'u destekleyen bir özellikleidir, fakat nesne yönelimli olmayan programlar da onlardan faydalanabilirler. Görüğünüz gibi custom manipülatörler I/O işlemlerinin sık olduğu programları daha açık ve daha verimli hale getirmenize yardım eder.

Bildığınız gibi manipülatörler temelde ikiye ayrılır: giriş akımları üzerinde çalışanlar ve çıkış akımları üzerinde çalışanlar. Bunlara ek olarak bir kategori daha oluşturabiliriz: argüman alanlar ve almayanlar. Parametresi olmayan manipülatörlerle parametreli manipülatörlerin oluşturulmuşları arasında önemli farklar vardır: Hatta parametreli manipülatörlerin oluşturulması, parametresizlerin oluşturulmasından çok daha zordur ve bu nedenle de kitabımıza bunu almayacağız. Halbuki kendi parametresiz manipülatörlerinizi yazmanız oldukça kolaydır, biz de bu konuya yer verdik. Parametresiz tüm manipülatör giriş fonksiyonları bu iskelete sahiptir:

```
ostream &manip-name(ostream &stream)
{
    // kodunuza buraya yazacaksınız.
    return akım;
}
```

Burada *manip-name* manipülatörünün ismidir ve *stream* akımı uyandıran referansıdır. Akımı gösteren bir referans döndürülür. Bu, manipülatör daha büyük bir I/O ifadesinin parçası ise gereklidir. Sunu anlamamız önemli: her ne kadar manipülatörün, işlemekte olan akımı gösteren tek bir argümanı varsa da, manipülatör bir çıkış işleminde çağrıldığında argüman kullanılmaz. Parametresiz tüm giriş manipülatör fonksiyonları bu iskelete sahiptir:

```
istream &manip-name(istream &stream)
{
    // kodunuza buraya yazacaksınız.
    return akım;
}
```

Bir giriş manipülatörü uyandırıldığı akımı gösteren bir referans alır. Bu akım manipülatör tarafından döndürülmeliidir.

HATIRLATMA Manipülatörlerin uyaran akıma referans döndürmesi önemlidir. Eğer bu yapılmazsa manipülatörleriniz giriş çıkış işlemleri sırasında kullanılamaz.

Örnekler

- İlk örnek olarak aşağıdaki program **setup()** isminde, alan genişliğini 10'a, toplam hane sayısını 4'e ve doldurma karakterini '*'a ayarlayan bir manipülatör oluşturmaktadır.

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

ostream &setup(ostream &stream)
{
    stream.width(10);
    stream.precision(4);
    stream.fill('*');
    return stream;
}

int main()
{
    cout << setup << 123.123456;
    return 0;
}
```

Sizin de gördüğünüz gibi, **setup** mevcut manipülatörlerle aynı şekilde bir I/O ifadesinin parçası olarak kullanılmaktadır.

- Özel manipülatörlerin kullanışlı olmaları için karmaşık olmaları gerekmekz. Örneğin burada gösterilen, basit **atn()** ve **note()** sık kullanılan kelime ve deyimlerin kısa bir şekilde çıkışını sağlar.

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

// Dikkat:
ostream &atn(ostream &stream)
{
    stream << "Attention: ";
    return stream;
}

// Lütfen not edin!:
ostream &note(ostream &stream)
{
    stream << "Please Note: ";
    return stream;
}

int main()
{
    cout << atn << "High voltage circuit\n";
    cout << note << "Turn off all lights\n";
    return 0;
}
```

Her ne kadar basit de olsalar sık kullanıldıkları takdirde bu manipülatörler sizi sıkıcı uzun yazma işlerinden kurtarır.

3. Bu program zil çalan be ekranдан şifre isteyen `getpass()` giriş manipülörünü oluşturmaktadır:

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

// Basit bir giriş manipülörü
istream &getpass(istream &stream)
{
    cout << '\a'; // zil sesi
    cout << "Enter password: ";
    return stream;
}

int main()
{
    char pw[80];
    do {
        cin >> getpass >> pw;
    } while (strcmp(pw, "password"));
    cout << "Logon complete\n";
    return 0;
}
```

Alıştırmalar

1. Sistem tarihi ve zamanını gösteren bir çıkış manipülörünü oluşturun. Bu manipülatöre `td()` ismini verin.
2. Çıkışı 16'hk taban şeklinde veren, `uppercase` ve `showbase` bayraklarını bir yapan, `sethex()` isminden bir çıkış manipülörünü oluşturun. `sethex()` tarafından yapılan değişiklikleri geri alan `reset()` isminden bir çıkış manipülörünü daha oluşturur.
3. Giriş akımından okuyan ve daha sonraki 10 karakteri göze almayan `skipchar()` adında bir giriş manipülörünü oluşturun.

9.2. Dosya I/O Temelleri

Artık dikkatimizi dosya I/O'suna verme zamanı geldi. Daha önceki bölümde belirtildiği gibi dosya I/O'su ve konsol I/O'su birbirleriyle yakından ilişkilidir. Aslında, konsol I/O'sunu destekleyen sınıf hiyerarşisi dosya I/O'sunu da destekler. Böylece, I/O hakkında öğrendiğiniz çoğu şeyi dosyalara da uygulamak mümkündür. Tabii ki, dosyalarla çalışmak için bazı yeni özellikler de eklenmiştir.

Dosya I/O'su işlemini gerçekleştirmek için programınıza `<fstream>` başlığını eklemelisiniz. Bu, `ifstream`, `ofstream` ve `fstream` dahil olmak üzere çeşitli sınıfları tamamlamaktadır. Bu sınıflar `istream` ve `ostream`'den türetilmiştir. Hatırlayın, `istream` ve `ostream`, `ios`'dan türetilmiştir ve bu yüzden `ifstream`, `ofstream` ve `fstream` `ios` tarafından tanımlanan tüm işlemlere erişim hakkına sahiptir (bir önceki bölümde tartışılmıştı).

C++'da dosyalar akımlara bağlanarak açılırlar. Üç çeşit akım vardır: giriş, çıkış ve giriş/çıkış. Dosya açmadan önce bir akım elde etmelisiniz. Bir giriş akımı oluşturmak için **ifstream** tipinde bir nesne deklare edin. Çıkış akımı oluşturmak için **ofstream** tipinden bir nesne deklare edin. Hem giriş hem çıkış işlemlerini gerçekleştirecek akımlar **fstream** tipinde nesneler olarak deklare edilmelidir. Örneğin bu kod bir giriş akımı, bir çıkış akımı ve de giriş ve çıkışı aynı anda yapabilen başka bir akım oluşturmaktadır:

```
ifstream in; // giriş
ofstream out; // çıkış
fstream io; // giriş ve çıkış
```

Akımları bir kere oluşturduktan sonra onu bir dosyaya bireleştirmenin bir yolu **open()** fonksiyonunu kullanmaktadır. Bu fonksiyon her üç akım sınıfının üyesidir. Her birinin prototipini aşağıda sizin için veriyorum:

```
void ifstream::open(const char *filename, openmode mode = ios::in);
void ofstream::open(const char *filename, openmode mode = ios::out |
ios::trunc);
void fstream::open (const char *filename, openmode mode = ios::in | ios::out);
```

Yol belirtecini içerebilen buradaki *filename* dosyanın ismini gösterir. *Mode* değeri dosyanın nasıl açılacağını belirler. Bu, aşağıdaki değerleri içeren, ios tarafından tanımlanmış bir sayımlardır.

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```

Bu değerlerden iki veya daha fazlasını **or** işlemine tabi tutarak bireleştirebilirsiniz. Bu değerlerin ne anlama geldiklerini beraberce görelim:

ios::app'in eklenmesi çıkışın tamamının dosyanın sonuna eklenmesine sebep olur. Bu değer sadece çıkış yapılabilecek dosyalarla kullanılabilir. **ios::ate**'in eklenmesi dosya açıldığında dosyanın sonunun aranmasına neden olur. **ios::ate**'in dosyanın sonunun aranmasına neden olsa da I/O işlemleri dosya içerisinde herhangi bir yerde yapılabilir.

ios::in değeri dosyanın girişe uygun olduğunu, **ios::out** değeri ise dosyanın çıkışa uygun olduğunu belirler.

ios::binary değeri dosyanın ikili modda açılmasını sağlar. Normalde tüm dosyalar metin kipinde açılır. Metin (Text) kipinde, satırbaşı, satır besleme işaretlerinin yeni satırlara dönüştürülmesi gibi çeşitli karakter dönüşümleri yapılabilir. Fakat dosya ikili kipte açıldığında bu tür karakter dönüşümleri yapılamaz. Herhangi bir dosya ister biçimlendirilmiş metin ister düz metin olsun ikili veya metin kipinde açılabilir. Aralarındaki tek fark karakter dönüşümlerinin yapılmış yapılmamıştır.

`ios::trunc` değeri, aynı isimdeki eski dosyanın silinmesine ve dosya uzunluğunun sıfıra indirilmesine neden olur. `ofstream` kullanarak çıkış akımı oluşturduğunuzda aynı isimdeki eski dosyalar kendiliğinden sıfırlanır.

Aşağıdaki kod `test` isminde bir çıkış dosyası açar:

```
ofstream mystream;
mystream.open("test");
```

`open()` fonksiyonunun *mode* parametresi, açılmakta olan akımın tipine uygun bir değer alır. Bu nedenle yukarıdaki örnekte onun değerini belirlemeyeceğiz.

`open()` başarısızlığa uğrarsa akım, boolean bir ifadede kullanıldığında `false`'a karşılık gelir. Bu durumdan, aşağıdaki gibi bir deyim kullanarak açma işleminin başarıya ulaşğını anlamak için faydalananabiliriz:

```
if(!mystream) {
    cout << "Cannot open file.\n";
    // hatayı kontrol et
}
```

Genelde dosyaya erişmeden önce `open()`'a yaptığınız çağrıının sonucunu kontrol etmeniz gereklidir. `fstream`, `ifstream` ve `ofstream`'in üyesi olan `is_open()` fonksiyonu ile de dosyayı düzgün açıp açmadığınızı kontrol etmeniz gereklidir. Bu fonksiyon prototipi budur:

```
bool is_open();
```

Eğer akım dosyaya bağlanmışsa `true` aksi halde `false` değerini verir. Örneğin aşağıdaki kod `mystream`'in açık olup olmadığını kontrol eder:

```
if(!mystream.is_open()) {
    cout << "File is not open.\n";
    // ...
```

Dosya açmak için `open()` fonksiyonunun kullanılmasında hiçbir yanlışlık yoktur, fakat genelde onu kullanmayacaksınız. Çünkü `ifstream`, `ofstream` ve `fstream` sınıflarının otomatik olarak dosya açan constructor fonksiyonları vardır. Constructor fonksiyonlarının `open()` fonksiyonu gibi bazı parametreleri ve varsayılan değerleri vardır. Sonuç olarak en sık kullanılan dosya açma şekli şöyledir:

```
ifstream mystream("myfile"); // giriş için dosya açılıyor
```

Belirttiğimiz gibi, eğer bazı nedenlerden dolayı dosya açılamazsa akım değişkeni, koşullu ifadelerde kullanıldığındaysa `false` değer verecektir. Sonuç olarak dosya açmak için ister constructor fonksiyonu kullanın ister `open()`'a açık bir çağrı yapın akımın değerini test ederek dosyanın gerçekten açılıp açılmadığını onaylamak isteyeceksiniz.

Dosya kapamak için `close()` üye fonksiyonunu kullanın. Örneğin `mystream` ismindeki akıma bağlı bir dosyayı kapamak için şu deyişi kullanın:

```
mystream.close();
```

`close()` fonksiyonu parametre almaz ve değer döndürmez.

`ios`'un `eof()` üye fonksiyonunu kullanarak giriş dosyasının sonuna ulaşıp ulaşmadığını anlayabilirsiniz. Prototipi şu şekildedir:

```
bool eof();
```

Dosyamın sonuna gelindiğinde true, aksi halde false değerini verir. Dosya açıldıktan sonra içinden metin okumak veya üzerine biçimlendirilmiş metin yazmak çok kolaydır. `<<` ve `>>` operatörlerini konsol I/O'su gerçekleştirirken kullandığınız şekilde kullanın. Fakat `cin` ve `cout` kullanmak yerine dosyaya bağlı bir akım kullanın. Bir açıdan `>>` ve `<<` kullanılarak dosya okuma ve yazma işlemlerinin gerçekleştirilmesi C'nin `fprintf()` ve `fscanf()` fonksiyonlarının kullanılmasına benzer. Bilgilerin tümü dosyaya ekranda gösterileceği biçimle kaydedilir. Sonuç olarak `<<` kullanılarak oluşturulan bir dosya biçimlendirilmiş bir metin dosyasıdır ve `>>` kullanılarak okunan her dosya biçimlendirilmiş bir metin dosyası olmalıdır. Genelde `>>` ve `<<` operatörlerini kullanarak istediğiniz biçimlendirilmiş metin içeren dosyalar ikili mod yerine metin olarak açılmalıdır. İkili mod, daha sonra bahsedeceğimiz biçimlendirilmemiş dosyalarda en iyi şekilde kullanılır.

Örnekler

1. Bu program bir çıkış dosyası oluşturur, üzerine bilgi yazar, dosyayı kapatır ve onu tekrar bir giriş dosyası olarak açar ve içindeki bilgileri okur:

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

int main()
{
    ofstream fout("test"); // çıkış dosyası oluştur

    if(!fout) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    fout << "Hello!\n";
    fout << 100 << ' ' << hex << 100 << endl;

    fout.close();

    ifstream fin("test"); // giriş dosyası açar

    if(!fin) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    char str[80];
    int i;

    fin >> str >> i;
```

```

    cout << str << ' ' << i << endl;
    fin.close();
    return 0;
}

```

Programı çalıştırdıktan sonra **test**'in içeriğini inceleyin. Aşağıdakileri içerecektir:

```
Hello!
100 64
```

Daha önce de belirtmiştık, **<<** ve **>>** operatörleri dosya I/O'su gerçekleştirmek için kullanılır. Bilgi tam olarak ekranda gözüktüğü gibi biçimlendirilir.

- Aşağıda disk I/O'suna verilen başka bir örnek veriyoruz. Bu program klavyeden girilen katarları okur onları diske yazar. Program kullanıcı **\$** girerse sona erer. Programı kullanmak için komut satırında çıkış dosyasının adını belirleyin.

```

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

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Usage: WRITE <filename>\n";
        return 1;
    }

    ofstream out(argv[1]); // çıkış dosyası

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    char str[80];
    cout << "Write katars to disk, '$' to stop\n";

    do {
        cout << ": ";
        cin >> str;
        out << str << endl;
    } while (*str != '$');

    out.close();
    return 0;
}

```

- Bu program bir metin dosyasını kopyalar ve tüm boşlukları **|** sembollerine dönüştürür. **eof()**'un giriş dosyasının sonunun kontrolü için nasıl kullanıldığına dikkat edin. **fin** giriş akımının kendi **skipws** bayraklarını nasıl 0 yapığına da dikkat edin. Bu baştaki boşlukların atlanmasını engeller.

```
// Boşluklar '|'lere dönürtülüyor.
#include <iostream>
```

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

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Usage: CONVERT <input> <output>\n";
        return 1;
    }

    ifstream fin(argv[1]); // giriş dosyasını açın
    ofstream fout(argv[2]); // çıkış dosyasını oluştur
    if(!fout) {
        cout << "Cannot open output file.\n";
        return 1;
    }
    if(!fin) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    char ch;

    fin.unsetf(ios::skipws); // boşlukları atlama
    while(!fin.eof()) {
        fin >> ch;
        if(ch==' ' || ch == '\t')
            if(!fout.eof()) fout << ch;
    }
    fin.close();
    fout.close();
    return 0;
}
```

4. C++'in orijinal I/O kütüphanesi ile modern Standart C++ kütüphanesi arasında, özellikle eski kodlarınızı dönüştürüyorsanız bilmeniz gereken birkaç fark bulunmaktadır. Bunlardan ilki şudur: orijinal I/O kütüphanesinde **open()** dosyanın koruma modunu belirleyen üçüncü bir parametreye sahipti. Bu parametre varsayılan olarak normal bir dosyayı belirliyordu. Modern I/O kütüphanesi bu parametreyi desteklemez.

İkinci fark da şudur: giriş ve çıkış için **fstream**'ı kullanarak bir akım açmak için eski kütüphaneyi kullanıyorsanız **ios::in** ve **ios::out mode** değerlerini açıkça belirlemelisiniz. **Mode**'un varsayılan değeri bulunmamaktadır. Bu hem **fstream** constructor'u hem de onun **open()** fonksiyonu için geçerlidir. Örneğin eski I/O kütüphanesini kullandığınızda burada gösterildiği gibi giriş ve çıkış için bir dosya açmak istiyorsanız **open()** fonksiyonunu çağrırmalısınız:

```
fstream mystream;
mystream.open("test", ios::in | ios::out);
```

Modern I/O kütüphanesinde, **fstream** tipinde bir nesne **mode** parametresi verilmesi gerektiğinde otomatik olarak giriş ve çıkış için dosyalar açar.

Son olarak da eski I/O sisteminde *mode* parametresi, *open()* fonksiyonunun dosya yoksa başarısızlığa uğramasına neden olan kendi *ios::nocreate*'ini veya dosya zaten varsa başarısızlığa uğramasına neden olan kendi *ios::noreplace*'ini de dahil ederdi. Bu değerler Standart C++ tarafından desteklenmemektedir.

Aliştırmalar

1. Metin dosyası kopyalayan bir program yazın. Bu program kopyalanan karakter sayısını hesaplasın ve ekranda göstersin. Dizindeki çıkış dosyasını listelediğinizde gördüğünüz ile burada gösterilen neden farklıdır?
2. Aşağıdaki tablodaki bilgileri **phone** ismindeki bir dosyaya yazan bir program oluşturun:

Isaac Newton, 415 555-3423
Robert Goddard, 213 555-2312
Enrico Fermi, 202 555-1111
3. Dosyadaki kelime sayısını sayan bir program yazın. Basit olsun diye iki tarafı da boş olan her şeyi kelime farz edin.
4. *is_open()* ne yapar?

9.3. Biçimlendirilmemiş İkili I/O

Önceki örneklerde oluşturduğumuz gibi biçimlendirilmiş metin dosyaları çeşitli durumlarda faydalı olabilir, fakat biçimlendirilmemiş ikili dosyaların esnekliğine sahip değildir. Biçimlendirilmemiş dosyalar << ve >> operatörleri kullanılarak verinin dönüştürüldüğü insan tarafından okunabilen metin yerine programınızın da içinde bulunan aynı ikili veri sunumunu içerir. Bu nedenle biçimlendirilmemiş I/O aynı zamanda "işlenmemiş" I/Oolarak da adlandırılır. C++ pek çok biçimlendirilmemiş dosya I/O fonksiyonunu da desteklemektedir. Biçimlendirilmemiş fonksiyonlar, dosyaların nasıl yazılması ve okunması gereği konusunda size detaylı bir kontrol sağlamaktadır.

En düşük seviyedeki biçimlendirilmemiş I/O fonksiyonları *get()* ve *put()*'tur. *get()*'i kullanarak bir byte okuyabilir *put()*'u kullanarak da bir byte yazabilirsiniz. Bu fonksiyonlar sırasıyla tüm giriş ve çıkış akım sınıflarının üyesidir. *get()* ve *put()* fonksiyonlarının pek çok çeşidi vardır, fakat en çok kullanılan şekilleri şudur:

```
istream &get(char &ch);
ostream &put(char ch);
```

get() fonksiyonu ilgili akımdan tek bir karakter okur ve bu değeri *ch*'nin içine koyar. Akıma bir referans döndürür. Eğer dosyanın sonuna gelindiğinde okuma işlemi yapılmaya çalışılırsa cevap olarak uyandırıcı akım ifade içerisinde kullanıldığında false değerini verir. *put()* fonksiyonu *ch*'yi akıma yazar ve akıma bir referans döndürür.

Veri bloklarını okumak ve yazmak için sırasıyla giriş ve çıkış akım sınıflarının üyeleri olan **read()** ve **write()** fonksiyonlarını kullanın. Bunların prototipleri şunlardır:

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

read() fonksiyonu, uyandırıcı akımdan gelen *num* byte'larını okur ve onları *buf* tarafından işaret edilen tampon belleğe koyar. **write()** fonksiyonu ise *num* bytelerini *buf* tarafından işaret edilen tampon bellekten alır ve ilgili akıma yazar. **streamsize** tipi bir çeşit tamsayıdır. Bu tipte bir nesne herhangi bir I/O işleminde transfer edilebilecek en çok sayıda byte'ı kaldırabilecek kapasitededir.

Eğer dosyanın sonuna *num* karakterler okunmadan önce ulaşılırsa **read()** durur ve tampon bellek mümkün olduğu kadar karakter içerir. **gcount()** üye fonksiyonu ile kaç tane karakterin okunmuş olduğunu bulabilirsiniz. Bu fonksiyonun prototipi şudur:

```
streamsize gcount();
```

Bu fonksiyon en son biçimlendirilmemiş giriş işlemi tarafından okunan karakter sayısını döndürür. Biçimlendirilmemiş dosya fonksiyonlarını kullanırken metin işlemleri yerine ikili olarak dosya açacaksınız. Bunun nedenini kolayca anlayabilirsiniz. **ios::binary**'nin belirlenmesi herhangi bir karakter dönüşümünün meydana gelmesini engeller. Bu tamsayılar, **float**'lar ve işaretçiler gibi verilerin ikili sunumları dosyada saklandığında önemlidir. Fakat dosya gerçekte sadece metin içeriği sürece, metin kipinde açılan bir dosya üzerinde biçimlendirilmemiş fonksiyonları kullanmanız tamamen doğrudur. Fakat bazı karakter dönüştürmelerinin meydana gelebileceğini unutmayın.

Örnekler

1. Bu program herhangi bir dosyanın içeriğini ekranda gösterecektir. **get()** fonksiyonunu kullanmaktadır.

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

int main(int argc, char *argv[])
{
    char ch;
    if(argc!=2) {
        cout << "Usage: PR <filename>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
        cout << "Cannot open file.\n";
        return 1;
    }
    while(!in.eof()) {
        in.get(ch);
```

```
    cout << ch;
}
in.close();
return 0;
)
```

2. Bu program kullanıcı dolar işaretini girene kadar dosyaya karakter yazmak için `put()`'u kullanmaktadır:

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

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
        cout << "Usage: WRITE <filename>\n";
        return 1;
    }

    ofstream out(argv[1], ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open file.\n";
        return 1;
    }

    cout << "Enter a $ to stop\n";
    do {
        cout << ": ";
        cin.get(ch);
        out.put(ch);
    } while (ch!='$');

    out.close();
    return 0;
}
```

Programın `cin`'den karakterleri okumak için `get()`'ı kullandığına dikkat edin. Bu baştaki boşlukların atlanması önemlidir.

3. Bu program `test` ismindeki dosyaya bir double bir de katar yazmak için `write()`'ı kullanmaktadır:

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

int main()
{
    ofstream out("test", ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }
```

```

    }

    double num = 100.45;
    char str[] = "This is a test";

    out.write((char *) &num, sizeof(double));
    out.write(str, strlen(str));

    out.close();

    return 0;
}

```

NOT

`write()`'a yapılan çağrıının içerisindeki `(char *)` tip zorlaması, karakter dizisi olarak tanımlanmamış bir tampon belleğe gakis yaparken geteklidir. C++'ın güçlü yazım denetimi yüzünden bir tipteki işaretçi otomatik olarak başka tipte bir işaretçiye dönüştürülemeyecektir.

- Bu program Örnek 3'teki program tarafından oluşturulan dosyayı okumak için `read()`'ı kullanmaktadır:

```

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

int main()
{
    ifstream in("test", ios::in | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    double num;
    char str[80];

    in.read((char *) &num, sizeof(double));
    in.read(str, 14);
    str[14] = '\0'; // boş sonlu str
    cout << num << ' ' << str;

    in.close();

    return 0;
}

```

Bir önceki örnekteki durumda olduğu gibi `read()`'in içerisindeki tip zorlaması gereklidir, çünkü C++ bir tipteki işaretçiyi bir başka tipe otomatik olarak dönüştüremeyecektir.

- Aşağıdaki program ilk önce dosyaya bir `double` değerler dizisi yazar ve sonra onları geri okur. Okunan karakter sayısını da rapor eder.

```

// gcount() örneği.
#include <iostream>
#include <fstream>

```

```

using namespace std;

int main()
{
    ofstream out("test", ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    double nums[4] = {1.1, 2.2, 3.3, 4.4};

    out.write((char *) nums, sizeof(nums));
    out.close();

    ifstream in("test", ios::in | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    in.read((char *) &nums, sizeof(nums));

    int i;
    for(i=0; i<4; i++)
        cout << nums[i] << ' ';
    cout << '\n';

    cout << in.gcount() << " characters read\n";
    in.close();

    return 0;
}

```

Alıştırmalar

1. Bir önceki konudaki (Konu 9.2) 1. ve 3. alıştırmalara verdığınız cevapları `get()`, `put()`, `read()` ve/veya `write()`'ı kullanacak şekilde tekrar yazın. (Bu fonksiyonlardan en uygunu sizce hangisiyse onu kullanın.)
2. Size aşağıdaki sınıf veriliyor. Sınıfın içeriğini dosyaya yazan bir program hazırlayın. Bunun için bir ekleyici fonksiyon oluşturun.

```

class account {
    int custnum;
    char name[90];
    double balance;
public:
    account(int c, char *n, double b)
    {
        custnum = c;
        strcpy(name, n);
        balance = b;
    }
    // ekleyici fonksiyonu burada oluşturun
};

```

9.4. Biçimlendirilmemiş Diğer I/O Fonksiyonları

Daha önce verdığımızda ek olarak `get()` fonksiyonunun aşırı yüklediği çeşitli şekiller vardır. Bunların en sık kullanılan üçünün prototipleri şunlardır:

```
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
int get();
```

İlki karakterleri `buf` tarafından işaret edilen dizeye okur. Bunu ya `num-1` karakterleri okunana, ya yani satır bulunana ya da dosyanın sonu gelene kadar yapar. `buf` tarafından işaret edilen dizi `get()` tarafından boş sonlandırılacaktır. Eğer giriş akımında yeni satır karakteri gelirse bu alınmaz. Bir sonraki giriş işlemine kadar akımın içinde kalır.

İkincisi yine karakterleri `buf` tarafından işaret edilen dizeye okur. Fakat bunu ya `num-1` karakterleri okunana, ya `delim` tarafından belirlenen karakter bulunana ya da dosyanın sonu gelene kadar yapar. `buf` tarafından işaret edilen dizi `get()` tarafından boş sonlandırılacaktır. Eğer giriş akımından delim karakteri gelirse bu alınmaz. Bir sonraki giriş işlemine kadar akımın içinde kalır.

`get()`'in aşırı yüklenmiş üçüncü şekli, akımdan bir sonraki karakteri döndürür. Dosyanın sonu geldiğinde EOF döndürür. `get()`'in bu şekli C'nin `getc()` fonksiyonuna benzermektedir.

Giriş gerçekleştiren diğer bir fonksiyon da `getline()` fonksiyonudur. Bu tüm giriş akımı sınıflarının üyesidir. Prototipleri şu şekildedir:

```
istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);
```

Bunlardan ilki karakterleri `buf` tarafından işaret edilen dizeye okur. Bunu ya `num-1` karakterleri okunana, ya yeni satır bulunana ya da dosyanın sonu gelene kadar yapar. `buf` tarafından işaret edilen dizi `getline()` tarafından boş sonlandırılacaktır. Eğer giriş akımında yeni satır karakteri gelirse bu alınır, fakat `buf`'ın içine konmaz.

İkincisi yine karakterleri `buf` tarafından işaret edilen dizeye okur. Fakat bunu ya `num-1` karakterleri okunana, ya `delim` tarafından belirlenen karakter bulunana ya da dosyanın sonu gelene kadar yapar. `buf` tarafından işaret edilen dizi `getline()` tarafından boş sonlandırılacaktır. Eğer giriş akımından delim karakteri gelirse bu alınır, fakat `buf`'ın içine konmaz.

Sizin de görebileceğiniz gibi, `getline()`'ın iki versiyonu `get(buf, num)` ve `get(buf, num, delim)` versiyonlarına gerçekte eşittir. Her ikisi de girişten karakter okur ve onları `buf` tarafından işaret edilen dizeye koyar. Bunu ya `num-1` karakterleri okunana, ya `delim` tarafından belirlenen karakter bulunana ya da dosyanın sonu gelene kadar yapar. `get()` ile `getline()` arasındaki fark, `getline()`'ın delim karakterini okuyup silmesi, `get()`'in ise bunu yapmamasıdır.

Giriş akımdaki bir sonraki karakteri **peek()**'i kullanarak onu akımdan silmeden elde edebilirsiniz. Bu fonksiyon giriş akım sınıflarının üyesidir ve prototipi şudur:

```
int peek();
```

Akımdan bir sonraki karakteri döndürür. Dosyanın sonu geldiğinde de EOF döndürür.

putback()'i kullanarak bir akımdan gelen son karakteri bu akıma döndürebilirsiniz. Bu fonksiyon giriş akım sınıflarının üyesidir. Prototipi şudur:

```
istream &putback(char c);
```

Burada *c* okunan son karakterdir.

Çıkış gerçekleştiğinde veri, akıma bağlı olan fiziksel aygıta hemen yazılır. Bunun yerine bilgi dahili bir tamponda tampon dolana kadar saklanır. Ancak bundan sonra tamponun içeriği diske yazılır. Fakat, **flush()** fonksiyonunu çağırarak tampon bellek dolmadan önce bilginin fiziksel olarak diske yazılmasını sağlayabilirsiniz. **flush()** çıkış akımı sınıflarının bir üyesidir ve prototipi şudur:

```
ostream &flush();
```

flush()'a yapılan çağrılar, program uygunsuz ortamlarda (elektriğin sık sık gidip geldiği durumlar örneğin) çalıştırılacağı zaman garantilenebilir.

Örnekler

1. Bildiğiniz gibi katarı okumak için **>>** kullandığınızda, boşluk karakterini ilk görüdüğü yerde okumayı durdurur. Bu karakter gereksiz diye onun, bir katarı okurken boş karakterleri dikkate almasını engeller. Ancak bu problemin üstesinden **getline()** fonksiyonu kullanarak gelebilirsiniz. Aşağıdaki örneği inceleyelim:

```
// boşluk içeren bir katarı okumak için getline() kullanır
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char str[80];

    cout << "Enter your name: ";
    cin.getline(str, 79);

    cout << str << '\n';

    return 0;
}
```

Bu programda, **getline()** tarafından kullanılan sabitleyici yeni komut satırıdır. Bu **getline()**'in **gets()** fonksiyonu gibi davranışını sağlar.

2. Gerçek uygulamalarda, hangi tip verinin giriş olarak yapıldığını bilmediğiniz durumları daha kolay yönetebilmeksize imkan veren **peek()** ve **putback()**, bu özelliklerinden dolayı çok kullanışlıdır. Birazdan vereceğim örnek bu özelliğini kullanacak... Program bir dosyadan ya tamsayı değerlerini ya da katar değerlerini okuyabiliyor. Katarlar ve tamsayılar karışık sırada olabilir.

```
// peek() Örneği
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main()
{
    char ch;
    ofstream out("test", ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    char str[80], *p;

    out << 123 << "this is a test" << 23;
    out << "Hello there!" << 99 << "sdf" << endl;
    out.close();

    ifstream in("test", ios::in | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    do {
        p = str;
        ch = in.peek(); // Hangi tip karakterin bulunduğuunu sapla
        if(isdigit(ch)) {
            while(isdigit(*p=in.get())) p++; // tamsayıyı oku
            in.putback(*p); // akıma karakter döndür.
            *p = '\0'; // boş sonlu katar
            cout << "Integer: " << atoi(str);
        }
        else if(isalpha(ch)) { // katarı oku
            while(isalpha(*p=in.get())) p++;
            in.putback(*p); // akıma karakter döndür.
            *p = '\0'; // boş sonlu katar
            cout << "Katar: " << str;
        }
        else in.get(); // dikkate alma
        cout << '\n';
    } while(!in.eof());

    in.close();
    return 0;
}
```

Aliştırmalar

1. Örnek 1'deki programı yeniden düzenleyerek `getline()` yerine `get()` kullanımını sağlayın. Program farklı şekilde mi işledi?
2. Bir metin dosyasından, bir seferde tam satır okuyabilen ve ekranda görüntüleyebilen bir program yazın.
3. `flush()`'ı bazı durumlarda çağrımanın neden gerekli olabileceğini sakin kafayla düşünün.

9.5. Rasgele Erişim

C++'ın I/O sisteminde, rasgele erişimi, her biri giriş ve çıkış akımlarının birer üyesi olan `seekg()` ve `seekp()` fonksiyonlarıyla gerçekleştirirsiniz. En genel formları aşağıdaki gibidir.

```
istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
```

Buradaki `off_type`, `offset`'in sahip olabileceği değeri tutabilen `ios` tarafından tanımlanmış tamsayı tipidir. `seekdir` şu değerlere sahip olan `ios` tarafından tanımlanmış değer grubudur:

Değer	Anlamı
<code>ios::beg</code>	Baştan itibaren ara
<code>ios::cur</code>	Şu anki pozisyondan itibaren ara
<code>ios::end</code>	Sondan ara

C++'ın I/O sistemi dosyaya bağlı iki işaretçiyi yönetir. Birincisi bir sonraki giriş işleminin nerede gerçekleşeceğini belirleyen `get pointer`'dır. İkincisi ise bir sonraki çıkış işleminin nerede gerçekleşeceğini belirleyen `put pointer`'dır. Her giriş çıkış işlemi gerçekleştiğinde uygun işaretçi sırasıyla kendiliğinden artırılır. Ancak `seekg()` ve `seekp()` fonksiyonlarını kullanarak sırasız bir şekilde de dosyalara erişmek mümkündür.

`seekg()` fonksiyonu ilgili dosyanın `get pointer`'ını belirtilen `origin`'den itibaren `offset` kadar hareket ettirir. `seekp()` fonksiyonu ilgili dosyanın `put pointer`'ını belirtilen `origin`'den itibaren `offset` kadar hareket ettirir.

Genelde `seekg()` ve `seekp()` ile erişilen dosyalar ikili modunda açılmalıdır. Bu işlem karakter dönüştümlerinin dosyada uygun olmayan bir yere hareketin gerçekleşmesini önlüyor. Her dosya işaretçisinin yerini aşağıda verilen üye fonksiyonlarının yardımı ile bulursunuz.

```
pos_type tellg();
pos_type tellp();
```

Buradaki `pos_type`, `ios` tarafından tanımlanan, dosyanın pozisyonunu gösteren, en yüksek değeri tutabilecek bir tamsayı tipidir.

seekg() ve **seekp()**'nin, dosya işaretçilerini, **tellg()**'den **tellp()**'den dönen değerler tarafından belirlenen yere hareket ettiren, aşırı yüklenmiş halleri vardır. Prototipleri aşağıdaki gibidir:

```
istream &seekg(pos_type position);
ostream &seekp(pos_type position);
```

Örnekler

- Şimdiki program **seekp()** fonksiyonu ile ilgilidir. Dosyanın içerisindeki özel bir karakteri değiştirmenizi sağlar. Komut satırına dosya ismini, dosyada değiştirmek istediğiniz byte numarası ve yeni karakteri yazın. Dosyanın hem yazma hem de okuma işlemi için açıldığını unutmayın.

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

int main(int argc, char *argv[])
{
    if(argc!=4) {
        cout << "Usage: CHANGE <filename> <byte> <char>\n";
        return 1;
    }

    fstream out(argv[1], ios::in | ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open file.\n";
        return 1;
    }

    out.seekp(atoi(argv[2]), ios::beg);

    out.put(*argv[3]);
    out.close();

    return 0;
}
```

- Bir sonraki program **get pointer'ı** dosyanın ortasına getirmek için **seekg()**'yi kullanır ve dosyanın o noktadan sonraki içeriğini gösterir. Dosyanın ismi ve okumaya başlayacağı pozisyon komut satırında belirtilmelidir.

```
// seekg() örneği
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main(int argc, char *argv[])
{
    char ch;

    if(argc!=3) {
```

```

        cout << "Usage: LOCATE <filename> <loc>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    in.seekg(atoi(argv[2]), ios::beg);

    while(!in.eof()) {
        in.get(ch);
        cout << ch;
    }

    in.close();

    return 0;
}

```

Alıştırmalar

1. Metini tersten gösteren bir program yazın. İpucu: Bunu programı oluşturmaya başlamadan önce düşünün. Çözüm düşündüğünüzden daha basit olabilir.
2. Bir metin içerisindeki her karakter çiftinin yerlerini değiştiren bir program hazırlayın. Örneğin dosyada "1234" varsa program çalıştırınca sonra aynı bölgede "2143" olsun. (Dosyadaki, karakter sayısının çift olduğunu düşünün)

9.6. I/O Durumunu Kontrol Etme

C++ I/O sistemi, her I/O işleminden sonraki durum bilgisini saklar. Bir I/O akımının mevcut durumu **iostate** tipinden bir nesnede tanımlanmıştır. Bu da **ios** tarafından tanımlanan değer grubudur.

İsmi Anlamı

goodbit	Hata yok
eofbit	Dosya sonuna erişildi
failbit	Bir I/O hatası oluştu
badbit	Fatal bir I/O hatası oluştu

Eski derleyicilerde, I/O durum bayrakları **iostate** tipinden bir nesne yerine bir **int** içinde saklanırdu.

I/O durum bilgisini elde etmek için iki yolunuz var. Birincisi, **ios**'un üyesi olan **rdstate()** fonksiyonunu çağırabilirsiniz. Prototipi şu şekildedir:

```
iostate rdstate();
```

Hata bayraklarının mevcut durumunu gösterir. Az önceki bayrakların listesinden tahmin edeceğiniz gibi, **rdstate()** herhangi bir hata olmadığından **goodbit** döndürür. Aksi halde, herhangi bir hata döndürecektir.

Hata tespit etmenin diğer bir yolu da aşağıda verilen **ios** üyesi fonksiyonlardan bir veya birkaçını kullanmaktır.

```
bool bad();
bool eof();
bool fail();
bool good();
```

eof() fonksiyonunu daha önceden incelemiştiğimiz **bad()** fonksiyonu eğer **badbit** aktif ise **true** değeri döndürür. **good()** fonksiyonu da eğer hata yoksa **true** değeri döndürür. Aksi takdirde **false** döndürecektir.

Hata bir kere meydana geldiğinde, programınızın devam etmeden önce o hatanın temizlenmesi gereklidir. Bunu gerçekleştirmek için **ios** üye fonksiyonu olan **clear()**’ı kullanın. Prototipi aşağıdaki gibidir.

```
void clear(iostate flags = ios::goodbit);
```

Eğer *flags* **goodbit** (varsayılan hali) ise, bütün hata bayrakları temizlenir. Aksi takdirde *flags*’i istediğiniz değere ayarlayın.

Örnekler

- Şimdiki programımız **rdstate()** ile ilişkili... Program, bir metin dosyasının içeriğini gösterir. Eğer bir hata meydana gelirse fonksiyon **checkstatus()** kullanarak bildiriyor.

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

void checkstatus(ifstream &in);

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Usage: DISPLAY <filename>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    char c;
    while(in.get(c)) {
```

```

        cout << c;
        checkstatus(in);
    }

    checkstatus(in); //son durumu kontrol et
    in.close();

    return 0;
}

void checkstatus(ifstream &in)
{
    ios::iostate i;

    i = in.rdstate();

    if(i & ios::eofbit)
        cout << "EOF encountered\n";
    else if(i & ios::failbit)
        cout << "Non-Fatal I/O hata\n";
    else if(i & ios::badbit)
        cout << "Fatal I/O hata\n";
}

```

Onceki program en azından bir "error." gösterir. `while` döngüsü sona erdiğinde, `checkstatus()`'a yapılan çağrı bekleniği gibi EOF'un meydana geldiğini bildirir.

2. Bu program bir metin dosyasını görüntüler. Dosya hatasını tespit etmek için `good()` kullanır.

```

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

int main(int argc, char *argv[])
{
    char ch;
    if(argc!=2) {
        cout << "PR: <filename>\n";
        return 1;
    }
    ifstream in(argv[1]);
    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    while(!in.eof()) {
        in.get(ch);
        //hatayı kontrol et
        if(!in.good() && !in.eof())
            cout << "I/O Hata...terminating\n";
        return 1;
    }
    cout << ch;
}
in.close();
return 0;
}

```

Aliştırma

1. Bir önceki konudaki çözümlerinize hata denetimi ekleyin.

9.7. Özelleştirilmiş I/O ve Dosyalar

Bir önceki bölümde kendi sınıflarınıza göre insertion ve extraction operatörlerinin nasıl asırı yüklediğini öğrenmiştiniz. O bölümde sadece konsol I/O'su için tartışılmıştı. Ancak bütün C++ akımları aynı olduğundan örneğin aynı aşırı yüklenmiş inserter fonksiyonu, ekrana veya bir dosyaya çıkış vermek için de hiçbir değişiklik yapmadan kullanılabilir. Bu C++'ın I/O'ya en önemli ve kullanışlı yaklaşımlarından biridir.

Bir önceki bölümde belirttiğim gibi aşırı yüklenmiş inserter'lar ve extractor'lar hatta I/O manipülatörleri genel ifade ile yazıldığı sürece her akımla kullanılabilirler. Eğer özel kodlanmış bir akım I/O fonksiyonu içerisinde kullanırsanız, onun kullanımı tabii ki o akıma özel olacaktır. I/O fonksiyonlarınızı mümkün olduğunca genelleştirmeye yönelik meniz iste bu sebepten kaynaklanıyor.

Örnekler

1. Aşağıdaki programda, **coord** sınıfı << ve >> operatörlerini aşırı yükliyor. Hem ekrana hem de dosyaya yazmak için operatör fonksiyonlarını kullanabileceğinize dikkat edin.

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

class coord {
    int x, y;
public:
    coord(int i, int j) { x = i; y = j; }
    friend ostream &operator<<(ostream &stream, coord ob);
    friend istream &operator>>(istream &stream, coord &ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ' ' << ob.y << '\n';
    return stream;
}

istream &operator>>(istream &stream, coord &ob)
{
    stream >> ob.x >> ob.y;
    return stream;
}

int main()
{
    coord o1(1, 2), o2(3, 4);
```

```

    ofstream out("test");

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    out << o1 << o2;

    out.close();

    ifstream in("test");

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    coord o3(0, 0), o4(0, 0);
    in >> o3 >> o4;

    cout << o3 << o4;

    in.close();
    return 0;
}

```

2. Bütün I/O manipülatörleri dosyalarla kullanılabilir. Örneğin bu bölümde daha önce verilen bir programın aşağıdaki yenilenmiş halinde ekrana yazan manipülatör aynı zamanda dosyaya yazıyor.

```

#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;

// Dikkat:
ostream &atn(ostream &stream)
{
    stream << "Attention: ";
    return stream;
}

// lütfen not edin:
ostream &note(ostream &stream)
{
    stream << "Please Note: ";
    return stream;
}

int main()
{
    ofstream out("test");

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    // ekrana yazar

```

```

cout << atm << "High voltage circuit\n";
cout << note << "Turn off all lights\n";

// dosyaya yazar
out << atm << "High voltage circuit\n";
out << note << "Turn off all lights\n";

out.close();

return 0;
}

```

Aliştırma

- Kendi başınıza önceki bölümdeki programları kurcalayın ve her birini bir dosya üzerinde deneyin.

Pekiştirme Testi

Bu noktada aşağıdaki alıştırmaları yapabiliyor ve soruları cevaplayabiliyor olmalısınız.

- Üç sekme çıkış verip, alanın genişliğini 20'ye ayarlayan bir çıkış manipülatörü oluşturun. Manipülatörünüzün işleyişini gösterin.
- Karakterleri okuyan fakat alfabetik olmayanları eleyen bir giriş manipülatörü oluşturun. İlk alfabetik karakter okunduğunda manipülatörün onu giriş akımına dönürtmesini ve geri gelmesini sağlayın. Bu manipülatöre **findalpha** adını verin.
- Bir text dosyasını kopyalayan program hazırlayıncı. İşlem sırasında karakterlerin büyülüklerini tersine çevirsin (Örnek a > A)
- Bir text dosyasını okuyan ve hangi harfin kaç adet bulunduğuunu belirleyen bir program hazırlayıncı.
- Eğer hala yapmadıysanız, alıştırma 3,4 ve üstü için çözümlerinizde eksiksiz hata detayı ekleyin.
- Hangi fonksiyon **get pointer**'ının ve hangi fonksiyon **put pointer**'ının pozisyonunu değiştirir?

Bütünleştirme Testi

Burada, bu bölümdeki konularla, önceki bölümlerde anlatılan konular arasında ilişki kurup kuramayacağınızı denetler.

- Aşağıdaki kod bir önceki bölümde verilmiş olan **inventory** sınıfının düzenlenmiş halidir. **store()** ve **retrieve()** fonksiyonlarını dolduran bir program hazırlayıncı. Daha sonra, disk üzerinde birkaç kayıttan oluşan küçük bir veri dosyası oluşturun ve rasgele I/O kullanarak kullanıcının kayıt numarasını vermek suretiyle istediği şeyin bilgisini görebilmesini sağlayın.

```
#include <iostream>
```

```
#include <iostream>
#include <cmath>
using namespace std;
#define SIZE 40

class inventory {
    char item[SIZE]; // name of item
    int onhve; // number on hve
    double cost; // cost of item
public:
    inventory(char *i, int o, double c)
    {
        strcpy(item, i);
        onhve = o;
        cost = c;
    }
    void store(fstream &stream);
    void retrieve(fstream &stream);
    friend ostream &operator<<(ostream &stream, inventory ob);
    friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
    stream << ob.item << ":" << ob.onhve;
    stream << " on hve at $" << ob.cost << '\n';
    return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
    cout << "Enter item name: ";
    stream >> ob.item;
    cout << "Enter number on hve: ";
    stream >> ob.onhve;
    cout << "Enter cost: ";
    stream >> ob.cost;

    return stream;
}
```

2. İlginç bir alıştırma olması için kendi başınıza karakterler için bir **stack** sınıfı oluşturun. Bu sınıf karakterleri bellekteki bir dizi yerine, bir disk dosyasına depolasın.

www.Gerogoku.com

BÖLÜM 10

Sanal Fonksiyonlar

Türetilmiş Sınıflara İşaretçiler

Sanal Fonksiyonlara Giriş

Sanal Fonksiyonlar Hakkında Ek Bilgi

Polimorfizmi Uygulamak

Bu bölüm C++'ın diğer önemli bir kavramını ele alıyor: Sanal fonksiyonlar... Sanal fonksiyonları bu derece önemli yapan şey onların çalışma zamanı polimorfizmi desteklemesi. Polimorfizm C++'da iki şekilde desteklenir. Birincisi, aşırı yüklenmiş operatörler ve fonksiyonlar üzerinden derleme zamanında desteklenmesi. İkincisi sanal fonksiyonlar üzerinden çalışma anında desteklenmesi. Birazdan göreceksiniz, çalışma anı polimorfizm size müthiş bir esneklik sağlar. Temel olarak sanal fonksiyonlar ve çalışma anı polimorfizm türetilmiş sınıflara olan işaretçilerdir. Bu sebeple, şimdiki bölüm bu tip işaretçilerin incelemesiyle başlıyor.

Gözden Geçirme Testi

Devam etmeden önce aşağıdaki soruları doğru cevaplayabiliyor ve alıştırmaları yapabiliyor olmalısınız.

1. Rakamların "E" harfi ile bilimsel notasyonda gösterilmesini sağlayan bir manipülatör hazırlayın.
2. Bir metin dosyasını kopyalayan program hazırlayın ve kopyalama işlemi sırasında bütün sekmeleri doğru sayıda boşluklara dönüştürün.
3. Komut satırında belirtilen kelimeyi, bir metin dosyası içerisinde arayan ve kaç defa bulduğunu belirten bir program hazırlayın. İpucu: Sağ solu boşluk olan her şeyi kelime kabul edin.
4. `put pointer'ini, out` denen akışta bağlı bir dosyada 234'üncü byte'ı gösterecek şekilde ayarlayan bir program hazırlayın.
5. C++'ın I/O sistemi hakkındaki durum bilgilerini hangi fonksiyonlar döndürüyor?
6. C++ I/O fonksiyonlarının, C benzeri I/O sisteminin yerine kullanılmasından kazanılacak bir avantajı belirtin.

10.1. Türetilmiş Sınıflara İşaretçiler

Bölüm 4'de C++'ın işaretçilerinden bir yere kadar bahsedilmişti. Ancak işaretçilerle ilgili özel bir kavramın belirtilmesi bu noktaya kadar ertelenmişti çünkü o kavram özellikle sanal fonksiyonlar ile ilişkili idi. Kısaca özellik şu: Taban(temel) sınıfı işaretçi olarak deklare edilen işaretçi aynı zamanda o tabandan türemiş herhangi bir sınıf da işaretçi olabilir. Örneğin, `base` ve ondan türeyen `derived` adında iki sınıf ele alalım. Bu durumda aşağıdaki deyim doğrudur:

```
base *p; // taban(temel) sınıf işaretçisi

base base_obj; // base tipinden nesne
derived derived_obj; // derived tipinden nesne

// p tabi ki taban nesneleri işaret edebilir.
p = &base_obj; // p taban nesnesine işaret eder.
```

```
// p hatasız bir şekilde türetilmiş nesnelere de işaretçi olabilir
p = &derived_obj; // p türetilmiş nesneye işaret ediyor.
```

Açıklamaların gösterdiği gibi bir taban işaretçisi o tabandan türeyen herhangi bir sınıfın nesnesine de işaret edebilir(yazım hatası vermez) Taban işaretçisi türetilmiş nesne için kullanılsa da, siz sadece, türetilmiş nesnenin, tabandan miras alınmış üyelerine erişebilirsiniz. Bu durum taban işaretçisinin sadece taban(temel) sınıf hakkında bilgisi olduğundan kaynaklanır. Türetilmiş sınıf tarafından eklenen üyeler hakkında hiçbir bilgisi yoktur.

Bir taban işaretçisinin türetilmiş nesneye işaret etmesi mümkün olsa da o nun tersi mümkün değildir. Türetilmiş nesnenin işaretçisi taban(temel) sınıfından bir nesneye erişmek için kullanılamaz. (Bu kısıtlamanın üstesinden gelmek için tip dönüşümü uygulanabilir fakat bu işlem tavsiye edilen bir yöntem değildir)

Önemli bir nokta: İşaretçi aritmetiği, işaretçinin, işaret ettiği veri tipi ile ilişkilidir. Bu yüzden, bir taban işaretçisinin türetilmiş bir nesneyi işaret etmesini sağlayıp değerini bir arttırırsamız, bir sonraki türetilmiş nesneyi göstermez. Bir sonraki taban nesnesini işaret edecektir (ne düşünüyorsa artık). Bu ayrıntıya dikkat edin.

Örnek

1. Bir taban işaretçisinin türetilmiş sınıfa erişmek için nasıl kullanıldığını gösteren bir program.

```
// Türetilmiş sınıfa işaretçi
#include <iostream>
using namespace std;

class base {
    int x;
public:
    void setx(int i) { x = i; }
    int getx() { return x; }
};

class derived : public base {
    int y;
public:
    void sety(int i) { y = i; }
    int gety() { return y; }
};

int main()
{
    base *p; // taban tipe işaretçi
    base b_obj; // tabandan bir nesne
    derived d_obj; // türetilmişten bir nesne

    //taban nesneye erişmek için p'yi kullan
    p = &b_obj;
    p->setx(10); // taban nesnesine eriş
    cout << "Base object x: " << p->getx() << '\n';

    //türetilmiş nesneye erişmek için p'yi kullan
```

```

p = &d_obj; // türetilmiş nesneyi göster
p->setx(99); // türetilmiş nesneye eriş

// y'yi ayarlamak için p kullanılamaz bu sebeple doğrudan yapın.
d_obj.sety(88);
cout << "Derived object x: " << p->getx() << '\n';
cout << "Derived object y: " << d_obj.gety() << '\n';

return 0;
}

```

Türetilmiş sınıfları gösteren işaretçilerin dışında örnekte gösterilen şekilde bir temel sınıf işaretçisi kullanmanın manası yoktur. Fakat bir sonraki kısımda türetilmiş nesneleri gösteren temel sınıf işaretçilerinin neden bu kadar önemli olduğunu göreceksiniz.

Aliştırmalar

1. Kendi başınıza önceki program üzerinde çalışın. Örneğin türetilmiş bir işaretçi deklare etmeyi ve onun, temel sınıf'a ait bir nesneye erişmesini sağlamayı deneyin.

10.2. Sanal Fonksiyonlara Giriş

Bir *sanal fonksiyon* temel sınıf içerisinde deklare edilmiş ve türetilmiş sınıf içerisinde tanımlanmış bir üye fonksiyondur. Sanal fonksiyon oluşturmak için fonksiyona ait bildirimin önüne **virtual** anahtar kelimesini koyun. Sanal fonksiyon içeren bir sınıf miras alındığında türetilmiş sınıf sanal fonksiyonu türetilmiş sınıf'a bağlı olarak yeniden tanımlar. Aslında, sanal fonksiyonlar, polimorfizmin altında yatan "bir arabirime birden fazla metot" felsefesini tamamlarlar. Temel sınıf içerisindeki sanal fonksiyon, kendisinin *arabirim formunu* tanımlar. Türetilmiş sınıf tarafından sanal fonksiyonun yeniden tanımlanması, onun özellikle türetilmiş sınıf'a bağlı olması nedeniyle kendi işlemini yerine getirir. Yani, tekrar tanımlanma özel *metot* oluşturur. Sanal fonksiyon türetilmiş sınıf tarafından tekrar tanımlanırken **virtual** anahtar kelimesine gerek yoktur.

Sanal fonksiyonlar diğer üye fonksiyonlarının çağrıldığı şekilde çağrılabılır. Fakat sanal fonksiyonu ilginç yapan ve çalışma anı polimorfizminin desteklemesini sağlayan şey, sanal fonksiyonun işaretçi üzerinden çağrıldığında meydana gelen şeydir. Bir önceki konudan bildiğiniz gibi temel sınıf işaretçisi, türetilmiş sınıf nesnesini işaret etmek için kullanılabilir. Temel sınıf işaretçisi, sanal fonksiyon içeren türetilmiş sınıfı işaret ettiğinde ve bu sanal fonksiyon bu işaretçi üzerinden çağrıldığında, C++, işaretçi tarafından *gösterilen nesne tipine* dayanarak, bu fonksiyonun hangi sürümünün çalıştırılacağına karar verir. Bu karar *çalışma sırasında* gerçekleştirilir. Başka bir açıdan düşünelim, sanal fonksiyonun hangi sürümünün çalıştırılacağına karar veren, çağrı meydana geldiğinde işaret edilen nesnenin tipidir. Sonuç olarak eğer sanal fonksiyon içeren temel sınıfından iki veya daha fazla sınıf türetildiyse, temel işaretçi tarafından farklı nesneler işaret edildiğinde, sanal fonksiyonun farklı sürümleri çalıştırılır. Bu şekilde çalışma anı polimorfizm elde edilmiş olur. Gerçekte sanal fonksiyon içeren sınıflara *polimorfik sınıf* olarak başvurulur.

Örnekler

1. Sanal fonksiyon kullanan kısa bir örnek:

```
// Sanal fonksiyon kullanan basit bir örnek.
#include <iostream>
using namespace std;

class base {
public:
    int i;
    base(int x) { i = x; }
    virtual void func()
    {
        cout << "Using base version of func(): "i
        cout << i << '\n';
    }
};

class derived1 : public base {
public:
    derived1(int x) : base(x) {}
    void func()
    {
        cout << "Using derived1's version of func(): ";
        cout << i*i << '\n';
    }
};

class derived2 : public base {
public:
    derived2(int x) : base(x) {}
    void func()
    {
        cout << "Using derived2's version of func(): ";
        cout << i+i << '\n';
    }
};

int main()
{
    base *p;
    base ob(10);
    derived1 d_ob1(10);
    derived2 d_ob2(10);

    p = &ob;
    p->func(); // base'in func()'ının kullanımı

    p = &d_ob1;
    p->func(); // derived1'in func()'ının kullanımı

    p = &d_ob2;
    p->func(); // derived2'nin func()'ının kullanımı

    return 0;
}
```

Bu program aşağıdaki çıktıyi verir:

```
Using base version of func(): 10
Using derived1's version of func(): 100
Using derived2's version of func(): 20
```

Türetilmiş sınıf içerisindeki sanal fonksiyonun yeniden tanımlanması ilk başta fonksiyonların aşırı yüklenmesi gibi gözükebilir. Ancak bu iki işlem tamamiyla birbirinden farklıdır. Birincisi, yeniden tanımlanmış sanal fonksiyon aynı tipte, aynı parametre sayısına sahip ve aynı döndürme tipinde olması gerekirkenten aşırı yüklenmiş fonksiyon, tip ve/veya parametre sayısı açısından farklı olmak zorundadır. (Gerçekte sanal fonksiyonu yeniden tanımlarken parametrenin tipini veya sayısını değiştirirseniz sanal yapısını kaybeder ve aşırı yüklenmiş hale gelir.) Üstelik sanal fonksiyonlar sınıfa üye olmak zorundadırlar. Bu durum aşırı yüklenmiş fonksiyonlar için geçerli değildir. Aynı zamanda destructor fonksiyonları sanal olabilirle, fakat constructor'lar olamazlar. Aşırı yüklenmiş fonksiyonlar ile tekrar tanımlanmış sanal fonksiyonların arasındaki farklardan dolayı *overriding* terimi sanal fonksiyonun yeniden tanımlanması için kullanılır.

Gördüğünüz gibi örnek program üç sınıf oluşturmaktadır. **base** sınıfı **func()** sanal fonksiyonunu tanımlar. Bu sınıfın özellikleri **derived1** ve **derived2** sınıfları tarafından kalıtımıla alınmıştır. Bu sınıfların her biri kendi bağımsız kodlarıyla **func()**'ı yeniden tanımlar. **main()**'in içerisinde **p** temel sınıf işaretçisi, **base**, **derived1** ve **derived2** tipinden nesneler ile deklare edilir. İlk önce **ob**'un (**base** tipinden bir nesne) adresi **p**'ye atanır. **func()**, **p** kullanılarak çağrıldığında **base**'deki sürüm çalıştırılır. Daha sonra **p**'ye **d_ob1**'in adresi atanır ve **func()** yeniden çağrılır. Hangi sanal fonksiyonun çağrılacağını işaret edilen nesnenin tipi belirlediğinden bu kez **derived1**'deki sürümün yeniden tanımlanmış sürümü çalıştırılır. Son olarak **p**'ye **d_ob2**'nin adresi atanır ve **func()** tekrar çağrılır. Bu kez **derived2** içerisinde tanımlanan **func()** sürümü işletilir.

Bir önceki örnektenden anlaşılması gereken nokta şudur: Taban sınıf tarafından erişildiğinde işaret edilen nesnenin tipi, yeniden tanımlanmış sanal fonksiyonun hangi sürümünün işletileceğini belirler. Bu karar çalışma amında verilir.

2. Sanal fonksiyonlar kalıtima göre hiyerarsiktir. Ayrıca türetilmiş sınıf sanal fonksiyonu yeniden tanımlamadığında temel sınıfta tanımlanan fonksiyon kullanılır. Örnek olarak bir önceki programın biraz değiştirilmiş halini inceleyelim:

```
// Sanal fonksiyonlar hiyerarsiktir.
#include <iostream>
using namespace std;

class base {
public:
    int i;
    base(int x) { i = x; }
    virtual void func()
    {
        cout << "Using base version of func(): ";
        cout << i << '\n';
    }
};

class derived1 : public base
{
public:
    derived1() { i = 100; }
    void func()
    {
        cout << "Using derived1's version of func(): ";
        cout << i << '\n';
    }
};

class derived2 : public base
{
public:
    derived2() { i = 20; }
    void func()
    {
        cout << "Using derived2's version of func(): ";
        cout << i << '\n';
    }
};
```

```

};

class derived1 : public base {
public:
    derived1(int x) : base(x) {}
    void func()
    {
        cout << "Using derived1's version of func(): ";
        cout << i*i << '\n';
    }
};

class derived2 : public base {
public:
    derived2(int x) : base(x) {}
    // derived2 func()'ı yeniden tanımlamıyor
};

int main()
{
    base *p;
    base ob(10);
    derived1 d_ob1(10);
    derived2 d_ob2(10);
    p = &ob;
    p->func(); // base'in func()'ının kullanımı
    p = &d_ob1;
    p->func(); // derived1'in func()'ının kullanımı
    p = &d_ob2;
    p->func(); // base'in func()'ının kullanımı

    return 0;
}

```

Program aşağıdaki çıktıyi verir:

```

Using base version of func(): 10
Using derived1's version of func(): 100
Using base version of func(): 10

```

Bu sürümde **derived2**, **func()**'ı yeniden tanımlamaz. **d_ob2**, **p**'ye atanıp **func()** yeniden çağrıldığında **base**'in sürümü kullanılır. Çünkü bu sınıf, sınıf hiyerarşisinde bir sonrakidir. Genelde türetilmiş sınıf sanal fonksiyonu yeniden tanımlamadığında temel sınıf sürümü kullanılır.

3. Bir sonraki örnek bir sanal fonksiyonun, çalışma anında oluşan rasgele olaylara nasıl cevap verebildiğini gösterir. Bu program, standart rasgele sayı üretici **rand()** tarafından döndürülen değere dayanarak **d_ob1** ve **d_ob2**'den birini seçer. Çalıştırılan **func()** sürümünün çalışma anında belirlendiğini unutmayın. (Gerçekten de derleme anında **func()**'ın hangi sürümünün çağrılacağını belirlemek mümkün değildir.)

```

/*
 * Bu örnek sanal fonksiyonların çalışma anında rastgele oluşan olaylara
 * cevap vermek için nasıl kullanılacağını gösterir.
 */

```

```

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

class base {
public:
    int i;
    base(int x) { i = x; }
    virtual void func()
    {
        cout << "Using base version of func(): ";
        cout << i << '\n';
    }
};

class derived1 : public base {
public:
    derived1(int x) : base(x) {}
    void func()
    {
        cout << "Using derived1's version of func(): ";
        cout << i*i << '\n';
    }
};

class derived2 : public base {
public:
    derived2(int x) : base(x) {}
    void func()
    {
        cout << "Using derived2's version of func(): ";
        cout << i+i << '\n';
    }
};

int main()
{
    base *p;
    derived1 d_ob1(10);
    derived2 d_ob2(10);
    int i, j;
    for(i=0; i<10; i++) {
        j = rand();
        if((j%2)) p = &d_ob1; // tekse d_ob1'yi kullan
        else p = &d_ob2; // çiftse d_ob2'yi kullan
        p->func(); // doğru fonksiyon çağrılıyor
    }
    return 0;
}

```

4. Burada sanal fonksiyonların kullanımına ilişkin daha pratik bir örnek veriyorum. Bu program bir şeklin iki boyutunu tutan **area** isminde sosyal bir temel sınıf oluşturur. Aynı zamanda **getarea()** adında sanal bir fonksiyon deklare eder. Öyle ki, fonksiyon türetilmiş sınıflar tarafından yeniden tanımlandığında türetilmiş sınıf tarafından tanımlanan şeklin tipinin alamını döndürür. Bu durumda temel sınıf içerisindeki **getarea()**'nın bildirimini arabirimin yapısını belirler. Esas iş onu kahtum yoluyla alan sınıflara düşer. Bu örnekte bir üçgenin ve dikdörtgenin alanı hesaplanmaktadır.

BÖLÜM 10: Sanal Fonksiyonlar

```

// Arabirimini tanımlamak için sanal fonksiyon kullanın.
#include <iostream>
using namespace std;

class area {
    double dim1, dim2; // şeklin boyutları
public:
    void setarea(double d1, double d2)
    {
        dim1 = d1;
        dim2 = d2;
    }
    void getdim(double &d1, double &d2)
    {
        d1 = dim1;
        d2 = dim2;
    }
    virtual double getarea()
    {
        cout << "You must override this function\n";
        return 0.0;
    }
};

class rectangle : public area {
public:
    double getarea()
    {
        double d1, d2;
        getdim(d1, d2);
        return d1 * d2;
    }
};

class triangle : public area {
public:
    double getarea()
    {
        double d1, d2;
        getdim(d1, d2);
        return 0.5 * d1 * d2;
    }
};

int main()
{
    area *p;
    rectangle r;
    triangle t;

    r.setarea(3.3, 4.5);
    t.setarea(4.0, 5.0);

    p = &r;
    cout << "Rectangle has area: " << p->getarea() << '\n';

    p = &t;
    cout << "Triangle has area: " << p->getarea() << '\n';

    return 0;
}

```

`area` içerisindeki `getarea()` tanımlaması sadece bir taşıyıcıdır ve gerçek bir işlevi yoktur. Çünkü `area` özel bir tip şekele bağlanmamıştır. `area` içerisinde `getarea()`'ya verilebilecek anlamlı bir tanımlama yoktur. Aslında, `getarea()`'nın kullanışlı olabilmesi için türetilmiş sınıf tarafından yeniden tanımlanması gereklidir. Bir sonraki kısımda işlemi gerçekleştirecek bir yol göreceksiniz:

Alıştırmalar

1. `num` adında temel sınıf oluşturan bir program hazırlayın. Bu sınıfın bir tamsayı değerini ve `shownum()` adında bir sanal fonksiyonu içermesine imkan verin. `num`'u temel alan `outhex` ve `outoct` adında iki adet türetilmiş sınıf oluşturun. Türetilmiş sınıfın `shownum()`'u yeniden tanımlamasını sağlayın. Böylece değeri sırasıyla, 16'lık ve 8'lik tabanda göstersin.
2. İki nokta arasındaki uzaklığını `double` cinsinden bulan, `dist` adında temel bir sınıf oluşturun. `dist` içerisinde `trav_time()` adında uzaklığını kat etmek için gereken zamanı hesaplayan sanal bir fonksiyon hazırlayın. (Mesafenin mil cinsinden, hızında saatte 60 mil olduğunu kabul edin.) `metric` adındaki türetilmiş sınıfta `trav_time`'ı yeniden tanımlayın. Böylece seyahat zamanını, mesafenin kilometre ve hızın saatte 100 km olarak verildiğini kabul ederek belirlesin.

10.3. Sanal Fonksiyonlar Hakkında Ek Bilgi

Önceki konuda bulunan örnek 4'ün gösterdiği gibi sanal fonksiyon temel sınıfta deklare edildiğinde, bazı zamanlar yapacağı işler anlamsız olabiliyor. Bu durumla sık sık karşılaşılır. Çünkü genellikle temel sınıfta kendi başına tam bir sınıfı tanımlamaz. Bunun yerine merkezde bir grup üye fonksiyonları ve değişkenleri tanımlar. Türetilmiş sınıflar ise onları tamamlarlar. Temel sınıf sanal fonksiyonunun gerçekleştirecek anlamlı bir işi yoksa türetilmiş sınıf bu fonksiyonu yeniden tanımlamalıdır. Bu olayı kesinleştirmek için C++ saf sanal fonksiyon kavramını destekler.

Saf sanal fonksiyonun temel sınıfta herhangi bir tanımı yoktur. Sadece fonksiyonun prototipi dahil edilmiştir. Saf sanal fonksiyon yapmak için aşağıdaki genel formu kullanın.

```
virtual type func-name(parameter-list) = 0;
```

Bu bildirimdeki anahtar nokta fonksiyonu 0'a eşitlenmektedir. Bu derleyiciye, temel sınıf ile ilişkili fonksiyonun kodunun bulunmadığını bildirir. Sanal fonksiyon saf yapıldığında her türetilmiş sınıfın bu fonksiyonu yeniden tanımlaması için zorlar. Eğer türetilmiş bir sınıf bunu yapmazsa derleme anı hata meydana gelir. Böylelikle sanal fonksiyonu saf yapmak, türetilmiş sınıfın o fonksiyonu yeniden tanımlamasını garantioler.

Bir sınıf en az bir adet saf sanal fonksiyon içeriyorsa o sınıf *soyut sınıf* olarak bilinir. Bir soyut sınıf gövdesi olmayan en az bir sanal fonksiyon içerdiginden teknik olarak, o sınıfın tip tanımlaması tamamlanmamıştır, ve hiç bir türetilmiş sınıf ondan türeyemez. Böylece,

soyut sınıflar sadece kalitimi gerçekleştirmek için vardır. Tek başlarına duramazlar. Şu da anlamak önemlidir ki buna rağmen soyut sınıflara işaretçi tanımlamanız mümkündür. Çalışma anı polimorfizmi gerçekleştirmek için kullanılır. (Aynı zamanda soyut sınıfa referans kullanmak da geçerlidir.)

Bir sanal fonksiyon miras alındığında yapısı ile beraber geçirilir. Yani bir türemiş sınıf, bir sanal fonksiyonu temel sınıfın kalitim ile alıyorsa, daha sonra o sınıf temel sınıf olur ve başka bir sınıf ondan türerse, sanal fonksiyon yeniden tanımlanabilir. Örneğin, eğer taban sınıf B, F() adında bir sanal fonksiyon içeriyor ve D1 B'den, D2 de D1'den türüyorsa hem D1 hem de D2, f()’ı kendi reel sınıfına bağlı olarak yeniden tanımlayabilir.

Örnekler

- Size bir önceki konudaki örnek 4 programının geliştirilmiş sürümünü ‘ü veriyorum. Bu sürümde `getarea()` fonksiyonu temel sınıfı saf olarak tanımlanıyor.

```
// Soyut sınıf oluşturur.
#include <iostream>
using namespace std;

class area {
    double dim1, dim2; // şeklin boyutları
public:
    void setarea(double d1, double d2)
    {
        dim1 = d1;
        dim2 = d2;
    }
    void getdim(double &d1, double &d2)
    {
        d1 = dim1;
        d2 = dim2;
    }
    virtual double getarea() = 0; // saf sanal fonksiyon
};

class rectangle : public area {
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return d1 * d2;
    }
};

class triangle : public area {
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return 0.5 * d1 * d2;
    }
};
```

```

    }

int main()
{
    area *p;
    rectangle r;
    triangle t;

    r.setarea(3.3, 4.5);
    t.setarea(4.0, 5.0);

    p = &r;
    cout << "Rectangle has area: " << p->getarea() << '\n';

    p = &t;
    cout << "Triangle has area: " << p->getarea() << '\n';

    return 0;
}

```

Artık `getarea()` saf halde ve her türemiş sınıfın onu yeniden tanımlayacağını garantiyor.

2. Aşağıdaki program saf sanal sınıfın miras alındığında sanal yapısını nasıl koruduğunu gösteriyor.

```

// Sanal fonksiyonlar kalıtılılığında sanal durumlarını muhafaza ederler.
#include <iostream>
using namespace std;

class base {
public:
    virtual void func()
    {
        cout << "Using base version of func()\n";
    }
};

class derived1 : public base {
public:
    void func()
    {
        cout << "Using derived1's version of func()\n";
    }
};

// Derived2 inherits derived1.
class derived2 : public derived1 {
public:
    void func()
    {
        cout << "Using derived2's version of func()\n";
    }
};

int main()
{
    base *p;

```

```

base ob;
derived1 d_ob1;
derived2 d_ob2;

p = &ob;
p->func(); // base'in func()'ını kullan

p = &d_ob1;
p->func(); // derived1'in func()'ını kullan

p = &d_ob2;
p->func(); // derived2'nin func()'ını kullan

return 0;
}

```

Bu programda **func()** sanal fonksiyonu, ilk olarak **derived1** tarafından kalıtım ile alınıyor ve kendisine göre yeniden tammlıyor. Daha sonra, **derived2**, **derived1**'den türetiliyor. **derived2**'de **func()** yine, yeniden tanımlanıyor.

Sanal fonksiyonlar hiyerarşik olduğundan, **d_ob2**'ye erişildiğinde eğer **derived2** **func()**'ı yeniden tanımlanmamışsa **derived1**'in **func()** fonksiyonu kullanılır. Eğer **derived1** ve **derived2** **func()**'ı yeniden tanımlamazsa, bütün referanslar temel içerisindeki tanımlanan bir yere yönlendirilir.

Aliştırmalar

1. Kendi başınıza iki örnek program ile çalışın. Özellikle Örnek 1'deki **area**'yı kullanarak bir nesne oluşturmaya çalışın ve hata mesajını gözleyin. Örnek 2'de **derived2**'deki **func()** tanımlamasını kaldırmayı deneyin. Bu durumda **derived1** içerisindeki sürümün kullanıldığını gösterin.
2. Soyut sınıf kullanılarak bir nesne neden oluşturulamaz?
3. Örnek 2'de **derived1** içerisindeki **func()**'ın sadece yeniden tanımlanma bölümünü kaldırırsınız ne olur? Program hala derlenebilip çalışabilir mi? Eğer öyleyse neden?

10.4. Polimorfizm Uygulama

Çalışma anı polimorfizmi gerçekleştirmek için, sanal fonksiyonların nasıl kullanılması gerektiğini artık biliyorsunuz. Onları nasıl ve neden kullanılması gerektiğini incelemenin zamanı geldi. Bu kitapta bir çok kez belirtildiği gibi polimorfizm, benzer (fakat teknik olarak farklı) durumlara genel bir arabirimin uygulanma işletmidir. Böylece, "bir arabirim çok metot" felsefesini tamamlar. Polimorfizm önemlidir çünkü karmaşık yapıları çok basitleştirebilir. İyi tasarılanmış tek bir arabirim farklı ama ilişkili olaylar için kullanılabilir ve karışıklık ortadan kalkar. Aslında, polimorfizm, benzer olayların mantıksal yakınlıklarının bir yerde toplanmasını sağlar. Böylece, programın anlaşılmasında ve saklanması kolaylaşır. İlişkili oylara genel bir arabirim üzerinden erişilirse hatırlamanız gereken şeyler azdır.

Genelde OOP'ye ve özellikle C++'a bağlanmış iki kavram var. Bunlar *early binding* (*erken bağlama*) ve *late binding* (*geç bağlama*)'dır. Bunların ne manaya geldiklerini bilmeniz önemlidir. "early binding" derleme anında bilinebilen olaylar olarak düşünülür. Özellikle derleme sırasında çözümlenen fonksiyon çağrıları bu kavrama uyar. "Normal" fonksiyonlar, aşırı yüklenmiş fonksiyonlar ayrıca sanal olmayan üye ve arkadaş fonksiyonlar erken sınır bütünlüğüne dahildir. Bu tip fonksiyonlar derlenirken çağrılmaları için gerekli olan bütün adres bilgileri derleme anında bilinir. "early binding"'in en büyük avantajı (ve geniş çapta kullanılmasının sebebi) çok yeterli ve verimli olmasıdır. Derleme zamanında fonksiyon sınırlarına yapılan çağrılar, fonksiyon çağrı tipleri içinde en hızlı olandır. En büyük dezavantaj ise esnekliği azaltmasıdır.

"Late binding" çalışma anında oluşan olaylar olarak düşünülür. Geç sınırlı fonksiyon çağrıları, çağrılan fonksiyonun program çalışmadan bilinmemeyen adresidir. C++'da bir sanal fonksiyon geç sınır nesnesidir. Sanal fonksiyona temel sınıf işaretçisi tarafından erişildiğinde, program, işaret edilen nesnenin tipini ve yeniden tanımlanmış fonksiyonlardan hangisini işletmesi gerektiğini, çalışma anında saptar. "Late binding"'in en büyük avantajı çalışma anında esneklik sağlamaasıdır. Programınız, uzun kodlara ihtiyaç duymadan rasgele oluşan oylara cevap verebilir. En büyük dezavantaj ise fonksiyon çağrısına daha fazla bilginin bağlanmasıdır. Bunun sonucunda bu tip çağrılar "early binding"e göre daha yavaş gerçekleşir.

Verim açısından düşündüğünüzde ne zaman "early binding" veya ne zaman "late binding" kullanacağınızı karar vermeniz gerekebilir.

Örnekler

- Size "bir arabirim çok metot" kavramını gösteren bir program veriyorum. Program tamsayı değerleri için soyut liste sınıfı oluşturuyor. Listeye olan arabirim, **store()** ve **retrieve()** adında iki saf sanal fonksiyon tarafından sağlanıyor. Bir değer depolamak için **store()** fonksiyonunu, değer almak için **retrieve()** fonksiyonunu çağrırsın. **list** temel sınıfı bu olaylar için herhangi bir mevcut metot tanımlamıyor. Bunun yerine her türetilmiş sınıf, hangi tipte listenin tutulacağını tanımlıyor. Programda, iki tip liste tamamlanıyor. Sıra ve yiğin. İki listede tamamen farklı işlene de her birine aynı arabirim kullanılarak erişiliyor. Bu programı dikkatlice incelemelisiniz.

```
// Sanal fonksiyonlar
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

class list {
public:
    list *head; // listenin başına işaretçi
    list *tail; // listenin sonuna işaretçi
    list *next; // bir sonraki elemata işaretçi
```

```
int num; // depolanacak değer

list() { head = tail = next = NULL; }
virtual void store(int i) = 0;
virtual int retrieve() = 0;
};

// kuyruk(sıra) tipi liste oluştur
class queue : public list {
public:
    void store(int i);
    int retrieve();
};

void queue::store(int i)
{
    list *item;

    item = new queue;
    if(!item) {
        cout << "Allocation error.\n";
        exit(1);
    }
    item->num = i;

    // listenin sonuna koy
    if(tail) tail->next = item;
    tail = item;
    item->next = NULL;
    if(!head) head = tail;
}

int queue::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "List empty.\n";
        return 0;
    }

    // listenin başından at
    i = head->num;
    p = head;
    head = head->next;
    delete p;

    return i;
}

// yiğin tipi liste oluştur
class stack : public list {
public:
    void store(int i);
    int retrieve();
};

void stack::store(int i)
{
    list *item;
```

```
item = new stack;
if(!item) {
    cout << "Allocation error.\n";
    exit(1);
}
item->num = i;

// yiğin benzeri işlem için listenin önüne koy
if(head) item->next = head;
head = item;
if(!tail) tail = head;
}

int stack::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "List empty.\n";
        return 0;
    }

    // listenin başından at
    i = head->num;
    p = head;
    head = head->next;
    delete p;

    return i;
}

int main()
{
    list *p;

    // sıra
    queue q_obj;
    p = &q_obj; // sıraya işaret et

    p->store(1);
    p->store(2);
    p->store(3);
    cout << "Queue: ";
    cout << p->retrieve();
    cout << p->retrieve();
    cout << p->retrieve();

    cout << '\n';

    // sınıf
    stack s_obj;
    p = &s_obj; // yiğine işaretçi

    p->store(1);
    p->store(2);
    p->store(3);

    cout << "Stack: ";
    cout << p->retrieve();
    cout << p->retrieve();
```

```

    cout << p->retrieve();
    cout << '\n';
    return 0;
}

```

2. Liste programındaki **main()** fonksiyonu liste sınıflarının yaptıklarını ayrıca işleyicilerini basitçe gösterdi. Ancak, çalışma anı polimorfizmin neden bu kadar güçlü olduğunu görmeye başlamak için öncekinin yerine bu **main()**'i kullanmayı deneyin.

```

int main()
{
    list *p;
    stack s_obj;
    queue q_obj;
    char ch;
    int i;
    for(i=0; i<10; i++) {
        cout << "Stack or Queue? (S/Q): ";
        cin >> ch;
        ch = tolower(ch);
        if(ch=='q') p = &q_obj;
        else p = &s_obj;
        p->store(i);
    }
    cout << "Enter T to terminate\n";
    for(;;) {
        cout << "Remove from Stack or Queue? (S/Q): ";
        cin >> ch;
        ch = tolower(ch);
        if(ch=='t') break;
        if(ch=='q') p = &q_obj;
        else p = &s_obj;
        cout << p->retrieve() << '\n';
    }
    cout << '\n';
    return 0;
}

```

Buradaki **main()**, sanal fonksiyonlar ve çalışma anı polimorfizm kullanarak, rasgele meydana gelen olayların nasıl ele alındığını gösteriyor. Program 0'dan 9'a kadar işleyen bir **for** döngüsünü işletiyor. Döngünün her adımında size, hangi tip listeye-yiğina veya sıraya değer koyacağınız soruluyor. Verdığınız cevaba göre p temel işaretçisi, doğru nesneyi işaret etmek için ayarlanıyor ve i'nin mevcut değeri saklanıyor. Döngü bir kere bittiğinde, hangi listeden değer çıkaracağınızı soran diğer döngü başlıyor. Burada da cevabınız hangi listenin kullanılacağını belirliyor.

Bu örnek deney amaçlı olsa da, buradan çalışma anı polimorfizmin, rasgele olaylara cevap vermek zorunda olan programları nasıl basitleştirdiğini görebilirsiniz. Örneğin, Windows işletim sistemi bir programla bağlantısını ona mesajlar göndererek sağlar. Rasgele birçok mesaj olduğundan, programınız kendisi ile ilişkili olanları almalı ve onlara cevap vermelidir. Bu mesajlara cevap vermenin bir yolu da sanal fonksiyonları kullanmaktır.

Aliştırmalar

1. Örnek 1'e başka tip bir liste ekleyin . bu sürümün sıralı bir listeyi tutmasını sağlayın. (artan sıradan) Bu listeye **sorted** ismini verin.
2. Kendi başınıza çeşitli tipteki problemlere çalışma amı polimorfizmin nasıl uygulanabileceğini düşünün.

Pekiştirme Testi

Bu noktada aşağıdaki alıştırmaları yapabiliyor ve sorulara cevap verebiliyor olmalısınız.

1. Sanal fonksiyon nedir?
2. Hangi tip fonksiyonlar sanal yapılmaz?
3. Çalışma amı polimorfizmi gerçekleştirmek için sanal fonksiyon nasıl yardımcı olur?
4. Saf sanal fonksiyon nedir?
5. Soyut sınıf nedir? Polimorfik sınıf nedir?
6. Aşağıdaki kod doğru mudur? Değilse neden?

```
class base {
public:
    virtual int f(int a) = 0;
    // ...
};

class derived : public base {
public:
    int f(int a, int b) { return a*b; }
    // ...
};
```

7. Sanal yapı miras alınır mı?
8. Kendi başınıza bu noktada sanal fonksiyonlarla çalışın. Bunlar önemli bir kavramlardır. Tekniklerde uzmanlaşmanız gereklidir.

Bütünleştirme Testi

Bu kısım önceki konularla burada anlatılanların nasıl birbirleriyle kaynaştığını kontrol eder.

1. Konu 10.4' teki liste örneğini geliştirin. + ve - operatörlerini aşırı yüklesin. +'yı eleman depolamak için - elemanı almak için kullanılsın.
2. Sanal fonksiyonların aşırı yüklenmiş fonksiyonlardan farkı nedir?
3. Kendi başınıza, kitapta daha önce verilen fonksiyon aşırı yükleme örneklerini bir daha inceleyin. Hangilerinin sanal fonksiyonlara dönüştürülebileceğini saptayın. Ayrıca, sanal fonksiyonun kendi programlama sorunlarınıza nasıl çözüm bulacağını düşünün.

BÖLÜM 11

Şablonlar ve Hata Denetimi

Soysal Fonksiyonlar

Soysal Sınıflar

Hata Denetimi

Hata Denetimi Hakkında Ek Bilgi

new Kaynaklı Hatalar Denetleme

Bu bölüm C++'ın iki yüksek seviyeli yeniliğini ele alacaktır: *Şablonlar* ve *Hata denetimi*. Hiçbiri orijinal C++ tanımaması olmasa da, her ikisi de bir kaç yıl önce Standart C++ tarafından tanımlanmıştır. Bütün modern C++ derleyicileri tarafından desteklenmektedir. Bu iki yenilik programlamada amaca ulaşmak için en önemli iki özelliği sağlar. Açık ve güvenli kodu oluşturma.

Şablonları kullanarak soysal fonksiyonları ve sınıfları oluşturmak mümkündür. Bir soysal fonksiyonda veya sınıfı işlenilen verinin tipi parametre olarak belirtilir. Bu özellik size bir fonksiyon veya sınıfın her veri tipi için ayrı ayrı sürümlerini yazma zorunluluğunu ortadan kaldırır. Böylece şablonlar size açık kod yazma imkanı verir. Soysal fonksiyon ve sınıfın her ikisi de bu bölümde tartışılacaktır.

Hata denetimi C++'ın alt sistemidir. Size çalışma anında yapısal ve kontrollü olarak hataları ele alma şansı tanır. C++'ın hata denetimi ile programınız hata meydana geldiğinde hatayı işleyen rutini kendiliğinden çalıştırır. Hata denetiminin temel avantajı, daha önce herhangi bir uzun programda elle hazırlanması gereken hata denetim kodlarının çoğunu otomatikleştirmesidir. Hata denetiminin doğru kullanımınız sizin güvenli kod yazmanızı yardımcı olur.

Gözden Geçirme Testi

Daha ileri gitmeden, aşağıdaki soruları doğru cevaplayabiliyor ve alıştırmaları doğru yapabiliyor olmalısınız.

1. Sanal fonksiyon nedir?
2. Saf sanal fonksiyon nedir? Eğer bir sınıf bildirimini saf sanal fonksiyonu içeriyorsa, bu sınıf'a ne ad verilir ve kullanımına ne tür sınırlamalar getirilir?
3. _____ fonksiyonlarının ve _____ sınıf işaretçilerinin kullanımıyla çalışma anı polimorfizm gerçekleştirilebilir.
4. Eğer, bir sınıf hiyerarşisinde, türetilmiş sınıf (saf-olmayan) sanal fonksiyonu düzenlemeyi kabul etmiyorsa, türetilmiş sınıf'a ait bir nesne o fonksiyonu çağrıduğunda ne olur?
5. Çalışma anı polimorfizmin temel avantajı nedir?

11.1. Soysal Fonksiyonlar

Bir soysal fonksiyon, çeşitli veri tiplerine uygulanacak bir grup işlem tanımları. Üstüne geçirilen parametreye bağlı olarak işleyen bir tip veriye sahiptir. Bu mekanizmayı kullanarak aynı genel yordam çok sayıda veri tipine uygulanabilir. Bildığınız gibi birçok algoritma hangi veri ile çalışmasına bakmaksızın aynı yapıyı sergiler. Örneğin *Quicksort* algoritması, tamsayılar da, float'lara da uygulansa aynıdır. Farklı olan sadece sıralanacak olan verinin tipidir. Soysal fonksiyon oluşturarak algoritmanın temelini veriden bağımsız olarak tanımlayabilirsiniz. Bu işlemi bir kez gerçekleştirdikten sonra derleyici program çalıştırıldıkta

sonra gelen veri tipine göre kodu kendi oluşturur. Başka bir deyişle, bir soysal fonksiyon oluşturduğunuzda, aslında kendi kendine aşırı yüklemeye yapan bir fonksiyon oluşturmuş oluyorsunuz.

Soysal fonksiyon template anahtar kelimesini kullanarak oluşturulur. *Template* kelimesinin normal anlamı C++'da kullanılan anahtar kelimeleri yansıtır. Fonksiyonun ne yapacağını tanımlayan, detayları derleyiciye bırakın bir şablonu oluşturmak için kullanılır. Şablon fonksiyonun genel formu aşağıda gösterilmiştir.

```
template <class Ttype> ret-type fonksiyon-ismi(parameter list)
{
    // fonksiyonun kendisi
}
```

Buradaki *Ttype* fonksiyon tarafından kullanılan veri tipi için taşıyıcı isimdir (placeholder). Bu isim fonksiyon içerisinde kullanılabilir. Fakat sadece bir taşıyıcıdır (placeholder). Derleyici bu taşıyıcıyı fonksiyonun yeni bir sürümünü oluşturma esnasında gerçek veri tipi ile değiştirir.

Şablon bildirimini tanımlamak için genelde **class** anahtar kelimesinin kullanılıyorsa da **typename** anahtar kelimesini de kullanabilirsiniz.

Örnekler

1. Aşağıdaki program kendisiyle gönderilen iki değişkenin değerlerini yer değiştiren soysal bir fonksiyon oluşturuyor. İki değişkenin içeriğini değiştirmek veri tipinden bağımsız olduğundan bu örnekte soysal fonksiyonu kullanmak iyi bir yöntemdir.

```
//Fonksiyon şablon örneği
#include <iostream>
using namespace std;
// Bu fonksiyon şablonu
template <class X> void swapargs(X a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
int main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;

    cout << "Original i, j: " << i << ' ' << j << endl;
    cout << "Original x, y: " << x << ' ' << y << endl;
    swapargs(i, j); // tamsayıları yer değiştir
    swapargs(x, y); // float'ları yer değiştir
    cout << "Swapped i, j: " << i << ' ' << j << endl;
    cout << "Swapped x, y: " << x << ' ' << y << endl;

    return 0;
}
```

template anahtar kelimesi soysal fonksiyonu oluşturmak için kullanıldı.

```
template <class X> void swapargs(X &a, X &b)
```

satırı derleyiciye iki şey gösterir. İlk olarak şablonun oluşturulduğunu daha sonra soysal tanımlamanın başladığını belirtir. Buradaki **X** taşıyıcı olarak kullanılan soysal tiptir. **template** kısmından sonra, değiştirilecek olan değerlerin veri tipi olarak **X**'i kullanan **swapargs()** fonksiyonu bildirilmiştir. **main()**'de **swapargs()** fonksiyonu iki farklı veri tipi kullanılarak çağrılmıştır: Tamsayı ve float için. Çünkü **swapargs()** soysal bir fonksiyondur. Derleyici **swapargs()**'in iki farklı sürümünü kendiliğinden oluşturur.- Biri tamsayı değerlerini değiştirmek için, diğerı kayan nokta değişkenlerini değiştirmek için... Bu programı şimdi derleyip hemen çalıştmalısınız.

Şablonlardan bahsederken bazen kullanacağımız ve diğer C++ kaynaklarında karşılaşabileceğiniz birkaç terim vardır. Örneğin soysal fonksiyon, (yani **template** bilirimiyle tanımlanan fonksiyon) template fonksiyon olarak da isimlendirilebilir. Derleyici bu fonksiyonun özel bir sürümünü oluşturduğunda, oluşturulan fonksiyona "*generated function (üretilmiş fonksiyon)*" denir. Oluşturma işlemine ise "*instantiating*" denir. Başka bir deyişle üretilmiş fonksiyon şablon fonksiyonun özel bir sürümür.

2. Soysal fonksiyon tanımlamasındaki **template** kısmı fonksiyon ismi ile aynı satırda olmak zorunda değildir. Örneğin, aşağıdaki **swapargs()** fonksiyonunun yeni söz dizimi de bilinen bir formdur.

```
template <class X>
void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

Eğer bu formu kullanırsanız, **template** satırı ile soysal fonksiyon tanımının başladığı satır arasına başka bir şey giremez. Örneğin aşağıdaki kod parçası derlenmez...

```
// bu kod derlenmez
template <class X>
int i; // bu hatadır
void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

Açıklamaların gösterdiği gibi, **template** bildirimi geri kalan fonksiyon tanımlamasından hemen önce gelmelidir.

3. Daha önce bahsetmiştık, **class** kelimesini kullanmak yerine şablon tanımlamasında soysal tipi belirtmek için **typename** anahtar kelimesini kullanabilirsiniz. Örnek olsun diye size **swapargs()** fonksiyonun diğer bildirim şeklini verdim.

```
// typename kullanımı
template <typename X> void swapargs(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}
```

typename anahtar kelimesi aynı zamanda şablon içerisinde bilinmeyen bir tipi belirtmek için kullanılabilir. Fakat bu kullanım şekli kitap içeriğinin dışındadır.

4. Virgül ile ayırarak **template** ile birden fazla farklı soysal veri tipi tanımlayabilirsiniz. Örneğin bu program iki soysal tip içeren bir soysal fonksiyon oluşturur.

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

template <class type1, class type2>

void myfunc(type1 x, type2 y)
{
    cout << x << ' ' << y << endl;
}

int main()
{
    myfunc(10, "hi");

    myfunc(0.23, 10L);

    return 0;
}
```

Bu örnekte derleyici, **myfunc()** fonksiyonun çeşitli sürümlerini oluşturduğunda **type1** ve **type2** taşıyıcı tiplerinin yerlerini sırasıyla **int**, **char***, **double** ve **long** ile değiştirdi.

HATIRLATMA Bir soysal fonksiyon oluşturduğunuzda, derleyiciye, fonksiyon çağrıldığında farklı birçok veri tipi için fonksiyonun değişik sürümlerini oluşturma imkanı vermiş oluyorsunuz.

5. Soysal fonksiyonlar aşırı yüklenmiş fonksiyonlara benzerler ancak bazı kısıtlamaları vardır. Fonksiyonlar aşırı yüklendiğinde her fonksiyonun merkezinde farklı işlevleri tanımlayabilirsiniz. Fakat soysal fonksiyon bunun aksine bütün sürümlerinde aynı işlevi gerçekleştirmek durumundadır. Örneğin, aşağıdaki aşırı yüklenmiş fonksiyonlar soysal fonksiyonlar tarafından değiştirilemezler çünkü aynı işi gerçekleştirmezler.

```

void outdata(int i)
{
    cout << i;
}

void outdata(double d)
{
    cout << setprecision(10) << setfill('#');
    cout << d;
    cout << setprecision(6) << setfill(' ');
}

```

6. Şablon fonksiyon kendi kendini gerektiği hallerde aşırı yüklese de birini belirli bir şekilde aşırı yükleyebilirsiniz. Eğer bir soysal fonksiyonu aşırı yüklerseniz, aşın yüklenmiş fonksiyon, özel sürüme bağlı olarak soysal fonksiyonu düzenler(veya "saklanır"). Örneğin örnek 1'in bu sürümünü inceleyin.

```

// Şablon fonksiyonun düzenlenmesi
#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}

// Burası swapargs()'in soysal sürümünü düzenler
void swapargs(int a, int b)
{
    cout << "this is inside swapargs(int,int)\n";
}

int main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;

    cout << "Original i, j: " << i << ' ' << j << endl;
    cout << "Original x, y: " << x << ' ' << y << endl;

    swapargs(i, j); // aşırı yüklenmiş swapargs()'i çağırır
    swapargs(x, y); // float'ları yer değiştirir.

    cout << "Swapped i, j: " << i << ' ' << j << endl;
    cout << "Swapped x, y: " << x << ' ' << y << endl;

    return 0;
}

```

Açıklamaların gösterdiği gibi **swapargs(i,j)** çağrıldığında, programda tanımlanmış **swapargs()**'ın aşırı yüklenmiş sürümü çağrılr. Bu sebeple derleyici **swapargs()** fonksiyonunun bu sürümünü oluşturmaz çünkü soysal fonksiyon açık aşırı yüklemeyele düzenlenmiştir.

Şablonu manuel olarak asırı yüklemec, bu örnekte gösterildiği gibi size, soysal fonksiyonun özel bir durumu işleyecek yeni sürümünü oluşturma imkanı tanır. Fakat, genelde değişik veri tipleri için bir fonksiyonun farklı sürümlerine ihtiyaç duyarsanız şablon yerine asırı yüklenmiş fonksiyonları kullanmalısınız.

Aliştırmalar

1. Eğer hala yapmadıysanız önceki örnekleri çalıştırın.
2. **min()** adında iki argümanından küçüğünü gösterecek soysal bir fonksiyon yazın. Örneğin **min(3,4)**, 3'ü, **min('a','c')**, a'yi döndürsün. Fonksiyonunu bir programla çalıştırın.
3. Şablon fonksiyon olmak için **find()** adında iyi bir gönüllü var. Bu fonksiyon diziyi nesne için arar. Uyan nesne varsa indeks numarasını, bir şey bulamazsa -1 değerini döndürür. Size **find()**'ın özel sürümü için bir prototip veriyorum. **find()**'ı soysal bir fonksiyona dönüştürün ve çözümünüzü bir program ile gösterin. (**size** parametresi dizideki eleman sayını belirtiyor)

```
int find(int object, int *list, int size)
{
    // ...
}
```

4. Kendi cümlelerinizle, soysal fonksiyonların neden kıymetli olduğunu ve oluşturduğunuz programların kaynak kodunu nasıl basitleştirebileceğini açıklayın.

11.2. Soysal Sınıflar

Soysal fonksiyonları tanımlamaya ek olarak soysal sınıflarda tanımlayabilirsiniz. Bunu yaptığınız zaman, kendisi tarafından kullanılan bütün algoritmaları tanımlayan bir sınıf oluşturursunuz. Fakat sınıfa nesneler oluşturulduğu zaman, işlenecek verinin gerçek tipi parametre olarak belirtilmelidir.

Eğer sınıf genelleştirilmiş bir mantık içeriyorsa soysal sınıf kullanmak kullanışlıdır. Örneğin, tamsayı topluluğuyla çalışan bir algoritma karakter topluluğu ile de çalışabilir. Aynı zamanda, posta adreslerinin bağlantılı listesini tutan aynı mekanizma aynı zamanda, otomatik kisim bilgisinin de bağlantılı listesini tutar. Soysal sınıfı kullanmakla bir sırayı, bağlantılı listeleri ve diğerlerini tutabilirsiniz. Derleyici, nesne oluşturulurken belirtilen tipe göre, kendiliğinden doğru tipte nesne üretir.

Soysal sınıfın genel bildirim şekli şu şekildedir:

```
template <class Ttype> class class-name {
```

Burada *Type*, sınıf uyarıldığında belirtilen taşıyıcı tip ismidir. Eğer gerekliyse, virgül kullanarak birden fazla soysal veri tipi oluşturabilirsiniz.

Bir kere soysal sınıfı oluşturduğunuzda, aşağıdaki genel formu kullanarak bu sınıfa özel bir erişim oluşturmuş olursunuz.

```
class-name <type> ob;
```

Soyal sınıfa ait üye fonksiyonlar da kendiliğinden soysal haldedir. **template** ile belirtilmek zorunda değildir. Buradaki *type* sınıfın üzerinde çalışacağı verinin tip ismidir.

Bölüm 14'te göreceğiniz gibi, C++, şablon sınıflar üzerine kurulmuş bir kütüphane içerir. Bu kütüphane "Standart Şablon Kütüphanesi" şeklinde bilinir(kısaltılmış STL'dir). En çok kullanılan algoritmalar ve veri yapılarının soysal sürümlerini içerir. Eğer STL'yi etkili bir şekilde kullanmak istiyorsanız, şablon sınıflarını ve notasyonlarını tam olarak kavramalısınız.

Örnekler

1. Bu program çok basit bir bağıntılı liste sınıfı oluşturur ve daha sonra sınıfın işleyişini, karakterleri depolayan bağıntılı bir liste oluşturarak gösterir.

```
// basit, soysal bağıntılı liste
#include <iostream>
using namespace std;

template <class data_t> class list {
    data_t data;
    list *next;
public:
    list(data_t d);
    void add(list *node) {node->next = this; next = 0; }
    list *getnext() { return next; }
    data_t getdata() { return data; }
};

template <class data_t> list<data_t>::list(data_t d)
{
    data = d;
    next = 0;
}

int main()
{
    list<char> start('a');
    list<char> *p, *last;
    int i;
    // liste oluştur
    last = &start;
    for(i=1; i<26; i++) {
        p = new list<char> ('a' + i);
        p->add(last);
        last = p;
    }
}
```

```

// listeyi takip et
p = &start;
while(p) {
    cout << p->getdata();
    p = p->getnext();
}

return 0;
}

```

Gördüğünüz gibi, soysal sınıfın bildirimi soysal fonksiyonun bildirimine benziyor. Liste tarafından saklanan verinin gerçek tipi sınıf bildiriminde soysaldır. Gerçek veri tipinin belirlenmesi için listeden bir nesne bildirilecek diye bir şey yoktur. Bu örnekte nesneler ve işaretçiler, liste veri tipinin **char** olması gerektiğini belirten **main()** içerisinde oluşturulmuştur. Şu bildirime dikkatlice bakın:

```
list<char> start('a');
```

İstenilen veri tipinin üçgen parantez içerisinde nasıl belirtildiğine dikkat edin.

Bu programa girip çalıştırmalısınız. Alfabenin karakterlerini depolayan ve gösteren bağlantılı bir liste oluşturur. Fakat **list** nesneleri oluşturulduğunda belirtilen verinin tipini basitçe değiştirerek liste tarafından depolanan verinin tipini değiştirebilirsiniz. Örneğin, şu bildirimi kullanarak tamsayıları saklayan başka bir nesne oluşturabilirsiniz.

```
list<int> int_start(1);
```

Kendi oluşturduğunuz veri tipleri için de **list**'i kullanabilirsiniz. Örneğin, eğer adres bilgisini saklamak istiyorsanız, şu yapıyı kullanın

```

struct addr {
    char name[40];
    char street[40];
    char city[30];
    char state[3];
    char zip[12];
}

```

Daha sonra **addr** tipinden nesneleri saklayan nesneleri üretmek için **list**'i kullanmak istiyorsanız aşağıdaki gibi bir bildirim oluşturun (**structvar**'ın geçerli bir **addr** yapısı sakladığını tahmin ediyoruz):

```
list<addr> obj(structvar);
```

2. İşte soysal sınıf ile ilgili bir örnek daha. İlk olarak Bölüm 1'de gösterilen **stack** sınıfı üzerinde yeniden çalışacağız. Fakat burada **stack** (yiğin), şablon sınıfın içerisinde oluşturulacak. Bu sebeple herhangi bir tip nesneyi saklamak için kullanılabilir. Bu örnekte hem karakter yiğini hem de kayan-nokta yiğini oluşturulmuştur.

```

// fonksiyon soysal sınıfı gösterir.
#include <iostream>
```

```
using namespace std;

#define SIZE 10

// Soysal yiğin sınıfı aç
template <class StackType> class stack {
    StackType stck[SIZE]; // stack'i tutar
    int tos; // stack'in en üst kısmının indeks'i

public:
    void init() { tos = 0; } // stack'i hazırlama
    void push(StackType ch); // nesneyi stack'e ata
    StackType pop(); // yiğindan nesneyi çek
};

// Bir nesne koy
template <class StackType>
void stack<StackType>:: push(StackType ob)
{
    if(tos==SIZE) {
        cout << "Stack is full.\n";
        return;
    }
    stck[tos] = ob;
    tos++;
}

// Nesneyi Çağırma
template <class StackType>
StackType stack<StackType>::pop()
{
    if(tos==0) {
        cout << "Stack is empty.\n";
        return 0; // boş yiğinde boşluk gönder.
    }
    tos--;
    return stck[tos];
}

int main()
{
    // Karakter yiğinlarının işleyişini gösterin:
    stack<char> s1, s2; // İki yiğin oluştur
    int i;

    // yiğini hazırlama
    s1.init();
    s2.init();

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
    for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

    // çift yiğinları gösterme işlemi
    stack<double> ds1, ds2; // İki yiğin oluşturur.
}
```

```

// yağınları ayarlama
ds1.init();
ds2.init();

ds1.push(1.1);
ds2.push(2.2);
ds1.push(3.3);
ds2.push(4.4);
ds1.push(5.5);
ds2.push(6.6);

for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";
for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop() << "\n";

return 0;
}

```

stack sınıfının (ve bir önceki **list** sınıfının) gösterdiği gibi, soysal fonksiyonlar ve sınıflar çok güçlü araçlardır. Size, birçok veri tipi ile kullanabilecek genel bir algoritma hazırlamana imkan vererek, programlama zamanınızı maksimuma çıkarır. Belli veri tipleri için çalışmasını gereken algoritmaları tek tek bildirme derdinden sizi kurtarır.

3. Bir şablon sınıf birden fazla soysal veri tipine sahip olabilir. Yapmanız gereken, sınıf tarafından ihtiyaç duyulacak veri tiplerini, virgülü liste kullanarak **template** bildiriminin içerisinde koymانızdır. Örneğin, aşağıdaki kısa örnek iki soysal veri tipi kullanan bir sınıf oluşturuyor.

```

/* Bu örnek sınıf bildirimini içerisinde iki soysal veri tipi kullanır */
#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
    Type1 i;
    Type2 j;
public:
    myclass(Type1 a, Type2 b) { i = a; j = b; }
    void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
    myclass<int, double> ob1(10, 0.23);
    myclass<char, char*> ob2('X', "This is a test");

    ob1.show(); // int ve double'i göster
    ob2.show(); // char, char '*' i göster

    return 0;
}

```

Bu program aşağıdaki çıktıyi verir:

```

10 0.23
X This is a test

```

Program iki tip nesne bildiriyor. **ob1** tamsayı ve **double** verisini, **ob2** ise karakter ve karakter işaretçisini kullanıyor. Her iki durumda da derleyici kendiliğinden, nesnelerin oluşturulduğu yolu sağlamak için, doğru verileri ve fonksiyonları üretir.

Aliştırmalar

- Eğer hala yapmadıysanız iki soysal sınıf örneğini derleyin ve çalıştırın. Değişik veri tiplerinden listeleri veya yiğinları bildirmeye çalışın.
- Soysal sıra sınıfını oluşturun ve gösterin.
- Yapıldırıcı fonksiyonu çağrıduğunda aşağıdaki görevleri yerine getiren, **input** adında soysal bir sınıf oluşturun.
 - Kullanıcıya bilgi girişi için uyarın
 - Kullanıcı tarafından girilen veriyi alın ve
 - Kullanıcıya verinin daha önceden belirlenmiş sınırlar içinde olup olmadığını bildirin

input tipindeki nesnenin şu şekilde bildirilmesi gereklidir:

```
input ob("prompt message", min-value, max-value)
```

Buradaki "prompt message" bilgi girişi için kullanıcıyı uyaran mesajdır. Kabul edilebilen maksimum ve minimum değerler *min-value* ve *max-value* tarafından belirtilir. (Not: Kullanıcı tarafından girilen veri tipi, *min-value* ve *max-value*'nın tipiyle aynı olacaktır.)

11.3. Hata Denetimi (İşleme)

C++, *hata işleme* denen yerleşik bir hata denetleme mekanizması içerir. Hata denetimini kullanarak çalışma anı hatalarını yönetebilir ve onlara cevap verebilirisiniz. C++'ın hata denetimi üç anahtar kelimeye dayanır. Bunlar **try**, **catch** ve **throw** kelimeleridir. Çoğu söz diziminde hata için denetlemek istenen program satırları **try** bloğu içerisinde alınır. Eğer **try** bloğu içerisinde bir yerde hata meydana gelirse atılır (**throw** kullanarak). Hata **catch** kullanılarak yakalanır ve işlenir. Aşağıdaki açıklamalarla bu durumları irdeleyeceğiz.

Belirtildiği gibi, hata atan herhangi bir komut satırı **try** bloğu içerisinde çalıştırılmalıdır. (**try** bloğu içerisinde çağrılan fonksiyonlar da hata atabilir) Herhangi bir hata hemen **try** bloğunun arkasından, **catch** ile yakalanmalıdır. **try** ve **catch**'ın genel formu aşağıda gösterilmiştir.

```
try {
    // try bloğu
}
catch (typeid arg) {
    // catch bloğu
}
```

```

    catch (type2 arg) {
        // catch bloğu
    }
    catch (type3 arg) {
        // catch bloğu
    }
    .
    .
    .
    catch (typeN arg) {
        // catch bloğu
    }
}

```

try bloğu programın hata için denetlemek istediğiniz kısmını içermelidir. Bu işlem bir fonksiyonun içerisindeki az sayıdaki komutu denetleyecek kadar özel de olabilir veya **main()**'den başlayacak kadar genel de olabilir. (Tüm programın denetlenmesini sağlar).

Bir hata üretildiğinde, o hata işlenmek üzere uygun **catch** komutu ile yakalanır. Uygun **try** ile kullanılan birden fazla **catch** olabilir. Kullanılan **catch** komutu hata tipi tarafından belirlenir. Yani eğer **catch** tarafından belirtilen veri tipi hatanın tipine uyuyorsa **catch** komutu işletilir. (Diğerleri geçilir) Bir hata yakalandığında **arg** onun değerini alır. Eğer hatanın kendisine erişmek istemiyorsanız, **catch** yapısı içerisinde sadece **type**'ı belirleyin - **arg** bir seçenektedir. Oluşturduğunuz sınıflar dahil her tip veri yakalanabilir.

throw komutunun genel formu aşağıda gösterilmiştir.

```
throw exception;
```

throw komutu uygun **try** bloğu içerisinde veya kodu blok çağrıları içerisinde olan herhangi bir fonksiyondan çalıştırılmalıdır (doğrudan veya dolaylı olarak). Atılan değer **exception**'dır.

Yakalayan **catch** komutu olmayan bir hata üretilirse, anormal bir program sonlandırması olabilir. Eğer derleyiciniz Standart C++ ile derliyorsa, işlenmemiş bir hatanın üretilmesi, standart kütüphane fonksiyonu olan **terminate()**'in çalıştırılmasına sebep olur. Normalde, **terminate()** programınızı sonlandırmak **abort()**'u çağırır. Ancak eğer isterseniz, kendi sonlandırma bloğunuza tanımlayabilirsiniz. Daha fazla bilgi için derleyicinizin kütüphane kaynaklarına bakmanız gerekebilir.

Örnekler

1. Aşağıda size C++'da hata denetiminin nasıl gerçekleştirildiğini gösteren basit bir program hazırladım.

```

// Basit bir hata denetim programı
#include <iostream>
using namespace std;

int main()
{
    cout << "start\n";
}

```

```

try { // try bloğuna başla
    cout << "Inside try block\n";
    throw 10; // bir hata atar
    cout << "This will not execute";
}
catch (int i) { // hatayı yakala
    cout << "Caught One! Number is: ";
    cout << i << "\n";
}

cout << "end";
return 0;
}

```

bu program aşağıdaki çıktıyi verir.

```

start
Inside try block
Caught One! Number is: 10
end

```

Programa dikkatlice bakın. Gördüğünüz gibi, üç komut içeren bir **try** bloğu var. Ayrıca tam sayı hmasını işleyen **catch(int i)** komutu mevcut. **try** bloğu içerisinde üç komuttan sadece ikisi işleyecektir. Bunlar sırasıyla **cout** ve **throw**'dur. Hata bir kere atıldığında kontrol **catch'e** devredilir ve **try** bloğu sonlandırılır. Yani **catch** çağrılmaz. Bunun yerine program işleyışı devredilir. (Yiğin bu olayı gerçekleştirmek için kendiliğinden reset'lenir) Böylelikle, **throw'u** takip eden **cout** komutu hiçbir zaman işletilmez.

catch komutu çalıştırıldıktan sonra program işleyışı **catch'i** takip eden komuta geçirilir. Fakat bazı durumlarda bir **catch** bloğunun işleyışı, **exit()**, **abort()** veya programın sonlanması sağlayan diğer bir fonksiyondan dolayı durdurulur. Çünkü hata işleme genellikle çok ciddi durumlarda kullanılır.

2. Belirtildiği gibi, hatanın tipi, **catch** komutu içerisinde belirtilen tiple uyumlu olmak zorundadır. Mesela önceki örnekte, eğer **catch** komutunun tipini değiştirirseniz, hata yakalanmaz ve anormal bir program sonlanması gerçekleşir. Bu değişim şu şekilde olabilir:

```

// Bu örnek çalışmayacak
#include <iostream>
using namespace std;

int main()
{
    cout << "start\n";

    try { // try bloğuna başla
        cout << "Inside try block\n";
        throw 10; // hata türet
        cout << "This will not execute";
    }
    catch (double i) { // int hatası için çalışmayacak

```

```

        cout << "Caught One! Number is: ";
        cout << i << "\n";
    }
    cout << "end";
}

return 0;
}

```

Bu program aşağıdaki çıktıyi verecektir çünkü tamsayı hatası **double catch** tarafından yakalanmamıştır.

```

start
Inside try block
Abnormal program termination

```

- Fonksiyon **try** bloğundan çağrılıyorsa, o fonksiyonun içerisindeki bir komut tarafından hata üretebilir. Örneğin bu geçerli bir yapıdır:

```

/* try bloğının dışından bir fonksiyondan hata atılıyor */
#include <iostream>
using namespace std;

void Xtest(int test)
{
    cout << "Inside Xtest, test is: " << test << "\n";
    if(test) throw test;
}
int main()
{
    cout << "start\n";

    try { // try bloğuna başla
        cout << "Inside try block\n";
        Xtest(0);
        Xtest(1);
        Xtest(2);
    }
    catch (int i) { // hata yakala
        cout << "Caught One! Number is: ";
        cout << i << "\n";
    }
    cout << "end";
}

return 0;
}

```

Bu program aşağıdaki çıktıyi verir:

```

start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught One! Number is: 1
end

```

- Bir **try**bloğu fonksiyona yerleştirilebilir. Bu durumda, fonksiyona her girildiğinde fonksiyona bağlı olan hat denetimi sıfırlanır. Mesela şu programı inceleyin bakalım.

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

// try/catch komutları main() dışında bir fonksiyon içerisinde de bulunabilir
void Xhandler(int test)
{
    try{
        if(test) throw test;
    }
    catch(int i) {
        cout << "Caught One! Ex. #: " << i << '\n';
    }
}

int main()
{
    cout << "start\n";
    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout << "end";
    return 0;
}
```

Bu program aşağıdaki çıktıyi gösterir:

```
start
Caught One! Ex. #: 1
Caught One! Ex. #: 2
Caught One! Ex. #: 3
end
```

Gördüğünüz gibi, üç hata atılmıştır. Her hatada, fonksiyon döner. Fonksiyon yeniden çağrıldığında hata denetimi sıfırlanır.

5. Daha önce bahsi geçmiştik. Bir **try**'la ilişkilendirilmiş birden fazla **catch**'e sahip olabilirsiniz. Aslında, sıkça yapılan bir şeydir. Fakat her **catch** farklı tipte hataları yakalarnmalıdır. Örneğin aşağıdaki program hem tamsayıları hem de katarları yakalıyor.

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

// Değişik tipte hatalar yakalanabilir
void Xhandler(int test)
{
    try{
        if(test) throw test;
        else throw "Value is zero";
    }
    catch(int i) {
        cout << "Caught One! Ex. #: " << i << '\n';
    }
    catch(char *str) {
        cout << "Caught a string: ";
```

```

        cout << str << '\n';
    }

int main()
{
    cout << "start\n";

    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout << "end";
    return 0;
}

```

Bu program aşağıdaki çıktıyi verir.

```

start
Caught One! Ex. #: 1
Caught One! Ex. #: 2
Caught a string: Value is zero
Caught One! Ex. #: 3
end

```

Gördüğünüz gibi, her **catch** komutu kendi tipine cevap veriyor.

Genellikle **catch** bildirimleri, programda oluşma sıralarına göre kontrol edilir. Sadece karşılaştırma işlemi gerçekleştirtilir. Diğer bütün **catch** blokları göz ardı edilir.

Aliştırmalar

1. C++'ın hata denetimini anlayabilmenin en iyi yolu onunla ilgilenmektir. Önceki programlara girin, derleyin ve çalıştırın. Daha sonra onlarla üzerinde kötü emellerinizi uygulayın. Deneyler yapın. Parçalara ayırip sonuçları gözleyin.
2. Bu kod parçasındaki yanlış nedir?

```

int main()
{
    throw 12.23;
}

```

3. Bu kod parçasındaki yanlış nedir?

```

try {
    // ...
    throw 'a';
    // ...
}
catch(char *) {
    // ...
}

```

4. Uygun bir **catch** komutu olmadığı halde bir hata üretilirse ne olur?

11.4. Hata Denetimleri Hakkında Ek Bilgi

C++'da hata denetimleriyle ilgili, onları biraz daha kullanışlı ve değişik şekilde kullanmak için bazı ek yenilikler mevcuttur.

Bazı durumlarda bir hata işleyicisinin sadece bir tip yerine bütün tipler için çalışmasını isteyebilirsiniz. Bu olayı gerçekleştirmek çok basittir. `catch`'ı aşağıda belirtilen formda kullanmanız yeterlidir.

```
catch(...) {
    // Bütün hataları işleyin
}
```

İşte bütün veri tiplerine uyan biçimini verdim. Bir fonksiyonun, onu çağırana göndereceği hataları sınırlayabilirsiniz. Başka bir deyişle, hangi tipte hataların dışarı atılacağını kontrol edebilirsiniz. Aslında bir fonksiyonun hiçbir hatayı dışarı çıkarmamasını da sağlayabilirsiniz. Bu kısıtlamaları yapmak için, fonksiyon tanımına bir `throw` komutunu eklemeniz gereklidir. Bu yapının genel formu şöyledir:

```
return-type func-name(arg-list) throw (type-list)
{
    // ...
}
```

Burada sadece virgülerle ayrılmış olan `type-list`'deki (tip listesi) veri tiplerinin hataları fonksiyon tarafından atılabilir. Başka tipte bir hatayı atmak anormal program sonlandırmamasına sebep olur. Eğer bir fonksiyonun herhangi bir hata atabilmesini istemiyorsanız, boş bir liste kullanın.

Eğer derleyiciniz Standart C++ ile derliyorsa, fonksiyon izin verilmeyen bir hatayı attığında standart kütüphane fonksiyonu olan `unexpected()` çağrıılır. Normalde, bu işlem anormal program sonlanması yol açan `terminated()` fonksiyonunun çağrılmamasına neden olur. Ancak eğer istiyorsanız kendi sonlandırma işleyicinizi tanımlayabilirsiniz. Bunun gerçekleştirilmesi için derleyicinizin dokümanlarına başvurmanız gerekebilir.

Eğer bir geçersiz durumu bir hata işleyicisinden yeniden atmak istiyorsanız, bu işlemi kendi başına `throw`'u hata olamadan çağrıarak gerçekleştirebilirsiniz. Bu olay böylece mevcut geçersiz durumun harici `try/catch` sürecine sokulmasına sebep olur.

Örnekler

- Aşağıdaki program şu `catch(...)` durumunu gösteriyor:

```
// Bu program bütün hataları yakalar
#include <iostream>
using namespace std;
void Xhandler(int test)
{
    try{
        if(test==0) throw test; // int'i at
        if(test=='a') throw 'a'; // char'i at
    }
}
```

```

        if(test==2) throw 123.23; // double'i at
    }
    catch(...) { // bütün hataları yakala
        cout << "Caught One!\n";
    }

int main()
{
    cout << "start\n";
    Xhandler(0);
    Xhandler(1);
    Xhandler(2);
    cout << "end";
    return 0;
}

```

Bu program aşağıdaki çıktıyi verir.

```

start
Caught One!
Caught One!
Caught One!
end

```

Gördüğünüz gibi 3 **throw** da sadece bir **catch** komutuyla yakalanmıştır.

2. **catch(...)**'in kullanımı, bir grup **catch**'in yapabileceğini yapmasıdır. Bu kapasitede çalıştığında kullanışlı bir standart veya "hepsini yakala" olayını sağlar. Örneğin aşağıda verilen program yukarıdakinin değişik bir sürümüdür. Bu yapı tam sayı hatalarını yakalar, ancak diğerlerini yakalamak için **catch(...)** formunu kullanmıştır.

```

// bu örnek normalde catch(...)'yı kullanır
#include <iostream>
using namespace std;

void Xhandler(int test)
{
    try{
        if(test==0) throw test; // int'i at
        if(test==1) throw 'a'; // char'i at
        if(test==2) throw 123.23; // double'i at
    }
    catch(int i) { // int hmasını yakala
        cout << "Caught " << i << '\n';
    }
    catch(...) { // diğer bütün hataları yakala
        cout << "Caught One!\n";
    }
}

int main()
{
    cout << "start\n";
    Xhandler(0);
    Xhandler(1);
    Xhandler(2);
}

```

```
    cout << "end";
    return 0;
}
```

Bu programın çıktısı aşağıdaki gibidir.

```
start
Caught 0
Caught One!
Caught One!
end
```

Programın gösterdiği gibi, normalde **catch(...)** kullanmak, açıkça ele almak istediğiniz bütün hataları denetlemek için iyi bir yol. Ayrıca, bütün hataları yakalamakla, işlenmemiş bir hatanın anormal program beklenmedik şekilde programı sonlandırmasını engellemiştir.

3. Şimdiki programımız bir fonksiyondan atılan hata tiplerini nasıl kısıtlayabileceğimizi gösteriyor:

```
// atılan tipleri sınırlama
#include <iostream>
using namespace std;
// Bu fonksiyon sadece int, char, ve double'ları atabilir
void Xhandler(int test) throw(int, char, double)
{
    if(test==0) throw test; // int'i at
    if(test==1) throw 'a'; // char'i at
    if(test==2) throw 123.23; // double'i at
}

int main()
{
    cout << "start\n";
    try{
        Xhandler(0); // also, try passing 1 and 2 to Xhandler()
    }
    catch(int i){
        cout << "Caught int\n";
    }
    catch(char c){
        cout << "Caught char\n";
    }
    catch(double d){
        cout << "Caught double\n";
    }

    cout << "end";
    return 0;
}
```

Bu programda **Xhandler()** fonksiyonu sadece tam sayı, karakter ve double hatalarını atabilir. Eğer başka tip bir hata atmaya çalışırsa anormal program sonlandırma olayı meydana gelir (Bu da **unexpected()**'in çağrılması demektir). Bunun bir örneğini görmek istiyorsanız listeden **int**'i çıkartın ve programı yeniden deneyin.

Bir fonksiyon sadece, onu çağrıran **try** bloğuna geri attığı hata tipleri için kısıtlanabilir. Yani fonksiyondaki **try** bloğu ancak o fonksiyon içerisinde yakalanacak ise, bütün hata tiplerini atabilir. Kısıtlama anacak fonksiyon dışına bir hata atıldığında uygulanabilir.

4. **Xhandler()**'a yapılan değişiklik onun herhangi bir hata atmasını engelliyor.

```
// Bu fonksiyon HİÇBİR hata atamaz.
void Xhandler(int test) throw()
{
    /* Bu komutlar artık işleymez.
       Onun yerine anormal program sonlandırmamasına sebep olurlar
    */
    if(test==0) throw test;
    if(test==1) throw 'a';
    if(test==2) throw 123.23;
}
```

5. Daha önce öğrenmişiniz, bir hatayı yeniden atabilirsiniz. Bunu işlemi yapmanın en uygun nedeni birden çok işleyicinin hataya crışmesidir. Örneğin bir hata işleyicisi belki bir tip hatayı ele alır ve ikinci işleyici diğerinden farklı bir iş gerçekleştirir. Bir hata yeniden, ancak bir **catch** bloğundan(veya bloktan çağrılan herhangi bir fonksiyondan) atılabilir. Bir hatayı yeniden attığınızda aynı **catch** tarafından yeniden yakalanmaz. Harici bir **catch**'e yönlenir. Aşağıdaki program yeniden bir hata atma işlemini gösteriyor. Program **char *** hatasını yeniden atıyor.

```
// yeniden hata atmanın bir örneği
#include <iostream>
using namespace std;

void Xhandler()
{
    try {
        throw "hello"; // char *'ı at
    }
    catch(char *) { // char *'ı yakala
        cout << "Caught char * inside Xhandler\n";
        throw ; // char *'ı fonksiyon dışına yeniden at
    }
}

int main()
{
    cout << "start\n";
    try {
        Xhandler();
    }
    catch(char *) {
        cout << "Caught char * inside main\n";
    }
    cout << "end";
    return 0;
}
```

Bu program aşağıdaki çıktıyi verir.

```

start
Caught char * inside Xhandles
Caught char * inside main
end

```

Alıştırmalar

- Devam etmeden önce, bu losimdaki bütün örnekleri derleyin ve çalıştırın. Her programın çıkışlarını inceleyip neden öyle bir çıkış verdığını izleyin.
- Bu kod parçasındaki hata nedir?

```

try {
    // ...
    throw 10;
}
catch(int *p) {
    // ...
}

```

- Önceki kod parçasındaki hatayı düzeltmenin bir yolunu gösterin.
- Hangi **catch** yapısı bütün hata tiplerini yakalar?
- Size **divide()** adında bir fonksiyonun iskeletini veriyorum

```

double divide(double a, double b)
{
    // hata denetimi ekle
    return a/b;
}

```

Bu fonksiyon a/b 'nin sonucunu döndürüyor. C++'ın hata denetimini kullanarak fonksiyona hata kontrolünü ekleyin. Özellikle sıfıra bölmeye durumunu engelleyin. Fonksiyonu küçük bir program ile kullanın.

11.5. new Kaynaklı Hataları Denetleme

Bölüm 4'te, modern **new** tanımaması sayesinde, bir yer ayırma işleminde problem çıktığında **new**'in bir hata atacağını öğrenmiştiniz. Bölüm 4'ten bu yana hatalar hiç ele alınmadı. O tip bir hatanın nasıl işleneceğini daha sonra açıklayacağım demiştim. Şimdi **new**'in başarısızlığında ne yapılması gerektiğini inceleme zamanı geldi. Başlamadan önce C++ tarafından belirlenmiş **new**'in davranışlarını içeren bu kısımda kavramları iyi belirlememiz gerekecek. Bölüm 4'ten hatırlayacağınız gibi, C++'ın keşfinden sonra **new**'in başarısızlığı durumunda göstereceği tepki birkaç defa değiştirilmiştir. Özellikle C++ ilk çıktığında **new**, hata durumunda boşluk dolduruyordu. Daha sonraları bu olay hata üretmeye dönüştürüldü ve bu hatanın ismi zaman içerisinde değişti. Son olarak **new**'in başarısızlığının normalde bir hata üretmesine karar verildi. Fakat bir seçenek olarak boş bir işaretçi de gönderebilmesi sağlandı. Böylelikle **new**, derleyici üreticileri tarafından farklı zamanlarda farklı şekilde tanıtıldı. Coğu derleyici **new**'i Standart C++ ile paralel olarak tanımlasa da

tümü henüz bunu yapmamaktadır. Eğer burada verilen kod örnekleri sizin derleyicinizle çalışmazsa, **new**'in nasıl tanımladığı konusundaki ayrıntılar için derleyicinizin dokümanlarına bakın. Standart C++'da yer ayırmayı gerçekleştiremeyecek **new**, **bad_alloc** hatalara bakın. Standart C++'da yer ayırmayı gerçekleştiremeyecek **new**, **bad_alloc** hataları atar. Eğer bu hatayı yakalamazsanız programınız sonlandırılır. Bu davranış biçimi, kısa ve basit programlar için iyi bir davranış olsa da gerçek uygulamalarda bu hatayı yakalamalı ve uygun biçimde işlemelisiniz. Bu hataya erişmek için **<new>** başlığını programınıza dahil etmelisiniz.

NOT İlk zamanlar bu hataya **xalloc** deniyordu. Ben bu kitabı hazırlayıorken, bir çok derleyici hala bu ismi kullanıyordu. Fakat **bad_alloc**, Standart C++ tarafından tanımlanmış bir ismidir ve gelecekte de o kullanılacaktır.

Standart C++'da yer ayırmaya başarısızlığı meydana geldiğinde **new**'in hata üretmek yerine boşluk döndürmesi de mümkündür. **new**'in bu kullanım şekli en çok eski kodları modern C++ derleyicileri ile derlediğinizde kolaylık sağlar. Ayrıca **malloc()**'u **new** ile değiştirdiğinizde bu form faydalı olur. Bu kullanım şeklini aşağıda görebilirsiniz.

```
p_var = new (nothrow) type;
```

Burada *p_var*, *type*'ın işaretçi değişkenidir. **new**'in **nothrow**.biçimi, yıllar önceki **new** ile aynı şekilde çalışır. Başarsızlık durumunda boş döndürdüğünden eski kodun içerisinde alabilirsiniz ve ayrıca bir hata denetimi eklemek zorunda da kalmazsınız. Fakat yeni kodlarda hatalar daha iyi bir alternatif sunar.

Örnekler

- Yer ayırmaya başarısızlığını denetleyen bir **try/catch** bloğunu kullanan **new**'e ilişkin bir örnek hazırladım.

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

int main()
{
    int *p;
    try {
        p = new int; // int için bellekte yer ayırır
    } catch (bad_alloc xa) {
        cout << "Allocation failure.\n";
        return 1;
    }
    for(*p = 0; *p < 10; (*p)++)
        cout << *p << " ";
    delete p; // belleği serbest bırakır
    return 0;
}
```

Burada yer ayırmaya başarısızlığı oluşursa bu **catch** ifadesi tarafından yakalanır.

2. Normal durumlarda yukarıdaki programın başarısızlığa uğrasa da aşağıdaki program `new`'in yer ayırma başarısızlığını zorlayarak hata atma yeteneğini kullanacaktır. Bu-
nu, yer kalmayana kadar bellekte yer ayıracak gerçekleştirecektir.

```
// Yer ayırma başarısızlığına zorlama.
#include <iostream>
#include <new>
using namespace std;

int main()
{
    double *p;
    // bu, belleği tamamen dolduracaktır
    do {
        try {
            p = new double[100000];
        } catch (bad_alloc xa) {
            cout << "Allocation failure.\n";
            return 1;
        }
        cout << "Allocation OK.\n";
    } while(p);
    return 0;
}
```

3. Aşağıdaki program `new(nothrow)` alternatifinin nasıl kullanılacağını gösterir. Bir önceki program gibi çalışır ve yer ayırma başarısızlığı oluşturur.

```
// new(nothrow) alternatifi örneği.
#include <iostream>
#include <new>
using namespace std;

int main()
{
    double *p;

    // bu, belleği tamamen dolduracaktır
    do {
        p = new(nothrow) double[100000];
        if(p) cout << "Allocation OK.\n";
        else cout << "Allocation Error.\n";
    } while(p);

    return 0;
}
```

Bu programda gösterildiği gibi `nothrow` yaklaşımını kullandığınız zaman her yer ayırma isteminden sonra `new` tarafından döndürülen işaretiyi kontrol etmek zorundasınız.

Aliştırmalar

1. Bir yer ayırma başarısızlığı meydana geldiğinde `new` ile `new(nothrow)`'un davranışları arasındaki farkı açıklayın.

2. Size aşağıdaki kod veriliyor. Bunu modern C++ stili koda dönüştürmek için iki ayrı yol gösterin.

```
p = malloc(sizeof(int));
if(!p) {
    cout << "Allocation error.\n";
    exit(1);
}
```

PEKİŞTİRME TSETİ

Bu noktada aşağıdaki alıştırmaları yapabiliyor ve soruları cevaplayabiliyor olmalısınız.

1. Bir dizi değerin modunu döndüren bir soysal fonksiyon oluşturun. (Bir grubun *modu* orada en sık meydana gelebilecek değerdir.)
2. Bir dizi değerin toplamını döndüren soysal fonksiyon oluşturun.
3. Soysal bir "bubble sort" oluşturun (veya istediğiniz herhangi bir sıralama algoritması kullanın.)
4. Değişik tipte nesneleri çift olarak yığında saklayabilecek şekilde **stack** sınıfı üzerinde yeniden çalışın ve çözümünüzü gösterin.
5. **try**, **catch** ve **throw**'un genel formlarını gösterin. Kendi cümlelerinizle işlevlerini açıklayın.
6. Yığın aşımlarını hata olarak işleyecek şekilde **stack** sınıfı üzerinde tekrar çalışın.
7. Derleyicinizin dokümanlarına göz atın. **terminate()** ve **unexpected()** fonksiyonlarını destekleyip desteklemediğini inceleyin. Genelde bu fonksiyonlar, seçtiğiniz herhangi bir fonksiyonu çağıracak şekilde yapılandırılabilirler. Eğer derleyicinizde durum buyسا işlenmemiş hataları işleyen size özel bir grup fonksiyonu oluşturmayı deneyin.
8. Düşünce sorusu: **new**'in başarısızlık durumunda hata üremesinin, boş döndürmesinden neden daha iyi bir yaklaşım olduğunu açıklayın.

Bütünleştirme Testi

Bu kısım sizin geçmiş bölgülerdeki konular da dahil olmak üzere, bu bölümde geçen konuları ne kadar kavradığınızı kontrol edecektir.

1. Bölüm 6, Konu 6.7, Örnek 3'te güvenli dizi sınıfı gösterilmişti. Kendiniz bunu soysal güvenli diziye dönüştürün.
2. Bölüm 1'de **abs()** fonksiyonunun aşırı yüklenmiş fonksiyonları oluşturulmuştu. Daha iyi bir çözüm için kendiniz herhangi bir sayısal nesnenin mutlak değerini döndüren soysal **abs()** fonksiyonunu oluşturun.

www.Georgokaku.com

BÖLÜM 12

Çalışma Anı Tip Tanıma ve Dönüştürme Operatörleri

Çalışma Anı Tip Tanıma (RTTI)

`dynamic_cast`

`const_cast`, `reinterpret_cast`, `static_cast`

Bu bölümde son C++'a zamanlarda eklenen iki yenilikten bahsedeceğiz. Çalışma anı tip tanıma (kasaltılmış RTTI) ve yeni tip dönüştürme operatörleri. RTTI size programınız sırasında bir nesnenin tipini tamma imkanı verir. Dönüştürme operatörleri ise size daha güvenli ve kontrollü dönüştürme imkanı sağlar. Görüğünüz gibi dönüştürme operatörlerinden biri o'nun **dynamic_cast**, RTTI ile doğrudan ilişkilidir. Bu iki konuyu aynı bölüm içerisinde ele almamızı sürecek.

Gözden Geçirme Testi

Bu bölüm başlamadan önce aşağıdaki soruları doğru bir şekilde cevaplayabiliyor ve alışturmaları yapabiliyor olmalısınız.

1. Sosyal fonksiyon nedir ve genel biçimini nasıldır?
2. Sosyal sınıf nedir ve genel formu nasıldır?
3. Bir argümanının diğerine göre kuvvetini alıp sonucu döndüren **gexp()** adında sosyal bir fonksiyon yazın.
4. Bölüm 9, Konu 9.7, Örnek 1'de tam sayı koordinatları saklayan **coord** adında bir sınıf oluşturulmuş ve programda gösterilmiştir. **Coord** sınıfının, her tipten koordinatı saklayabilen sosyal sürümünü oluşturun. Çözümünü bir programda gösterin.
5. **try**, **catch** ve **throw**'un C++'m hata denetimini beraber çalışarak nasıl sağladığını kısaca açıklayın.
6. Program işleyisi **try** bloğu üzerinden geçmeden **throw** kullanılır mı?
7. **terminate()** ve **unexpected()** hangi amaçlara hizmet eder?
8. **catch**'in hangi tipi bütün hataları yakalar?

12.1. Çalışma Anı Tip Tanımayı (RTTI) Anlama

Çalışma anı tip bilgisi size yabancı gelebilir çünkü C gibi polimorfik olmayan dillerde bu özellik bulunmuyordu. Polimorfik olmayan bu dillerde, tiplerin çalışma zamanında bilinmesine gerek yoktu çünkü her nesnenin tipi derlenme zamanında belliydi (Ör: Programın yazıldığı sırada). Ancak C++ gibi polimorfik dillerde derleme aşamasında bir nesnenin tipinin bilinmediği durumlar olabilir çünkü program işletilene kadar nesnenin önceki yapısı belirlenmemiş olabilir. Bildiğiniz gibi C++ polimorfizmi sınıf hiyerarşisi kullanımı yoluyla, sanal fonksiyonlarla ve taban sınıf işaretçileriyle tanımlar. Bu yaklaşımda, taban sınıf işaretçisi, taban sınıfından veya *bu tabandan türemiş olan nesnelere*, işaret etmek için kullanılır. Bu sebeple belli bir anda, taban işaretçisi tarafından ne tip bir nesnenin işaret edildiğini bilmek, her zaman mümkün olmayabilir. Bu saptama çalışma anı tip belirleme ile mutlaka kod işlerken yapılmalıdır.

Bir nesnenin tipini elde etmek için **typeid**'yi kullanınız. **typeid**'yi kullanabilmek için **<typeinfo>** başlığını programa dahil etmelisiniz. **typeid**'nin en genel formunu aşağıda verilen şekildedir:

```
typeid(object)
```

Buradaki *object* (nesne) tipini öğrenmek istediğiniz nesnedir. **typeid**, *object* tarafından tanımlanan nesne tip bilgisini içeren, **type_info** tipinden ayrı bir nesneye başvuru döndürür. **type_info** sınıfı aşağıdaki yerel üyeleri içerir.

```
bool operator== (const type_info &ob);
bool operator!= (const type_info &ob);
bool before (const type_info &ob);
const char *name();
```

Aşırı yüklenmiş == ve != tiplerin karşılaştırılmasını sağlar. **before()** fonksiyonu, eğer uyarılan nesne karşılaştırma sırasının parametresi olarak kullanılan nesneden önce geliyorsa *true* değeri döndürür. (Bu fonksiyon genellikle arka planda kullanılır. Döndürdüğü değerle kalıtım veya sınıf hiyerarşisi açısından hiçbir şey yapılmaz.) **name()** fonksiyonu tip ismine bir işaretçi döndürür.

typeid her nesnenin tipini elde etse de, bunun en etkin kullanımı polimorfik taban sınıfının işaretçisi üzerinden gerçekleşir. Bu durumda kendiliğinden, taban sınıf veya o tabandan türemiş olabilecek gerçek nesnenin tipini döndürür. (Bir taban sınıf işaretçisinin, o taban sınıfından veya tabandan türemiş olan sınıfın bir nesneye işaret edebileceğini unutmayın) Bu sayede taban sınıf işaretçisi ile işaret edilen nesnenin tipini çalışma alanında belirleyebilirsiniz. Aynı sistem başvurulara da uygulanabilir. **typeid** polimorfik sınıfın bir nesnenin başvurusuna uygulandığı zaman, gerçekten başvurulan türemiş tipten olabilecek nesnenin tipini döndürür. **typeid** polimorfik olmayan bir sınıfa uygulandığında işaretçinin veya başvurunun taban tipi elde edilir.

typeid'nin tip ismini argüman olarak kabul eden ikinci bir yapısı vardır. Bu form aşağıda verilmiştir.

```
typeid(type-name)
```

typeid'nin bu şeklini kullanmadızın nedeni belirtilen tipin bilgisini içeren **type_object** nesnesini elde etmektir. Bu sayede bu nesne tip karşılaştırma ifadesinde kullanılabilir.

typeid genellikle yeniden başvurulan işaretçilere uygulandığından (Ör: * operatörleri uygulanan), yeniden başvurulan işaretçinin boş olma durumu için bir hata oluşturulmuştur. Bu durumda **typeid**, **bad_typeid** hatasını atar.

Çalışma zamanı tip tanıma her programın kullanacağı bir şey değildir. Ancak, polimorfik tiplerle çalışığınızda, verilen her durumda size hangi tipte nesne üzerinde çalıştığını bilgisini sunar.

Örnekler

1. Aşağıdaki program typeid'ının kullanımını gösteriyor. Program ilk önce C++'in kendi iç tiplerinden olan int'in tip bilgisini verecek, daha sonra BaseClass tipinden olan p işaretçisi tarafından işaret edilen nesnelerin tipini gösterecek.

```
// typeid'yi kullanan örnek
#include <iostream>
#include <typeinfo>
using namespace std;

class BaseClass {
    virtual void f() {} //BaseClass'ı polimorfik yap
    // ...
};

class Derived1: public BaseClass {
    // ...
};

class Derived2: public BaseClass {
    // ...
};

int main()
{
    int i;
    BaseClass *p, baseob;
    Derived1 ob1;
    Derived2 ob2;
    // İlk önce iç tiplerden birinin bilgisini göster
    cout << "typeid of i is ";
    cout << typeid(i).name() << endl;

    // Polimorfik tipler için typeid'yi göster
    p = &baseob;
    cout << "p is pointing to an object of type ";
    cout << typeid(*p).name() << endl;

    p = &ob1;
    cout << "p is pointing to an object of type ";
    cout << typeid(*p).name() << endl;

    p = &ob2;
    cout << "p is pointing to an object of type ";
    cout << typeid(*p).name() << endl;
    return 0;
}
```

Program tarafından üretilen çıktı aşağıda gösterilmiştir:

```
typeid of i is int
p is pointing to an object of type class BaseClass
p is pointing to an object of type class Derived1
p is pointing to an object of type class Derived2
```

Anlattığım gibi **typeid** polimorfik tipten olan taban sınıf işaretçisine uygulandığında, işaret edilen nesnenin tipi çalışma anında elde edilir. Çıkış da bunu kanıtlıyor zaten... Deney amacıyla **BaseClass** içerisinde sanal **f()** fonksiyonu ile oynayın ve sonuçları gözleyin.

- Yine anlattığım gibi, **typeid** polimorfik taban sınıfa olan başvuruya uygulandığında, döndürülen, başvurulan gerçek nesnenin tipidir. Bu yenilikleri kullanmanız gereken durumlar nesneleri fonksiyonlara başvuru geçirdiğinizde ortaya çıkar. Örneğin, aşağıdaki programda **WhatType()** fonksiyonu **BaseClass** tipinden nesnelere başvuru parametresi bildirir. Bu işlem **WhatType()**'nın, **BaseClass** veya ondan türemiş sınıflara ait nesnelere başvuru olarak geçirilebileceği anlamına gelir. Bu parametreye **typeid** operatörü uygulandığında, geçenilen nesnenin gerçek tipi döndürülür.

```
// typeid ile başvuru kullanma
#include <iostream>
#include <typeinfo>
using namespace std;

class BaseClass {
    virtual void f() {} // BaseClass'ı polimorfik yapma
    // ...
};

class Derived1: public BaseClass {
    // ...
};

class Derived2: public BaseClass {
    // ...
};

// type id'yi başvuru parametresi ile gösterme.
void WhatType(BaseClass *ob)
{
    cout << "ob is referencing an object of type ";
    cout << typeid(ob).name() << endl;
}

int main()
{
    int i;
    BaseClass baseob;
    Derived1 ob1;
    Derived2 ob2;
    WhatType(baseob);
    WhatType(ob1);
    WhatType(ob2);
    return 0;
}
```

Program tarafından üretilen çıktı aşağıda gösterilmiştir:

```
ob is referencing an object of type class BaseClass
ob is referencing an object of type class Derived1
ob is referencing an object of type class Derived2
```

3. Bazı durumlarda nesnenin tip ismini elde etmek kullanışlı olsa da, genellikle bütün bilmek istediğiniz şey bir nesnenin tipinin diğeriley uyuşup uyuşmadığıdır. `typeid` tarafından döndürülen `type_info` nesnesi `==` ve `!=` operatörlerini aşırı yüklediğinden bu olayı gerçekleştirmek çok basitleşmiştir. Aşağıdaki program bu operatörlerin kullanımını gösteriyor.

```
// -- ve != operatörlerinin typeid'ye bağlı olarak gösterimi
#include <iostream>
#include <typeinfo>
using namespace std;

class X {
    virtual void f() {}
};

class Y {
    virtual void f() {}
};

int main()
{
    X x1, x2;
    Y y1;

    if(typeid(x1) == typeid(x2))
        cout << "x1 and x2 are same types\n";
    else
        cout << "x1 and x2 are different types\n";

    if(typeid(x1) != typeid(y1))
        cout << "x1 and y1 are different types\n";
    else
        cout << "x1 and y1 are same types\n";

    return 0;
}
```

Program tarafından üretilen çıktı aşağıda gösterilmiştir:

```
x1 and x2 are same types
x1 and y1 are different types
```

4. Önceki programlar `typeid`'nin mekanlığını gösterse de onun tam performansı sergileyememektedirler. Çünkü önceki programlarda geçen tiplerin bilgisi derleme aşamasında da bilinebilir. Şimdiki programımızda daha farklı bir olay incelenecak. Program ekrana şekillere çizen basit bir sınıf hiyerarşisi tanımlıyor. Hiyerarşinin üst tarafında `shape` soyut sınıfı var. Onun dışında `Line`, `Square`, `Rectangle` ve `NullShape` adında 4 adet alt sınıf oluşturuluyor. `generator()` fonksiyonu bir nesne üretiyor ve ona bir işaretçi döndürüyor. Oluşturulan gerçek nesne rasgele bir şekilde sayı üretici `rand()` gelen çıkıştı taban alarak belirleniyor. (Nesne oluşturan fonksiyonlara bazı zamanlar "nesne fabrikası (object factory)" denir). `main()` içerisinde, şekli olmayan `NullShape` nesnesi dışında her nesnenin şecli gösteriliyor.

Nesneler rasgele oluşturulduğundan bir sonraki nesnenin tipini bilmek mümkün değildir. Bu sebeple RTTI'nın kullanımı gerekmektedir.

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

class Shape {
public:
    virtual void example() = 0;
};

class Rectangle: public Shape {
public:
    void example() {
        cout << "*****\n* *\n*****\n";
    }
};

class Triangle: public Shape {
public:
    void example() {
        cout << "*\n* *\n*****\n";
    }
};

class Line: public Shape {
public:
    void example() {
        cout << "*****\n";
    }
};

class NullShape: public Shape {
public:
    void example() {
    }
};

// Shape' ten türeyen nesnelerin fabrikası
Shape *generator()
{
    switch(rand() % 4) {
        case 0:
            return new Line;
        case 1:
            return new Rectangle;
        case 2:
            return new Triangle;
        case 3:
            return new NullShape;
    }

    return NULL;
}

int main()
{
    int i;
```

```
Shape *p;

for(i=0; i<10; i++) {
    p = generator(); // bir sonraki nesneyi al

    cout << typeid(*p).name() << endl;

    // Nullshape dışında gelen nesnenin şeklini çiz.
    if(typeid(*p) != typeid(NullShape))
        p->example();
}

return 0;
}
```

Programın örnek çıktısı aşağıdaki gibidir.

```
class Rectangle
*****
*
*
*****
class NullShape
class Triangle
*
*
*
*****
class Line
*****
class Rectangle
*****
*
*
*****
class Line
*****
class Triangle
*
*
*
*****
class Triangle
*
*
*
*****
class Triangle
*
*
*
*****
class Line
*****
```

5. **typeid** operatörü şablon sınıflara da uygulanabilir. Örneğin, aşağıdaki programı ele alalım. Bir değer taşıyan şablon sınıflarının hiyerarşini oluşturuyor. **get_val()** adındaki sanal fonksiyon her sınıf tarafından tanımlanan değeri döndürüyor. **Num** si-

nifi için, değerin kendisi, **Square** için değerin karesi ve **square_root** için değerin karekökü tanımlanıyor. **Num**'dan türetilen nesneler **generator()** tarafından üretiliyor. **typeid** operatörü ise ne tip bir nesnenin üretildiğini saptıyor.

```
// typeid şablonları ile kullanılabilir
#include <iostream>
#include <typeinfo>
#include <cmath>
#include <cstdlib>
using namespace std;

template <class T> class Num {
public:
    T x;
    Num(T i) { x = i; }
    virtual T get_val() { return x; }
};

template <class T>
class Square : public Num<T> {
public:
    Square(T i) : Num<T>(i) {}
    T get_val() { return x*x; }
};

template <class T>
class Sqr_root : public Num<T> {
public:
    Sqr_root(T i) : Num<T>(i) {}
    T get_val() { return sqrt((double) x); }
};

// Num'dan türetilmiş nesneler için nesne fabrikası
Num<double> *generator()
{
    switch(rand() % 2) {
        case 0: return new Square<double> (rand() % 100);
        case 1: return new Sqr_root<double> (rand() % 100);
    }
    return NULL;
}

int main()
{
    Num<double> ob1(10), *p1;
    Square<double> ob2(100.0);
    Sqr_root<double> ob3(999.2);
    int i;

    cout << typeid(ob1).name() << endl;
    cout << typeid(ob2).name() << endl;
    cout << typeid(ob3).name() << endl;

    if(typeid(ob2) == typeid(Square<double>))
        cout << "is Square<double>\n";
    p1 = &ob2;

    if(typeid(*p1) != typeid(ob1))
        cout << "Value is: " << p1->get_val();
```

```

cout << "\n\n";

cout << "Now, generate some Objects.\n";
for(i=0; i<10; i++) {
    pi = generator(); //bir sonraki nesneyi al

    if(typeid(*pi) == typeid(Square<double>))
        cout << "Square object: ";
    if(typeid(*pi) == typeid(Sqr_root<double>))
        cout << "Sqr_root object: ";

    cout << "Value is: " << pi->get_val();
    cout << endl;
}

return 0;
}

```

Programın çıktısı aşağıda gösterilmiştir:

```

class Num<double>
class Square<double>
class Sqr_root<double>
is Square<double>
Value is: 10000

Now, generate some Objects.
Sqr_root object: Value is: 8.18535
Square object: Value is: 0
Sqr_root object: Value is: 4.89898
Square object: Value is: 3364
Square object: Value is: 4096
Sqr_root object: Value is: 6.7082
Sqr_root object: Value is: 5.19615
Sqr_root object: Value is: 9.53939
Sqr_root object: Value is: 6.48074
Sqr_root object: Value is: 6

```

Aliştırmalar

1. Neden RTTI, C++ için gerekli bir özelliktir?
 2. Örnek 1'de açıklanan denemeyi tekrar yapın. Çıkışta ne görüyorsunuz?
 3. Aşağıdaki kod doğru mudur?
- ```
cout << typeid(float).name();
```
4. Size aşağıdaki kod veriliyor. p'nin D2 nesnesini işaret ettiğini nasıl saptanacağını gösterin.

```

class B {
 virtual void f() {}
};

class D1: public B {
 void f() {}
};

```

```
class D2: public B {
 void f() {}
};

int main()
{
 B *p;
```

5. Örnek 5'deki num sınıfını baz alarak aşağıda yazılanlar doğru mudur?

```
typeid(Num<int>) == typeid(Num<double>)
```

6. Kendi başınıza RTTI'yi deneyin. Her ne kadar kullanımını biraz gizli gözükse de nesneleri çalışma anında yönetmenize imkan veren güçlü bir araçtır.

## 12.2. dynamic\_cast'i Kullanma

Her ne kadar C++, C tarafından tanımlanan eski dönüşümme operatörlerini halen desteklese de bunlara dört yenisini daha eklemektedir. Bunlar **dynamic\_cast**, **const\_cast**, **reinterpret\_cast** ve **static\_cast**'dır. **dynamic\_cast**'ı ilk önce inceleyeceğiz, çünkü RTTI ile alakalıdır. Diğer dönüşümme operatörleri bir sonraki bölümde ele alınacaktır.

**dynamic\_cast**, operatörü dönüşümenin geçerliliğini doğrulayan bir çalışma anı dönüşüm gerçekleştirir. Eğer, **dynamic\_cast**'ın işletilmesi sırasında dönüşüm geçersizse bu dönüşüm başarısızlığa uğrar. **dynamic\_cast**'ın genel formu aşağıdaki şekildedir:

```
dynamic_cast<target-type> (expr)
```

Buradaki *target-type* dönüşümün hedef tipini belirler ve *expr* ise yeni type dönüştürülen ifadedir. Hedef tip, bir işaretçi veya başvuru tipi olmalıdır. Ayrıca dönüştürülen ifade bir işaretçi veya referansı değerlendirmelidir. Böylelikle **dynamic\_cast** bir işaretçi tipini diğerine veya diğer bir referans tipine dönüştürmek için kullanılabilir.

**dynamic\_cast**'ın amacı polimorfik tiplerde dönüşüm sağlamaktır. Örneğin, **B** ve **D** altında iki polimorfik sınıfımız olsun. **D**'de **B**'den türemiş olsun. **dynamic\_cast** her zaman **D\*** işaretçisini **B\*** işaretçisine dönüştürebilir. Bunun nedeni taban işaretçisinin her zaman türemiş nesneyi işaret edebilmesidir. Fakat **dynamic\_cast**, eğer işaret edilen gerçek **D** nesnesi ise **B\*** işaretçisini **D\*** işaretçisine dönüştürülebilir. Genelde **dynamic\_cast** eğer dönüştürülen işaretçi (veya referans), hedef tipten bir nesneyi veya hedef tipten türemiş bir nesneyi gösteren işaretçi (veya referans ise) başarılı olacaktır. Aksi halde dönüşüm başarısız olur. Eğer dönüşüm başarısız olursa ve dönüşüm işaretçi içeriyorsa **dynamic\_cast** boş değerlendirmesi yapar. Eğer referans tiplerindeki **dynamic\_cast** başarısız olursa bir **bad\_cast** hatası atılır.

İste size basit bir örnek veriyorum. **Base**'in polimorfik bir sınıf olduğunu ve **Derived**'in **Base**'den türediğini farz edin.

```
Base *bp, b_obj;
Derived *dp, d_obj;
```

```
bp = &d_obj; // taban işaretçisi
dp = dynamic_cast<Derived *> (bp);
if(dp) cout << "Dönüştürme OK";
```

Burada, **bp** taban işaretçisinden türemiş **dp** işaretçisine dönüşüm gerçekleşir, çünkü gerçekte **bp**, **Derived** nesnesine işaret eder. Böylelikle bu kod **Dönüştürme Ok** mesajını gösterir. Fakat bir sonraki kodda dönüşüm başarısızlığa uğrar, çünkü **bp**, **Base** nesnesini işaret eder ve bir taban nesnenin türemiş nesneye dönüşümü geçersizdir.

```
bp = &b_obj; // taban işaretçisi taban nesnesini işaret eder.
dp = dynamic_cast<Derived *> (bp);
if(!dp) cout << "Dönüştürme başarısızlığı uğradı";
```

Dönüştürme başarısızlığa uğradığından bu kod **Dönüştürme başarısızlığı uğradı** mesajını gösterir. **dynamic\_cast** operatörü bazı durumlarda zaman zaman **typeid**'nin yerine kullanılabilir. Orneğin, **Base**'in **Derived** için polimorfik taban sınıf olduğunu farz edin. Aşağıdaki kodda **dp**'ye **bp** tarafından işaret edilen nesnenin adresi atanacaktır. Bu durum sadece nesnenin sadece **Derived** nesne olması durumunda gerçekleşir.

```
Base *bp;
Derived *dp;
// ...
if(typeid(*bp) == typeid(Derived)) dp = (Derived *) bp;
```

Bu durumda dönüşümü gerçekleştirmek için C tipi dönüşüm kullanılır. Bu işlem güvenlidir, çünkü **if** ifadesi dönüşümün gerçekleşmesinden önce **typeid** kullanarak dönüşümün olabilirliğini kontrol eder. Fakat, bunu gerçekleştirmenin daha iyi bir yolu **typeid** operatörünü ve **if** ifadesini **dynamic\_cast** ile değiştirmektir:

```
dp = dynamic_cast<Derived *> (bp);
```

Çünkü sadece dönüştürülen nesne, hedef tipten veya ondan türemiş bir nesneyse **dynamic\_cast** başarıya ulaşır. Bu ifade işlediğinde **dp**'nin içeriği ya boş olacaktır ya da **Derived** tipinden bir nesneye işaretçi içerecektir. **dynamic\_cast** sadece dönüşüm geçerli olduğunda başarıya ulaştığından belirli durumlardaki mantığı basite indirger.

## Örnekler

1. Aşağıdaki program **dynamic\_cast**'ı göstermektedir:

```
// dynamic_cast gösteriliyor.
#include <iostream>
using namespace std;

class Base {
public:
 virtual void f() { cout << "Inside Base\n"; }
 // ...
};

class Derived : public Base {
```

## BÖLÜM 12: Çalışma Anı Tip Tanıma ve Dönüşümme Operatörleri

```

public:
 void f() { cout << "Inside Derived\n"; }

int main()
{
 Base *bp, b_obj;
 Derived *dp, d_obj;

 dp = dynamic_cast<Derived *> (&d_obj);
 if(dp) {
 cout << "Cast from Derived * to Derived * OK.\n";
 dp->f();
 } else
 cout << "Error\n";

 cout << endl;

 bp = dynamic_cast<Base *> (&d_obj);
 if(bp) {
 cout << "Cast from Derived * to Base * OK.\n";
 bp->f();
 } else
 cout << "Error\n";

 cout << endl;

 bp = dynamic_cast<Base *> (&b_obj);
 if(bp) {
 cout << "Cast from Base * to Base * OK.\n";
 bp->f();
 } else
 cout << "Error\n";

 cout << endl;

 dp = dynamic_cast<Derived *> (&b_obj);

 if(dp)
 cout << "Error\n";
 else
 cout << "Cast from Base * to Derived * not OK.\n";

 cout << endl;

 bp = &d_obj; // bp Derived nesnesini işaret eder
 dp = dynamic_cast<Derived *> (bp);
 if(dp) {
 cout << "Casting bp to a Derived * OK\n" <<
 "because bp is really pointing\n" <<
 "to a Derived object.\n";
 dp->f();
 } else
 cout << "Error\n";
 cout << endl;

 bp = &b_obj; // bp Base nesnesini işaret eder
 dp = dynamic_cast<Derived *> (bp);
 if(dp)
 cout << "Error";
 else {

```

```

 cout << "Now casting bp to a Derived *\n" <<
 "is not OK because bp is really\n" <<
 "pointing to a Base object.\n";
}

cout << endl;

dp = sd_obj; // dp Derived nesnesini işaret eder
bp = dynamic_cast<Base *> (dp);

if(bp) {
 cout << "Casting dp to a Base * is OK.\n";
 bp->f();
} else {
 cout << "Error\n";
}

return 0;
}

```

Programın çıktısı aşağıdaki gibidir:

```

Cast from Derived * to Derived * OK.
Inside Derived

Cast from Derived * to Base * OK.
Inside Derived

Cast from Base * to Base * OK.
Inside Base

Cast from Base * to Derived * not OK.

Casting bp to a Derived * OK
because bp is really pointing
to a Derived object.
Inside Derived

Now casting bp to a Derived *
is not OK because bp is really
pointing to a Base object.

Casting dp to a Base * is OK.
Inside Derived

```

2. Aşağıdaki örnek **dynamic\_cast**'in **typeid**'nin yerini değiştirmek için nasıl kullanıldığını gösterir.

```

// dynamic cast'i typeid'nin yerini değiştirmek için kullanın.
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
public:
 virtual void f() {}
};

class Derived : public Base {
public:

```

```

void derivedOnly() {
 cout << "Is a Derived Object\n";
}

int main()
{
 Base *bp, b_ob;
 Derived *dp, d_ob;

 // *****
 // typeid kullan
 // *****
 bp = &b_ob;
 if(typeid(*bp) == typeid(Derived)) {
 dp = (Derived *) bp;
 dp->derivedOnly();
 }
 else
 cout << "Cast from Base to Derived failed.\n";
 bp = &d_ob;
 if(typeid(*bp) == typeid(Derived)) {
 dp = (Derived *) bp;
 dp->derivedOnly();
 }
 else
 cout << "Error, cast should work!\n";

 // *****
 // dynamic_cast kullan
 // *****
 bp = &b_ob;
 dp = dynamic_cast<Derived *> (bp);
 if(dp) dp->derivedOnly();
 else
 cout << "Cast from Base to Derived failed.\n";

 bp = &d_ob;
 dp = dynamic_cast<Derived *> (bp);
 if(dp) dp->derivedOnly();
 else
 cout << "Error, cast should work!\n";

 return 0;
}

```

Gördüğünüz gibi **dynamic\_cast**'in kullanımı taban işaretçiyi türemiş işaretçiye dönüştürülmesi mantığını basitleştirir. Program aşağıdaki çıktıyi verir:

```

Cast from Base to Derived failed.
Is a Derived Object
Cast from Base to Derived failed.
Is a Derived Object

```

3. **dynamic\_cast** operatörü şablon sınıflarla da kullanılabilir. Örneğin aşağıdaki programda bir önceki konunun Örnek 5'indeki şablon sınıf kullanılmaktadır. Böylece **genera\*or( )** fonksiyonundan döndürülen nesnenin tipini saptamak için **dynamic\_cast**'i kullanır.

```
// dynamic_cast şablonları da kullanılabilir.
#include <iostream>
#include <typeinfo>
#include <cmath>
#include <cstdlib>
using namespace std;

template <class T> class Num {
public:
 T x;
 Num(T i) { x = i; }
 virtual T get_val() { return x; }
};

template <class T>
class Square : public Num<T> {
public:
 Square(T i) : Num<T>(i) {}
 T get_val() { return x*x; }
};

template <class T>
class Sqr_root : public Num<T> {
public:
 Sqr_root(T i) : Num<T>(i) {}
 T get_val() { return sqrt((double) x); }
};

// Num'dan türeyen nesneler için bir fabrika.
Num<double> *generator()
{
 switch(rand() % 2) {
 case 0: return new Square<double> (rand() % 100);
 case 1: return new Sqr_root<double> (rand() % 100);
 }
 return NULL;
}

int main()
{
 Num<double> ob1(10), *p1;
 Square<double> ob2(100.0), *p2;
 Sqr_root<double> ob3(999.2), *p3;
 int i;

 cout << "Generate some objects.\n";
 for(i=0; i<10; i++) {
 p1 = generator();

 p2 = dynamic_cast<Square<double> *> (p1);
 if(p2) cout << "Square object: ";
 p3 = dynamic_cast<Sqr_root<double> *> (p1);
 if(p3) cout << "Sqr_root object: ";

 cout << "Value is: " << p1->get_val();
 cout << endl;
 }

 return 0;
}
```

## Aliştırmalar

- Kendi sözcüklerinizle `dynamic_cast`'in amacını açıklayın.
- Size aşağıdaki kod veriliyor. `dynamic_cast`'i kullanarak, sadece ve sadece `ob`, `D2`'nin bir nesnesi ise `p`'ye bazı `ob` nesnelerini gösteren bir işaretçiyi nasıl atayacağınızı gösterin.

```
class B {
 virtual void f() {}
};

class D1: public B {
 void f() {}
};

class D2: public B {
 void f() {}
};

B *p;
```

- Konu 12.1, Örnek 4'deki `main()` fonksiyonunu dönüştürün ve `NullShape` nesnesinin görüntülenmesini engellemek için `typeid` yerine `dynamic_cast` kullanmasını sağlayın.
- Bu konunun Örnek 3'ünün `Num` sınıfını kullanırsak aşağıdaki kod çalışır mı?

```
Num<int> *Bp;
Square<double> *Dp;
// ...
Dp = dynamic_cast<Num<int>> (Bp);
```

## 12.3. `const_cast`, `reinterpret_cast` ve `static_cast`

Her ne kadar `dynamic_cast` yeni dönüşüm operatörlerinin en önemlisi olsa da diğer üçü de programcılar için değerlidir. Onların genel formlarını burada gösteriyoruz:

```
const_cast<target-type> (expr)
reinterpret_cast<target-type> (expr)
static_cast<target-type> (expr)
```

Burada *target-type* dönüştürmenin hedef tipini ve *expr* ise yeni tipে dönüştürülen ifadeyi belirler. Genelde bu dönüştürme operatörleri, belirli tip dönüşümlerinin gerçekleştirilmesini, C stili dönüşümlerin sağladığından daha güvenli ve daha açık hale getirir.

`const_cast` operatörü açık bir şekilde dönüşümdeki `const` ve/veya `volatile`'ın üzerine yazmak için kullanılır. Hedef tip, `const` veya `volatile`'ın özelliklerinin değiştirilmesi haricinde kaynak tipiyle aynı olmalıdır. `const_cast`, en sık `const-ness`'i yok etmek için kullanılır.

`static_cast` operatörü polimorfik olmayan bir dönüşüm gerçekleştirir. Örneğin bir taban sınıf işaretçisini bir türemiş sınıf işaretçisine dönüştürmek için kullanılabilir. Ayrıca herhangi bir standart dönüşüm için de kullanılabilir. Çalışma sırasında kontrol yapılmaz.

**reinterpret\_cast** operatörü bir değişken tipini bir diğerine, oldukça farklı bir işaretçi tipine çevirir. Aynı zamanda bir işaretçiyi bir tamsayıya ve bir tamsayıyı da bir işaretçiyi dönüştürebilir. **reinterpret\_cast**, içerik yönünden karşılaşılamayan işaretçi tiplerinin dönüşümü için kullanılmalıdır.

Sadece **const\_cast**, **const-ness'i** dönüştürebilir. Yani, ne **dynamic\_cast**, ne **static\_cast**, ne de **reinterpret\_cast** nesnenin **const-ness'ini** değiştiremez.

## Örnekler

1. Aşağıdaki program **reinterpret\_cast**'in kullanımını gösterir.

```
// reinterpret_cast'i kullanan bir örnek.
#include <iostream>
using namespace std;

int main()
{
 int i;
 char *p = "This is a string";

 i = reinterpret_cast<int> (p); // işaretçiyi tamsayıya dönüştürür

 cout << i;

 return 0;
}
```

Burada **reinterpret\_cast**, p işaretçisini bir tamsayıya dönüştürür. Bu dönüşüm önemli bir değişimini temsil eder ve **reinterpret\_cast**'in iyi bir kullanımıdır.

2. Aşağıdaki program **const\_cast**'i gösteriyor.

```
// const_cast'i gösteriyor.
#include <iostream>
using namespace std;

void f(const int *p)
{
 int *v;

 // const-ness'iyi dönüştür,
 v = const_cast<int *> (p);

 *v = 100; // şimdi, nesneyi v üzerinden değiştirdi
}

int main()
{
 int x = 99;

 cout << "x before call: " << x << endl;
 f(&x);
 cout << "x after call: " << x << endl;

 return 0;
}
```

Program şu çıkışını vermektedir:

```
x before call: 99
x after call: 100
```

Gördüğünüz gibi, **f()**'ın parametreleri bir **const** işaretçi olarak belirlenmişse de, **x**, **f()** tarafından değiştirilmiştir.

**const\_cast**'in **const-ness'i** dönüştürmek için kullanılmasının potansiyel olarak tehlikeli bir özellik olduğu vurgulanmalıdır. Onu dikkatli bir şekilde kullanın.

3. **static\_cast** operatörü aslında ilk dönüşüm operatörünün yerine geçmiştir. Basitçe polimorfik olmayan bir dönüşüm gerçekleştirmektedir. Örneğin aşağıdaki program bir **float** sayısını bir **int**'e dönüştürmektedir.

```
// static_cast'i kullanıyor
#include <iostream>
using namespace std;

int main()
{
 int i;
 float f;

 f = 199.22;
 i = static_cast<int> (f);
 cout << i;

 return 0;
}
```

## Alıştırmalar

1. **const\_cast**, **reinterpret\_cast** ve **static\_cast**'i açıklayın.
2. Aşağıdaki programda bir hata vardır. **const\_cast**'i kullanarak nasıl düzeltileceğini gösterin.

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

void f(const double &i)
{
 i = 100; // Hata -- const_cast'i kullanarak düzelt
}

int main()
{
 double x = 98.6;
 cout << x << endl;
 f(x);
 cout << x << endl;
 return 0;
}
```

3. **const\_cast**'in normalde özel durumlar için saklanması nedenini açıklayın.

## Pekiştirme Testi

Bu noktada aşağıdaki alıştırmaları yapabiliyor ve soruları cevaplayabiliyor olmalısınız.

1. `typeid`'nin işleyişini anlatın.
2. `typeid`'yi kullanmak için hangi başlığı dahil etmelisiniz?
3. Standart dönüşüm ek olarak C++, dört dönüşüm operatörü daha tanımlamaktadır. Bunlar nelerdir ve ne amaçla kullanılır?
4. Aşağıdaki programı kullanıcı tarafından seçilen nesne tipini rapor edecek şekilde tamamlayın.

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

class A {
 virtual void f() {}
};

class B : public A {
};

class C: public B {
};

int main()
{
 A *p, a_obj;
 B b_obj;
 C c_obj;
 int i;

 cout << "Enter 0 for A objects, ";
 cout << "1 for B objects or ";
 cout << "2 for C objects.\n";

 cin >> i;

 if(i==1) p = &b_obj;
 else if(i==2) p = &c_obj;
 else p = &a_obj;

 // kullanıcı tarafından seçilen nesne tipini rapor et

 return 0;
}
```

5. `dynamic_cast`'in bazı zamanlar nasıl `typeid`'ye bir alternatif olabildiğini açıklayın.
6. `typeid` operatörü tarafından ne tipte bir nesne alınır?

## Bütünleştirme Testi

Bu kısım sizin geçmiş bölümlerdeki konular da dahil olmak üzere, bu bölümde geçen konuları ne kadar kavradığınızı kontrol edecektir.

1. Konu 12.1, Örnek 4'deki programı, **generator()** fonksiyonu içerisinde bellekte yer ayırma hatasını izlemek için hata denetimi yapacak şekilde değiştirebilir.
2. Soru 1'deki **generator()** fonksiyonunu **new**'in **nothrow** sürümünü kullanacak şekilde değiştirebilir. Hataları denetleyin.
3. Özel zorlukta: Kendi kendinize en üstünde **DataStruct** adında soyut bir sınıf olan bir sınıf hiyerarşisi oluşturun. İki tane **concrete** alt sınıfı oluşturun. Biri yiğin diğerı kuyruk olsun. Aşağıdaki prototipe sahip **DataStructFactory()** adında bir fonksiyon oluşturun:

```
DataStruct *DataStructFactory(char what);
```

**DataStructFactory()**, *what* s ise yiğin (stack), q ise kuyruk (queue) oluşturulsun. Oluşturulan nesneye bir işaretçi döndürün. Fabrika fonksiyonunuzun çalıştığını gösterin.

www.Gerogoku.com

## BÖLÜM 13

# Namespace'ler, Dönüştürme Fonksiyonları ve Çeşitli Konular

**Namespace'ler**

**Dönüştüm Fonksiyonu Oluşturma**

**static Sınıf Üyeleri**

**const Üye Fonksiyonlar ve mutable**

**Yapıcılara Son Bir Bakış**

**Bağlantı Belirticileri ve asm Anahtar Sözcüğü**

**Dizi-Tabanlı I/O**

Bu bölümde namespace'ler, dönüşüm fonksiyonları, static ve const sınıf üyeleri ve C++'ın diğer özelliklerini ele alacağız.

## Gözden Geçirme Testi

Daha ileri gitmeden, aşağıdaki soruları doğru cevaplayabiliyor ve alıştırmaları doğru yapabiliyor olmalısınız.

1. Dönüşürme operatörleri nedir ve ne iş yaparlar?
2. `type_info` nedir?
3. Nesne tipini hangi operatör tanımlar?
4. Size aşağıdaki kodu veriyorum. `p`'nin bir **Base** nesnesini mi yoksa bir **Derived** nesnesini mi işaret ettiğinin nasıl anlaşılacağını anlatın.

```
class Base {
 virtual void f() {}
};

class Derived : public Base {
};

int main()
{
 Base *p, b_obj;
 Derived d_obj;
 // ...
}
```

5. `dynamic_cast` eğer dönüştürülen hedef tipten bir nesneye işaretçi ise veya hedef tipten \_\_\_\_\_ bir nesneyse başarılı olur. (Boşlukları doldurun.)
6. `dynamic_cast`, `const-ness`'leri dönüştürebilir mi?

### 13.1. Namespace'ler (Bildirim Bölgeleri)

Namespace'lerden Bölüm 1'de kısaca bahsetmiştik. Şimdi onları detaylarıyla incelemenin zamanı geldi. Namespace'ler C++'a son zamanlarda eklenmiş bir yenilikti. Amaçları değişken isimlerini, isim çakışmalarını önlemek amacıyla yerelleştirmektir. C++ ortamında değişken, fonksiyon ve sınıf isimlerinde bir patlama yaşanmıştır. Namespace'lerin keşfinden önce bütün bu isimler global bir namespace içerisinde boşluk bulmak için yarışıyordu ve bir çok çakışma meydana geliyordu. Örneğin programınız `toupper()` adında bir fonksiyon tanımladığında, standart kütüphane fonksiyonu olan `toupper()` fonksiyonun(parametre listesine bağlı olarak) üzerine bindirilebilir. Çünkü her iki isimde aynı bildirim bölgesi içerisinde bulunmaktadır. İsim çakışmaları aynı program tarafından, iki veya daha fazla üçüncü parti yazılım kullanıldığında çoğalır. Bu durumda karmaşıklığın meydana gelmesi olağan hale gelir. Hatta bir kütüphane tarafından tanımlanan isim diğer kütüphane tarafından tanımlanan isimle çakışabilir.

Anahtar kelimeye **Namespace** isminin verilmesi, bu probleme çözüm getirdiğinden gerçekleşmiştir. İçerisinde belirtilen isimlerin görünebilirliğini yerelleştirdiğinden, bildirim bölgeleri değişik alanlarda isim çakışmasına imkan vermeden aynı isimlerin kullanılabilirliğine ortam sağlar. Bildirim bölgelerinin dikkate değer en verimli kullanımını herhalde C++ standart kütüphanesinde gerçekleştirilmiştir. C++'ın ilk sürümlerinde C++ kütüphanesi global bildirim bölgesinde tanımlanmıştı (tabii ki tek bildirim bölgesi idi) Ancak şimdi C++ kütüphanesi, isim çakışma ihtimalini azaltan, std denen kendi bildirim bölgesinde tanımlanmıştır. Siz de programlarınızda, çakışacağınızı tahmin ettiğiniz isimlerinizin görünebilirliğini, kendi bildirim bölgelerinizi oluşturarak yerelleştirebilirsiniz. Bu olay özellikle sınıf veya fonksiyon kütüphaneleri oluşturuyorken önem kazanır.

**Namespace** anahtar kelimesi bildirilebilir bir bölge oluşturarak, global bildirim bölgesini bölümler. İşin aslı, bildirim bölgesi bir alan tanımlar. **Namespace**'ın genel formu aşağıda gösterilmiştir:

```
namespace name {
 // bildirimler
}
```

**namespace** ifadesinin içerisinde tanımlanan her şey o bildirim bölgesinin içerisindeidir. Size **namespace** ile ilgili bir örnek veriyorum:

```
namespace MyNameSpace {
 int i, k;
 void myfunc(int j) { cout << j; }

 class myclass {
 public:
 void seti(int x) { i = x; }
 int geti() { return i; }
 };
}
```

Burada **i**, **k**, **myfunc( )** ve **myclass** sınıfı **MyNameSpace** bildirim bölgesi tarafından tanımlana alanın parçası halindedir.

Bildirim bölgesi içerisinde tanımlanan değişkenlere aynı bildirim bölgesi içerisinde doğrudan başvurulabilir. Örneğin, **MyNameSpace**'teki **return i** ifadesi **i**'yi doğrudan kullanıyor. Ancak, **namespace** bir bölge tanımladığından, bildirim bölgesinde tanımlanan nesnelere başvururken o bildirim bölgesinin dışından kapsam çözümleme operatörü kullanmak gereklidir. Örneğin, **MyNameSpace** dışından **i**'ye 10 değeri atamak için şu ifadeyi kullanmak zorundasınız.

```
MyNameSpace::i = 10;
```

Ya da **MyNameSpace** dışında, **myclass** tipinden bir nesne bildirmek için aşağıdaki ifadeyi kullanmalısınız:

```
MyNameSpace::myclass ob;
```

Genelde, bir bildirim bölgesinin üyesine dışarıdan erişmek için, üye isminin önüne bildirim bölgesinin isminden sonra kapsam çözümleme operatörü yerleştirir.

Tahmin ettiğiniz gibi, eğer programınız bildirim bölgesi üyelerine sık sık erişim ihtiyacı duyuyorsa, her seferinde kapsam çözümleme operatörünü kullanmak zorunda kalırsınız. **Using** ifadesi işte bu problemi çözmek için keşfedilmiştir. **Using** ifadesinin iki genel formu vardır:

```
using namespace name;
using name::member;
```

Birinci yapıda *name*, erişmek istediğiniz bildirim bölgesinin ismini belirtir. Bu formu kullandığınızda, belirtilen bildirim bölgesinin içerisinde kullandığınız bütün üyeleri, şu anki bildirim bölgesine taşınır ve herhangi bir belirtece gerek kalmaz. İkinci yapıda bildiri bölgesinde, sadece belli bir üye görünebilir yapılır. Örneğin, **MyNameSpace**'ı kullandığımızı farz edersek, aşağıdaki **using** ifadeleri geçerli olur:

```
using MyNameSpace::k; // sadece k görünür yapıldı
k = 10; // Geçerlidir

using namespace MyNameSpace; // Butun üyeler görünür yapıldı
i = 10; // Geçerlidir çünkü butun üyeler görünür haldedir
```

Aynı isimde birden fazla bildirim bölgesi olabilir. Bu işlem bildirim bölgesinin farklı bölgelere ayrılmasına imkan verir. Aşağıdaki örneği iyi inceleyin.

```
namespace NS {
 int i;
}

// ...

namespace NS {
 int j;
}
```

Burada **NS** iki parçaya ayrılmıştır. Ancak, her parçanın içeriği aynı bildirim bölgesi içersindedir.

Bir bildirim bölgesi diğer bütün bölgelerin dışında bildirilmelidir. Bu noktada bir istisna var. O da bildiri bölgesinin bir diğerinin içerisinde yuvalanabilmesidir. Yani bu demektir ki, bir fonksiyona yerleştirilmiş bildirim bölgesi kuramazsınız.

Bildirim bölgelerinin *unnamed namespace* (isimsiz bildirim bölgesi) denen özel bir tipi vardır. Bu tip bir bildirim bölgesi bir dosya içerisinde tek olarak değişkenleri tanımlamana-za imkan verir. Aşağıdaki gibi bir genel yapısı vardır.

```
namespace {
 // declarations
}
```

Birim dosyalarının bildirim bölgeleri size, sadece bir dosyanın bölümü altında bilinen tek dosyalarının kullanımını sağlar. Yani sınırlı bildirim bölgesi içeren dosya içerisinde, o alana ait özniteliklerin kullanımı yapılmadan doğrudan kullanılabilir. Fakat dosya dışında değişkenler tanınmaz.

Genellikle küçük ve orta ölçekli programlar için bildirim bölgeleri kullanmaya ihtiyaç duyulmaz. Ancak, eğer çok kod içeren kütüphaneler oluşturacaksanız veya en uyumlu şekilde program geliştirecekseniz kodunuzu bildirim bölgeleri ile bölümlemeyi düşünün.

## Örnekler

- Size namespace'in kullanımını gösteren bir program hazırladım.

```
// Namespace Örneği
#include <iostream>
using namespace std;

// namespace tanımla
namespace firstNS {
 class demo {
 int i;
 public:
 demo(int x) { i = x; }
 void seti(int x) { i = x; }
 int geti() { return i; }
 };

 char str[] = "Illustrating namespaces\n";
 int counter;
}

// Diğer bir namespace tanımla
namespace secondNS {
 int x, y;
}

int main()
{
 // Kapsam çözümleme operatörünü kullan
 firstNS::demo ob(10);
 /* ob bir kere bildirildiğinde
 üye fonksiyonları namespace sınıması olamadan kullanılabilirler */
 cout << "Value of ob is : " << ob.geti();
 cout << endl;
 ob.seti(99);
 cout << "Value of ob is now : " << ob.geti();
 cout << endl;

 // str'yi şu anki kapsamda getir
 using firstNS::str;
 cout << str;

 // Bütün firstNS'ı şu anki kapsamda getir.
 using namespace firstNS;
 for(counter = 10; counter; counter--)
 cout << counter << " ";
}
```

```

cout << endl;

// secondNS namespace'ı kullanın
secondNS::x = 10;
secondNS::y = 20;

cout << "x, y: " << secondNS::x;
cout << ", " << secondNS::y << endl;

// Diğer namespace'ı görünür kılın
using namespace secondNS;
demo xob(x), yob(y);

cout << "xob, yob: " << xob.geti() << ", ";
cout << yob.geti() << endl;

return 0;
}

```

Programın çıkışı aşağıdaki gibidir.

```

Value of ob is : 10
Value of ob is now : 99
Illustrating namespaces
10 9 8 7 6 5 4 3 2 1
x, y: 10, 20
xob, yob: 10, 20

```

Program bir önemli noktayı bize göstermektedir: Bir bildirim bölgesini kullanmakla diğerinin üzerine bindirilmez. Bir bildirim bölgesini görünür kildiğınızda, sadece etkin olan diğer bildiri bölgelerinin alanına isimlerini ekler. Böylelikle program sona erdiğinde std, firstNs ve secondNS bildirim bölgeleri global bildirim bölgelerine eklenmiştir.

2. Belirtildiği gibi, bir bildirim bölgesi bir dosya içerisinde dosyalara ayrılabilir. İçeriği eklenebilir olur. Şu örneği ele alalım:

```

// Bildirim bölgeleri eklenebilir.
#include <iostream>
using namespace std;

namespace Demo {
 int a; // Demo bildirim bölgesinde
}
int x; // Bu global bildirim bölgesinde

namespace Demo {
 int b; // Burası da Demo bildirim bölgesi
}

int main()
{
 using namespace Demo;
 a = b = x = 100;
 cout << a << " " << b << " " << x;

 return 0;
}

```

Burada **Demo** bildirim bölgesi hem **a**'yı hem de **b**'yı tutmaktadır. **x** ise bölgenin dışındadır.

3. Anlattığım gibi, Standart C++ tüm kütüphancyı **std** denen kendi bildirim bölgesinde tanımlar. İşte kitaptaki çoğu programın aşağıdaki satırı içermesinin sebebi budur.

```
using namespace std;
```

Bu ifade **std** bildirim bölgesinin mevcut bildirim bölgesine getirilmesini sağlar. Bu da size, kütüphanede tamamlanan fonksiyon sınıf isimlerine **std::** operatörü kullanmadan erişme imkanını verir.

Elbette eğer isterseniz, her ismi **std::** ile açık şekilde belirleyebilirsiniz. Örneğin aşağıdaki program, kütüphancyı global bildirim bölgesine almıyor:

```
// açık std:: operatörü ile 'belirleme'
#include <iostream>

int main()
{
 double val;

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

 std::cin >> val;

 std::cout << "This is your number: ";
 std::cout << val;

 return 0;
}
```

Burada **cout** ve **cin** kendi bildirim bölgeleri tarafından belirlenmiştir. Yani standart çıkışa bir şey yazmak istiyorsanız **std::cout**, standart girişten okumak istiyorsanız **std::cin** ifadesini kullanmak zorundasınız.

Programınız Standart C++ kütüphanesini sınırlı olarak kullanacaksa onu global bildirim bölgesine çekmek istemeyebilirsiniz. Ancak eğer programınız kütüphane isimlerine yüzlerce referans içerecekse, mevcut bildirim bölgенize **std**'yi eklemek her ifadeyi tek tek **std::** ile belirlemekten çok daha kısa bir yol olacaktır.

4. Eğer standart kütüphaneden az sayıda isim kullanacaksanız yine **using** ifadesini tek tek vererek sadece kullanacağınız isimleri belirtmek daha etkin bir çözüm olacaktır. Bu sayede o isimlere yine **std::**'yi kullanmadan erişebilirsiniz. Ancak bu şekilde tüm kütüphaneyi bildirim değil sadece o isimleri bildiri bölgesine çekmiş olursunuz. Size bir örnek vereyim:

```
// global bildirim bölgesine birkaç isı getirme
#include <iostream>

// cout ve cin'e erişim kazanma
using std::cout;
```

```

using std::cin;
int main()
{
 double val;
 cout << "Enter a number: ";
 cin >> val;
 cout << "This is your number: ";
 cout << val;

 return 0;
}

```

Burada **cin** ve **cout** doğrudan kullanılabilir. Fakat diğer **std::** bildiri bölgesi görünürlüğe getirilmemiştir.

5. Açıkladığım gibi, orijinal C++ kütüphanesi global bildirim bölgesinde tanımlanmıştı. Eğer eski C++ programlarını dönüştürüyorsanız ya **using namespace std** ifadesini eklemeli ya da her kütüphane üyesi referansına **std::** ifadesini eklemelisiniz. Bu işlem özellikle eski .h başlıklarınızı yeni tip başlıklarla değiştiriyorsanız önemlidir. Unutmayın eski .h başlıkları içeriklerini global bildirim bölgesine koymalıydınız. Yeni tip başlıklar ise içeriğini **std** bildirim bölgесine koymalıdır.
6. C'de eğer bir global ismin, tanım bölgesini bildirdiği bir dosyaya sınırlamak istiyorsanız, ismini **static** olarak bildirmeniz gerekiyor. Örneğin bir programın parçası olan aşağıdaki iki dosyayı ele alın:

```

Dosya Bir
static int counter;
void f1() {
 counter = 99; // OK
}

```

```

Dosya İki
extern int counter;
void f2() {
 counter = 10; // hata
}

```

İkinci hata var çünkü **counter** birincisinde tanımlanmıştır ve Dosya Bir'de kullanılır. Dosya İki'de **counter**, **extern** olarak belirtilse bile halen kullanılamaz. Her kullanma girişimi bir hata doğurur. Dosya Bir'de **Counter**'ın önüne **static** ifadesini koymakla, programcı onun tanı alanını o dosyaya sınırlamıştır.

C++'da **static** global bildirimleri kullanmak halen geçerli olsa da, bu işlemi gerçekleştirmenin daha iyi bir yolu isimlendirilmemiş bildirim bölgesi kullanmaktır. Aşağıda bir örneğini hazırladım:

```

Dosya Bir
namespace {
 int counter;
}
void f1() {
 counter = 99; // OK
}

```

```

Dosya İki
extern int counter;
void f2() {
 counter = 10; // hata
}

```

Burada da **counter** Dosya Bir'de sınırlanmıştır. Static yerine isimlendirilmemiş bildiri kullanımını standart C++ tarafından tavsiye edilmektedir.

## Aliştırmalar

- Bölüm 9'da verilen bu örneği `using namespace std` ifadesini kullanmayacak şekilde değiştirin.

```
// Boşlukları 's' e çevir
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
 if(argc!=3) {
 cout << "Usage: CONVERT <input> <output>\n";
 return 1;
 }
 ifstream fin(argv[1]); // giriş dosyasını aç
 ofstream fout(argv[2]); // çıkış dosyasını oluştur
 if(!fout) {
 cout << "Cannot open output file.\n";
 return 1;
 }
 if(!fin) {
 cout << "Cannot open input file.\n";
 return 1;
 }
 char ch;
 fin.unsetf(ios::skipws); // boşlukları atla
 while(!fin.eof()) {
 fin >> ch;
 if(ch==' ') ch = '|';
 if(!fin.eof()) fout << ch;
 }
 fin.close();
 fout.close();
 return 0;
}
```

- İsimlendirilmemiş bildirim bölgesinin işlevini açıklayın.
- `using` iki formu arasındaki farkı belirtin.
- Bu kitaptaki bir çok programın neden `using` ifadesini içerdigini açıklayın. Bir alternatif açıklayın.
- Oluşturduğunuz tekrar edilebilir kodun neden kendi bildirim bölgesine koymak isteyebileceğinizi açıklayın.

## 13.2. Bir Dönüşüm Fonksiyonu Oluşturma

Bazı zamanlar bir tip nesneyi diğer tipten bir nesneye dönüştürmek faydalı olabilir. Böyle bir dönüşümü aşırı yüklenmiş operatör ile gerçekleştirmek mümkün olsa da, biraz daha basit (ve iyi) bir yol var. Dönüşüm fonksiyonu kullanmak (conversion fonksiyon). Bir *dönüşüm fonksiyonu* bir nesneyi genelde C++ iç tiplerinden olan değer uyumlu bir diğer nesneye dönüştürür. Ashında dönüşüm fonksiyonu, bir nesneyi, nesnenin kullandığı ifadenin

tipine göre uyumlu bir değere kendiliğinden dönüştürür. Dönüşüm fonksiyonunun genel halini aşağıda sizler için hazırladım:

```
operator type() { return value; }
```

Buradaki *type* dönüştürülecek hedef tipi, *value* ise dönüşüm tamamlandıktan sonraki nesnenin değerini gösteriyor. Dönüşüm fonksiyonları *type* tipinde bir değer döndürüyor. Herhangi bir parametre belirtilmez. Dönüşüm fonksiyonu dönüşümü hangi sınıf için gerçekleştiriyorsa o sınıfın bir üyesi olmalıdır.

Örneklerin birazdan göstereceği gibi dönüşüm fonksiyonları, bir nesnenin değerini diğer bir tipe dönüştürürken, C++'taki diğer metodlara nazaran daha anlaşılır bir yaklaşım sunuyor. Çünkü işlem nesnenin doğrudan, hedef tipi de içeren ifadeye dahil edilmesini sağlıyor.

## Örnekler

1. Aşağıdaki programda, **coord** sınıfı bir nesneyi tamsayıya dönüştüren bir dönüşüm fonksiyonu içeriyor. Bu durumda, fonksiyon iki koordinatın çarpını döndürüyor. Ancak, özel uygulamanıza uygun olan her dönüşümme izin veriliyor.

```
// Basit bir dönüşüm programı.
#include <iostream>
using namespace std;

class coord {
 int x, y;
public:
 coord(int i, int j) { x = i; y = j; }
 operator int() { return x*y; } // Dönüşüm fonksiyonu
};

int main()
{
 coord o1(2, 3), o2(4, 3);
 int i;

 i = o1; // Kendiliğinden tamsayıya dönüştür
 cout << i << '\n';

 i = o2 + o1; // o2'yi tamsayıya dönüştür
 cout << i << '\n';

 return 0;
}
```

Bu program 6 ve 112 görüntüler. Bu örnekte, **o1**'in bir tamsayıya atandığında ve **o2**'nin daha büyük bir tamsayı ifadesinin parçası olduğunda dönüşüm fonksiyonun çağrıldığına dikkat edin. Belirtildiği gibi dönüşüm fonksiyonu kullanarak, oluşturduğunuz sınımların C++'ın normal ifadelerine, bir grup aşırı yüklenmiş karışık operatör fonksiyon hazırlamadan, entegre edilmesine imkan tanıyorsunuz.

2. Dönüşüm fonksiyonlara ilişkin bir örnek daha hazırladım. Burada **strtype** tipinden bir katar **str**'ye bir karakter işaretçisi olacak şekilde dönüştürülüyor.

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

class strtype {
 char str[80];
 int len;
public:
 strtype(char *s) { strcpy(str, s); len = strlen(s); }
 operator char *() { return str; } // char*'a dönüştür
};

int main()
{
 strtype s("This is a test\n");
 char *p, s2[80];

 p = s; // char '*'a dönüştür
 cout << "Here is string: " << p << '\n';
 // fonksiyon çağrısında char '*'a dönüştür
 strcpy(s2, s);
 cout << "Here is copy of string: " << s2 << '\n';

 return 0;
}
```

Bu program aşağıdaki çıktıyi verir.

```
Here is string: This is a test
Here is copy of string: This is a test
```

Gördüğünüz gibi sadece **s** nesnesi **p**'ye (**char\*** tipinde) atandığında dönüşü fonksiyonu çağrılmıyor. Aynı zamanda **s**, **strcpy()**'nin bir parametresi olarak kullanıldığından da çağrılmıyor. Unutmayın **strcpy()**'nin prototipi şu şekildedir:

```
char *strcpy(char *s1, const char *s2);
```

Kalıpta gösterilen **s2**'nin **char\*** tipinden olmasından dolayı **char\***'a olan dönüşüm fonksiyonu kendiliğinden çağrılr. Bu olay size sınıflarınızı C++'ın standart kütüphane fonksiyonlarına nasıl entegre edeceğinizi açık şekilde gösteriyor.

## Aliştırmalar

1. Örnek 2'de geçen **strtype** sınıfını kullanarak, **int** tipine dönüşüm yapan bir dönüşüm fonksiyonu hazırlayın. Bu dönüşümde **str**'de tutulan metnin uzunluğunu döndürün. Fonksiyonunuzun çalıştığını basit bir programla gösterin.
2. Size aşağıdaki sınıf veriliyor:

```
class pwr {
 int base;
```

```
int exp;
public:
 pwr(int b, int e) { base = b; exp = e; }
 // tamsayıya dönüştürün
};
```

pwr tipinden bir nesneyi tamsayı tipine dönüştüren bir program hazırlayın. Fonksiyonun **base**<sup>exp</sup> döndürmesini sağlayın.

### 13.3. static Sınıf Üyeleri

Bir sınıf üye değişkeninin **static** olarak bildirilmesi mümkündür. **Static** üye değişkenleri kullanarak birkaç teknik problemden kurtulabilirsiniz. Bir üye değişkenini **static** olarak bildirdiğinizde o değişkenin sadece bir kopyasının bulunmasını sağlıyorsunuz. O sınıfın kaç kopya ürettiğinizin bir önemi olmuyor. Her nesne basitçe bir değişkeni paylaşıyor. Unutmayın normal üye değişkeni için her nesne oluşturulduğunda o değişkenin yeni bir kopyası oluşturulur. O kopyaya sadece o nesne üzerinden erişilebilir. (Yani normal değişkenler kullanıldığında her nesne kendi değişken kopyalarını kullanır) Ancak **static** üye değişkeninin sadece bir kopyası vardır ve kendi sınıfının tüm nesneleri o değişkeni paylaşır. Ayrıca aynı **static** değişken, static üyesi olan sınıfın türetilenler(sınıf) tarafından da kullanılır.

İlk düşündüğünüzde alışılmadık gelebilir. Bir **static** üye değişkeni kendi sınıfından herhangi bir nesne oluşturulmadan daha önce bulunur. Aslında, bir **static** sınıf üyesi bildirdiği sınıf ile tam alamı sınırlanmış global bir değişkendir. Birazdan gelecek örneklerde bakığınızda göreceksiniz ki gerçekte nesnelerden bağımsız olarak **static** üye değişkenine erişmek mümkündür.

Sınıf içerisinde **static** veri üyesi bildirdiğinizde onu tanımlıyor olmazsınız. Bunun yerine bu tanımlamayı sınıf dışında başka bir yerde sağlamalısınız. Bunu gerçekleştirmek için **static** değişkenini hangi sınıfa ait olduğunu belirlemek açısından kavram çözümleme operatörü kullanarak yeniden bildirirsınız.

Bütün **static** üye değişkenleri normalde 0 ile başlatılır. Ancak eğer isterseniz **static** sınıf değişkenine kendi seçtiğiniz bir ilk değeri verebilirsiniz.

Şunu aklınızda bulundurun, C++'ın **static** değişkenlerini desteklemesinin temel sebebi global değişken ihtiyacını önlemektir. Global değişkenlere bağlı olan sınıflar hemen hemen her zaman C++ ve OOP için önemli olan depolama kurallarını çiğnerler.

Bir üye fonksiyonun **static** olarak bildirilmesi de mümkündür fakat bu kullanım pek yaygın değildir. **static** olarak bildirilen bir üye fonksiyonun sadece kendi sınıfının **static** üyelerine erişebilir. (Elbette, bir **static** fonksiyon **static** olmayan global veri ve fonksiyonlara da erişebilir) **static** üye fonksiyonun **this** işaretçisi yoktur. Sanal **static** üye fonksiyonlarına izin verilmez. Ayrıca **static** üye fonksiyonu sınıfından bir nesne tarafından uygulanabilir veya herhangi bir nesneden bağımsız olarak sınıf ismi ve kapsam çözümleme operatörü ile çağrılabilir.

## Örnekler

- Size static üye değişkenini tanıtmak için bir örnek hazırladım:

```
// static üye değişkeni örneği.
#include <iostream>
using namespace std;

class myclass {
 static int i;
public:
 void seti(int n) { i = n; }
 int geti() { return i; }
};

// myclass::i tanımı. i halen myclass için private'dır
int myclass::i;

int main()
{
 myclass o1, o2;
 o1.seti(10);

 cout << "o1.i: " << o1.geti() << '\n'; // 10 gösterir
 cout << "o2.i: " << o2.geti() << '\n'; // Bu da 10 gösterir

 return 0;
}
```

Bu program aşağıdaki çıktıyi verir.

```
o1.i: 10
o2.i: 10
```

Bu programa baktığımızda aslında sadece **o1** nesnesinin **static** üye **i**'yi ayarladığını görüyoruz. Ancak **i**, hem **o1** hem de **o2** tarafından(aynı zamanda **myclass** sınıfının her üyesi tarafından) paylaşıldığı için her ikisi de sonucu görüntülemek için **geti()**'yi çağrıyor.

**myclass** içerisinde **i**'nin nasıl bildirildiğine dışında ise nasıl tanımlandığına dikkat edin İkinci adı ise **i** için depo tanımlanmasını garantiler. Teknik olarak bir sınıf bildirisi sadece bir bildirimdir. Çünkü **static** veri üyesi belleğin sadece o üye için ayrılmışlığını temsil eder. Depolama için yer ayırmının meydana gelmesi için ayrı bir tanımlamaya gerek duyulur.

- static** üye değişkeni o sınıfın nesnelerin oluşturulmasından daha önce bulunduğu program içerisinde herhangi bir nesneden erişilebilir. Örneğin, bir önceki programın diğer bir çeşidi olan aşağıdaki program **i**'nin değerini herhangi özel bir nesneye referans kullanmadan 100'e ayarlıyor. Kapsam çözümleme operatörünün ve sınıf isminin **i**'ye erişmek için nasıl kullanıldığına dikkat edin.

```
// static üye değişkeninin nesneden bağımsız olarak kullanılması.
#include <iostream>
using namespace std;
```

```

class myclass {
public:
 static int i;
 void seti(int n) { i = n; }
 int geti() { return i; }
};

int myclass::i;

int main()
{
 myclass o1, o2;

 // set i directly
 myclass::i = 100; // nesne referans gösterilmemiştir.

 cout << "o1.i: " << o1.geti() << '\n'; // 100 görüntüler
 cout << "o2.i: " << o2.geti() << '\n'; // yine 100 görüntüler

 return 0;
}

```

i, 100'e ayarlandığından program şu çıkışı verir.

```

o1.i: 100
o2.i: 100

```

3. **static** sınıf değişkeninin en genel kullanımı paylaşılan kaynağa olan erişimin koordine etmektir. Örneğin bir disk dosyası, yazıcı veya bir ağ sunucusu olabilir. Daha önceki programlama deneyimlerinizden biliyor olabilirsiniz, paylaşılan kaynağa erişimi düzenlemek olayları sıralamaya benzer. **Static** üye değişkenlerin paylaşılan kaynakları nasıl düzenledikleri anlamak için aşağıdaki programı inceleyin. Program kendisi **static** karakter dizisi olan **outbuf** adında genel bir tampon içeren **output** adında bir sınıf oluşturur. Bu tampon **outbuf()** tarafından gönderilen çıkışı almak için kullanılır. Bu fonksiyon **str**'nin içeriğini bir seferde bir karakter olmak üzere gönderir. Bu işlemi önce tampona erişimi yaparak daha sonra **str**'nin içerisindeki bütün karakterleri göndererek gerçekleştirir. Çıkışı bitirdikten sonra diğer nesnelerin tampona erişimini geçerli kılar. Bu işlemleri aşağıdaki kodu çalışırken takip edebilmelisiniz.

```

// Paylaşılmuş kaynak örneği
#include <iostream>
#include <cstring>
using namespace std;

class output {
 static char outbuf[255]; // Paylaşılan kaynak budur
 static int inuse; // Eğer 0 ise tampon kullanılabilir
 static int oindex; // outbuf'un indeksi
 char str[80];
 int i; // str içindeki bir sonraki karakterin indeksi
 int who; // nesneyi tanır, 0'dan büyük olamalı
public:
 output(int w, char *s) { strcpy(str, s); i = 0; who = w; }

 void putchar(char c)
 {
 if (inuse == 0)
 return;
 if (c == '\n')
 i = 0;
 else
 outbuf[oindex] = c;
 oindex++;
 }

 void flush()
 {
 if (inuse == 0)
 return;
 if (i > 0)
 outbuf[i] = '\0';
 cout << outbuf;
 inuse = 0;
 }
};

```

```

/* Tampon bekleniyorsa bu fonksiyon -1 döndürür,
 çıkış bittiğse 0 döndürülür
 ve eğer hala tampon kullanılıyorsa who döndürülür */
int putbuf()
{
 if(!str[i]) { // çıkış bitti
 inuse = 0; // tamponu bırak
 return 0; // sinyal kesilmesi
 }
 if(!inuse) inuse = who; // tamponu al
 if(inuse != who) return -1; // başkası tarafından kullanılmış
 if(str[i]) { // hala gönderilecek karakter var
 outbuf[oindex] = str[i];
 i++;
 oindex++;
 outbuf[oindex] = '\0'; // her zaman boş sonlandırmalı tut
 return 1;
 }
 return 0;
}
void show() { cout << outbuf << '\n'; }
;

char output::outbuf[255]; // bu paylaşılan kaynak
int output::inuse = 0; // eğer 0 ise tampon kullanılabilir
int output::oindex = 0; // outbuf'un indeksi

int main()
{
 output o1(1, "This is a test"), o2(2, " of statics");
 while(o1.putbuf() | o2.putbuf()) ; // karakter yollar
 o1.show();

 return 0;
}

```

4. **static** üye fonksiyonlarının sınırlı uygulamaları vardır. Fakat bunların iyi bir kullanımım henüz hiç bir nesne oluşturulmadıkken **private static** versine "ön hazırlama" uygulanmasıdır. Örneğin bu tamamen geçerli bir C++ programıdır.

```

#include <iostream>
using namespace std;

class static_func_demo {
 static int i;
public:
 static void init(int x) {i = x;}
 void show() {cout << i;}
};
int static_func_demo::i; // i'yi tanımla

int main()
{
 // nesne oluşturulmadan önce static veriyi hazırla
 static_func_demo::init(100);
 static_func_demo x;
 x.show(); // 100 görüntüle

 return 0;
}

```

Burada `i`, `static_func_demo` nesnesi bulunmadan önce `init()`'e yapılan çağrı ile hazırlanmıştır.

## Aliştırmalar

1. Örnek 3 üzerinde yeniden çalışarak, hangi nesnenin karakter çıkışı gerçekleştirdiğini ve hangilerinin tamponun kullanımını yüzünden karakter çıkışının bloke edildiğini göstermesini sağlayın.
2. `static` üye değişkeninin ilginç bir kullanımı da, zamanda verilen herhangi bir noktada, bir sınıfın mevcut olan nesne sayılarının izinin tutulmasıdır. Bu gerçekleştirmenin yolu `static` üye değişkeninin, her sınıf yapılandırıcısının çağrılmamasında 1 artrılması ve her sınıf yok-edicisinin çağrılmamasında 1 eksiltilmesidir. Anlatılan şekilde bir yapı oluşturun ve çalıştığını gösterin.

## 13.4. const Üye Fonksiyonları ve `mutable`

Sınıf üye fonksiyonları `const` olarak bildirilebilir. İşlem yapıldığında bu fonksiyon onu uyaran nesneyi değiştiremez. Aynı zamanda bir `const` nesne `const` olmayan bir üye fonksiyonu uyaramaz. Ancak, bir `const` üye fonksiyonu `const` veya `const` olmayan nesneler tarafından çağrılabilir. Bir üye fonksiyonunu `const` olarak belirlemek için aşağıdaki form örneğini kullanın:

```
class X {
 int some_var;
public:
 int f1() const; // const üye fonksiyonu
};
```

Gördüğünüz gibi `const` fonksiyonun parametre bildiriminden hemen sonra geliyor. Bazı zamanlar bir sınıfın, bir `const` fonksiyonunun değiştirebileceği, bir veya daha fazla üyesi olabilir. Fakat siz fonksiyonun kendi herhangi bir nesnesini değiştirmesi istemiyorsunuz. Bu işlemi `const-ness`'in üzerine bindiren `mutable`'ı kullanarak gerçekleştirebilirsiniz. Yani bir `mutable` üye `const` üye fonksiyonu tarafından değiştirilebilir.

## Örnekler

1. Bir üye fonksiyonunu `const` olarak bildirmek onu uyaran nesneyi değiştirmesini engeller. Örneğin aşağıdaki programa göz atın.

```
/*const üye fonksiyonlarını gösterir. Bu program derlenmez. */
#include <iostream>
using namespace std;

class Demo {
 int i;
public:
```

```

int geti() const {
 return i; // ok
}
void seti(int x) const {
 i = x; // hatalı
}
}

int main()
{
 Demo ob;
 ob.seti(1900);
 cout << ob.geti();

 return 0;
}

```

Bu program derlenmez çünkü **seti()** **const** olarak bildirilmiştir. Bu demektir ki uyanan nesneyi değiştirmeye izin verilmeyez. Program **i**'yi değiştirmeye çalıştığından, hata meydana gelecektir. Bunun aksine, **geti()**, **i**'yi değiştirmeyi denemediğinden tamamen geçerlidir.

2. Seçilen üyelerin **const** üye fonksiyonu tarafından değiştirilmesine izin vermek için, onları **mutable** olarak belirtin. İşte size bir örnek:

```

// mutable'ı tanıtır.
#include <iostream>
using namespace std;

class Demo {
 mutable int i;
 int j;
public:
 int geti() const {
 return i; // ok
 }
 void seti(int x) const {
 i = x; // Şimdi OK.
 }

 /* Aşağıdaki fonksiyon derlenmez.
 void setj(int x) const {
 j = x; // Still Wrong!
 }
 */
}

int main()
{
 Demo ob;
 ob.seti(1900);
 cout << ob.geti();
 return 0;
}

```

Burada **i**, **mutable** olarak belirtilmiştir. Bu sebeple **seti()** fonksiyonu tarafından değiştirilebilir. Ancak, **j** **mutable** olmadığından **setj()** onun değerini değiştiremez.

## Aliştırmalar

1. Aşağıdaki program kurulan zaman bittiğinde ses çikaran basit bir zamanlayıcıyı temsil eder. Zaman aralığını ve artışı **countdown** nesnesi oluşturulduktan belirleyebilirsiniz. Aslında aşağıda verilen şekliyle program derlenmez. Bunu düzeltmek size düşüyor:

```
// Programda bir hata var
#include <iostream>
using namespace std;

class CountDown {
 int incr;
 int target;
 int current;
public:
 CountDown(int delay, int i=1) {
 target = delay;
 incr = i;
 current = 0;
 }

 bool counting() const {
 current += incr;
 if(current >= target) {
 cout << "\a";
 return false;
 }
 cout << current << " ";
 return true;
 }
};

int main()
{
 CountDown ob(100, 2);

 while(ob.counting());
}

return 0;
}
```

2. Acaba bir **const** üye fonksiyonu **const** olmayan bir fonksiyonu çağrıabilir mi? Neden?

## 13.5. constructor'lara Son Bir Bakış

Yapıldırıcılar (constructor) kitabın başlarında ele alındıysa da bilinmesi gereken bir kaç nokta daha kaldı. Aşağıdaki program üzerinde biraz düşünün.

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

class myclass {
 int a;
```

```

public:
 myclass(int x) { a = x; }
 int geta() { return a; }
};

int main()
{
 myclass ob(4);
 cout << ob.geta();

 return 0;
}

```

Burada **myclass**'ın yapılandırıcısı bir parametre alıyor. **ob**'nin **main()**'de nasıl bildirilmesine dikkat edin. **ob**'den sonra parantez içinde belirtilen 4 değeri, **a**'yı hazırlamak için kullanılan **myclass()**'ın **x** parametresine gönderilen argümandır. Kitabın başından beri kullanılan hazırlama formu buydu. Ancak, başka bir alternatif var. Örneğin aşağıdaki ifade de **a**'ya 4 değeri vererek hazırlıyor.

```
myclass ob = 4; // kendiliğinden myclass(4)'e dönüştürülür
```

Açıklamanın gösterdiği gibi hazırlamanın formu kendiliğinden **myclass** yapılandırıcısını argümanı 4 olarak çağrıyor. Yani önceki ifade derleyici tarafından sanki şu şekilde yazılmış gibi ele alındı:

```
myclass ob(4);
```

Genelde, bir argümanı olan bir yapılandırıcınız olduğunda nesneyi hazırlamak, ya **ob(x)** ya da **ob=x** ifadesini kullanabilirsiniz. Bunun sebebi, ne zaman bir argümanı olan bir yapılandırıcı oluşturduğunuzda kapalı olarak o argümanın tipinden, sınıfın tipine bir dönüşüm biçimini oluşturmuş oluyorsunuz.

Eğer kapalı dönüşümlerin yapılmasını istemiyorsanız, **explicit** kullanarak engellemeye çalışabilirsiniz. **explicit** belirteci sadece yapılandırıcılara uygulanır. **explicit** olarak belirtilen bir yapılandırıcı, sadece hazırlamanın normal yapılandırıcı notasyonunu kullandığında kullanılır. Herhangi bir şekilde kendiliğinden dönüşüm gerçeklemez. Örneğin, eğer **myclass** yapılandırıcısı **explicit** olarak bildirilirse kendiliğinden dönüşüm olanağı sağlanmaz. İşte **myclass()**'ın **explicit** olarak bildirilmesi:

```

#include <iostream>
using namespace std;

class myclass {
 int a;
public:
 explicit myclass(int x) { a = x; }
 int geta() { return a; }
};

```

Artık sadece şu yapılandırma biçimine izin verilir.

```
myclass ob(110);
```

## Örnekler

- Bir sınıfta birden fazla dönüştüren yapılandırıcı olabilir. Örneğin **myclass**'ın şu şablonunu ele alalım.

```
#include <iostream>
#include <cstdlib>
using namespace std;
class myclass {
 int a;
public:
 myclass(int x) { a = x; }
 myclass(char *str) { a = atoi(str); }
 int geta() { return a; }
};

int main()
{
 myclass ob1 = 4; // myclass(4)'e dönüştürür.
 myclass ob2 = "123"; // myclass("123")'e dönüştürür;

 cout << "ob1: " << ob1.geta() << endl;
 cout << "ob2: " << ob2.geta() << endl;

 return 0;
}
```

Her iki yapılandırıcı da değişik tipte argüman kullanıldığından (tabi ki öyle olmalı), her hazırlama ifadesi kendiliğinden eşdeğer yapılandırıcı çağrısına dönüştürülür.

- Yapılandırıcının ilk argümanının tipinden, yapılandırıcının kendisine çağrıya kendiliğinden dönüşüm ilginç fikirleri doğuruyor. Örneğin Örnek 1'deki **myclass**'ı ele alalım. Aşağıdaki **main()** fonksiyonu **int** ve **char\***'dan **ob1** ve **ob2**'ye yeni değerler atamak için dönüşümleri kullanıyor.

```
int main()
{
 myclass ob1 = 4; // myclass(4)'e dönüştürür.
 myclass ob2 = "123"; // myclass("123")'e dönüştürür;

 cout << "ob1: " << ob1.geta() << endl;
 cout << "ob2: " << ob2.geta() << endl;

 // yeni değerler atamak için kendiliğinden dönüşümleri kullan
 ob1 = "1776"; // ob1 = myclass("1776")'e dönüştür;
 ob2 = 2001; // ob2 = myclass(2001)'e dönüştür;

 cout << "obi: " << ob1.geta() << endl;
 cout << "ob2: " << ob2.geta() << endl;

 return 0;
}
```

- Az önce gösterilen dönüşümleri engellemek için yapılandırıcıları aşağıda gösterildiği gibi **explicit** olarak belirtmelisiniz.

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

class myclass {
 int a;
public:
 explicit myclass(int x) { a = x; }
 explicit myclass(char *str) { a = atoi(str); }
 int geta() { return a; }
};
```

## Alişturmalar

1. Örnek 3'te eğer sadece `myclass(int)` `explicit` yapılsaydı `myclass(char*)` halen kapalı dönüşümlere izin verir miydi? (İpucu: Deneyin.)
2. Aşağıdaki kod parçası işler mi?

```
class Demo {
 double x
public:
 Demo(double i) { x = i; }
 // ...
};

// ...
Demo counter = 10;
```

3. `explicit` anahtar kelimesinin kullanımının dezavantajını belirtiniz. ( Başka bir deyişle, bazı durumlarda kapalı yapılandırıcı dönüşümlerinin neden C++'ın dikkat çekici yeniliklerinden olduğunu açıklayın)

## 13.6. Bağlantı Belirticilerinin Kullanılması ve `asm` Anahtar Kelimesi

C++, diğer dillerle iletişime geçebilmek için iki önemli yenilik içerir. Bunlardan bir tanesi derleyiciye C++ programınızdaki bir veya daha fazla fonksiyonun diğer bir dille bağlantı kuracağını belirten *bağlantı belirteç (linkage specifiers)*'leridir. Bu dilde değişik bir isimlendirme yaklaşımı, parametre gönderme, yoğun çözümleme ve benzeri farklılıklar olabilir. İkinci `asm` anahtar kelimesidir. Bu kelime size makine dili komutlarını C++ kodunun içerişine gömmenizi sağlar. Her ikisi de burada tartışılacaktır.

Normalde, C++ programındaki bütün fonksiyonlar derlenmiş ve C++ fonksiyonları olarak bağlanmıştır. Ancak, C++ derleyicisine bir fonksiyona bağlanması söyleyebilir ve öylece diğer tipte dillerle uyumlu olabilir. Bazıları size fonksiyonları Pascal, Ada veya Fortran gibi dillerle bağlamayı sağlar. Bir fonksiyonun farklı diller için bağlanabilmesi için, aşağıdaki genel formu kullanın:

```
extern "language" fonksiyon-prototype;
```

Buradaki *language* belirtilen dilin bağlantı kurmasını istediğiniz dilin ismidir. Eğer bir fonksiyondan daha fazlasının bağlantı kurmasını istiyorsanız aşağıdaki formu kullanabilirsiniz:

```
extern "language" {
 fonksiyon-prototype
}
```

Bütün bağlantı tanımlamaları global olmalıdır: Bir fonksiyonun içerisinde bulunamaz. Bağlantı tanımlamlarının en genel kullanımı C++ programlarını C koduna bağlarken ortaya çıkar. "C" bağlantısı tanımlanırsa derleyicinizin fonksiyonların ismini gömülüştür tip bilgisiyle düzenlemesini engellemiş olursunuz. Çünkü C++ fonksiyonları aşırı yükleme ve üye fonksiyonlar oluşturma yeteneğinden dolayı, fonksiyonun bağlantı ismine genellikle tip bilgisi de eklenmiş oluyor. C aşırı yükleme ve üye fonksiyonları desteklemediğinden karışık ismi tanıyamaz. "C" bağlantısı bu problemleri sizden uzak tutar.

Makine dili rutinlerini genellikle C++ programına eklemek mümkün olsa da makine dilini biraz daha kolay şekilde kullanmanın başka bir yolu var. C++, bir C++ fonksiyonuna makine dili kodu gömmenize imkan tanıyan, **asm** denen anahtar bir kelime içerir. O komutlar olduğu gibi derlenir. Satır içi makine kodu kullanmanızın avantajı, programınızın tamamının C++ şeklinde tamamlanması ve ayrıca makine dili için herhangi hariçten bir dosya kullanmanızın gerekmemesi. **asm** anahtar kelimesinin genel kullanımı aşağıdaki gibidir:

```
asm ("op-code");
```

*op-code* programınıza gömülü makine dili kodunun kendisi. Belirtilmesi gereken önemli bir nokta var: Bazı derleyiciler aşağıda verdiğim üç çeşit **asm** ifadesini desteklerler.

```
asm op-code;
asm op-code newline
asm {
 komut akışı
}
```

Burada *op-code* tırnakla kapatılmıyor. Çünkü gömülü makine dili bağımsız olarak belirtilmiştir. Bu konu hakkında ek bilgi için derleyicinizin dokümanlarına bakabilirsiniz.

**NOT** Microsoft Visual C++ makine dili kullanmak için **\_asm** ifadesini kullanır, **asm**'nin kullanımı ile benzer.

## Örnekler

1. Bu program **func()**'i C++ yerine C olarak bağlar.

```
// Bağlantı belirtecini tanıt.
#include <iostream>
using namespace std;

extern "C" int func(int x); // C fonksiyonu olarak bağla
```

```
// bu program artık C olarak bağlanır.
int func(int x)
{
 return x/3;
}
```

Bu fonksiyon şimdi bir C derleyicisi tarafından derlenen kodla bağlanabilir.

- Aşağıdaki kod derleyiciye `f1()`, `f2()` ve `f3()`'ün C fonksiyonları şeklinde bağlanması gerektiğini söyler:

```
extern "C" {
 void f1();
 int f2(int x);
 double f3(double x, int *p);
}
```

- Bu kod bazı makine dili komutlarını `func()`'ın içerisinde gömer:

```
// Bu fonksiyonu denemeyin!
void func()
{
 asm("mov bp, sp");
 asm("push ax");
 asm("mov cl, 4");
 // ...
}
```

**HATIRLATMA** Sını içi makine dilini boşıyla kullanmak için makine dili ile programlamaya hakim olmalısınız. Makine dili kullanımını kavramak için de derleyicinizin dokümanlarına bakabilirsiniz.

## Aliştirma

- Kendi kendinize derleyicinizin dokümanında bulunan bağlantı tanımlamaları ve makine dili arabirimini kısımlarını çalışın.

### 13.7. Dizi Tabanlı I/O

Konsol ve dosya I/O'suna ek olarak C++, giriş veya çıkış aygıtları için karakter dizileri kullanan oldukça geniş bir grup fonksiyon içerir. C++'in dizi tabanlı I/O'su kavram olarak C'de bulunan dizi tabanlı I/O ile paralellik gösterse de (özellikle C'nin `sscanf()` ve `sprintf()` fonksiyonları) C++'inki daha esnek ve kullanışlıdır. Çünkü kullanıcı tarafından tamamlanabilir tiplerin içerisinde entegre edilebilmesini sağlar. Dizi tabanlı I/O'nun bütün özelliklerini burada açıklamak mümkün olmasa da bundan sonra en önemli olanları ve sıkça kullanılanları inceleyeceğiz.

En başta anlamamız gereken, dizi tabanlı I/O halen akımlar üzerinden işlemekte olduğunu. Bölüm 8 ve 9'da C++ I/O'su hakkında öğrendiğiniz her şey dizi tabanlı I/O için de geçerlidir. Fakat dizi tabanlı I/O'nun tüm avantajlarından yararlanabilmek için birkaç yeni fonksiyon daha öğrenmelisiniz. Bu fonksiyonlar bir akımı belleğin bir bölgesine bağlar. Bu

olay bir kere gerçekleştiğinde bütün I/O işlemleri bildiğiniz gibi I/O fonksiyonları üzerinden meydana gelir.

Dizi tabanlı I/O'yu kullanmaya başlamadan önce **<strstream>** başlığını dosyanıza eklemelisiniz. Bu başlıkta **istrstream**, **ostrstream** ve **strstream** tanımlanmıştır. Bu sınıflar, sırasıyla dizi tabanlı giriş, çıkış ve giriş-çıkış akımları oluştururlar. Taban olarak **ios**'a sahiptirler. Böylece **istream**, **ostream** ve **iostream** içinde bulunan bütün fonksiyonlar ve manipülatörler aynı zamanda **istrstream**, **ostrstream** ve **strstream**'de de bulunmaktadır.

Cıktı vermek amacıyla bir karakter dizisi kullanmak için **ostrstream** constructor'unın aşağıdaki genel formunu kullanın :

```
ostrstream ostr (char *buf, streamszie size, openmode mode = ios::out);
```

Burada *ostr*, *buf* dizisiyle bağlantılı olan akımdır. Dizinin boyutu *size* ile belirtilir. Genelde *mode* çıkışa ayarlanmıştır. Fakat, eğer isterseniz **ios** tarafından tanımlanan herhangi bir çıkış mod bayrağını kullanabilirsiniz. (Ayrıntılar için Bölüm 9'a bakın.)

Dizi bir kere çıkış için açıldığında karakterler dizi dolana kadar depolanmaya devam eder. Dizinin taşmasına izin verilmez. Dizinin taşmasına neden olacak herhangi bir hareket I/O hatasına sebep olur. Diziye kaç tane karakter yazıldığını anlamak için aşağıda gösterilen **pcount( )** fonksiyonunu kullanın :

```
streamszie pcount();
```

Bu fonksiyonu akım ile bağlantılı bir şekilde çağrımalısınız. Fonksiyon, boş sonlandırıcılar da dahil olmak üzere diziye kaç karakter yazıldığını döndürür. Bir diziye giriş amacıyla açmak için **istrstream** constructor'unın şu formunu kullanın:

```
istrstream istr (const char *buf);
```

Burada *buf* giriş için kullanılan diziye bir işaretcidir. Giriş akımına *istr* denilecektir. Giriş diziden okunurken dizinin sonu geldiğinde **eof()** true değerini döndürür. Bir diziye giriş çıkış işlemleri amacıyla açmak için **strstream** constructor'unın şu formunu kullanın:

```
strstream iost (char *buf, streamszie size, openmode mode = ios::in | ios::out);
```

Burada *iost buf* tarafından işaret edilen ve *size* uzunluğundaki diziyi kullanan bir giriş çıkış akımı olacaktır. Bazı C++ kaynaklarında karakter tabanlı akımlarla *char \*streams* şeklinde de karşılaşabiliriz. Daha önce anlatılan bütün I/O fonksiyonlarının dizi tabanlı I/O ile çalıştığını bilmemiz gereklidir. Bunlara ikili I/O ve rasgele erişimli fonksiyonlar dahildir.

#### NOT

Karakter tabanlı akım sınıfları Standard C++ tarafından onaylanmıştır. Bunun anlamı, bunların hala geçerli olduğunu söylemektedir. Fakat C++ dilinin gelecekteki sürümleri bunu desteklemeyebilir. Karakter tabanlı akım sınıflarını bu kitapta ele aldık, çünkü hala geniş çapta kullanılmaktadır. Ancak yeni kodlarda Bölüm 14'te anlatılan **container**'ları kullanmak isteyeceksiniz.

## Örnekler

- Burada dizini çıkış için nasıl açıldığını ve diziye nasıl veri yazıldığını gösteren kısa bir örneği inceleyelim:

```
// Dizi tabanlı çıkış kullanınan kısa bir örnek.
#include <iostream>
#include <strstream>
using namespace std;
int main()
{
 char buf[255]; // tampon gönder

 ostrstream ostr(buf, sizeof buf); // çıkış dizisini aç

 ostr << "Dizi tabanlı I/O uses streams just like ";
 ostr << "'normal' I/O\n" << 100;
 ostr << ' ' << 123.23 << '\n';

 // manipülatörleri de kullanabilirsiniz.
 ostr << hex << 100 << ' ';
 // veya format bayrakları:
 ostr.setf(ios::scientific);
 ostr << dec << 123.23;
 ostr << endl << ends;

 // sonuç katarını göster
 cout << buf;

 return 0;
}
```

Bu program şu çıkışı verir:

```
Dizi tabanlı I/O uses streams just like 'normal' I/O
100 123.23
64 01.2323e+02
```

Gördüğünüz gibi aşırı yüklenmiş I/O operatörleri, mevcut I/O manipülatörleri, üye fonksiyonları ve format bayrakları, dizi tabanlı I/O kullandığınızda tam fonksiyonel hale gelir. (bu işlem kendi sınıflarınız için oluşturduğunuz tüm manipülatörler veya aşırı yüklenmiş I/O operatörleri için de doğrudur.)

Bu program `ends` manipülatörünü kullanarak çıkış dizini manuel olarak boş sonlandırır. Dizi kendiliğinden boş sonlandırılsa da sonlandırılmasa da işletime bağlıdır. Böylelikle eğer uygulamanız için boş sonlanma önemliyse bu işlemi manuel olarak gerçekleştirmek en iyisidir.

- Burada dizi tabanlı giriş için bir örnek hazırladım:

```
// Dizi tabanlı giriş kullanınan bir örnek.
#include <iostream>
#include <strstream>
using namespace std;

int main()
```

```

 char buf[] = "Hello 100 123.125 a";
 istream istr(buf); // giriş dizisini aç

 int i;
 char str[80];
 float f;
 char c;

 istr >> str >> i >> f >> c;

 cout << str << ' ' << i << ' ' << f;
 cout << ' ' << c << '\n';

 return 0;
}

```

Bu program giriş dizisi **buf**ta bulunan verileri okur ve yeniden görüntüler.

3. Şunu aklımızdan çıkarmayın : Bir giriş dizisi akıma bir kere bağlandıktan sonra bir dosya ile aynı şekilde görülür. Örneğin bu program, **buf**'ın içeriğini okumak için **get()** ve **eof()** fonksiyonlarını kullanır:

```

// Dizi tabanlı I/O ile get() and eof()'u gösterir.
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
 char buf[] = "Hello 100 123.125 a";

 istream istr(buf);
 char c;

 while(!istr.eof()) {
 istr.get(c);
 if(!istr.eof()) cout << c;
 }

 return 0;
}

```

4. Bu program dizi üzerinde giriş çıkış işlemleri gerçekleştirir:

```

// Giriş çıkış dizisini göster.
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
 char iobuf[255];

 stringstream iostr(iobuf, sizeof iobuf);

 iostr << "This is a test\n";
 iostr << 100 << hex << ' ' << 100 << ends;
}

```

```

char str[80];
int i;

iosr.getline(str, 79); // '\n' kadar katarı okur
iosr >> dec >> i; // 100'ü oku

cout << str << ' ' << i << '\n';

iosr >> hex >> i;
cout << hex << i;

return 0;
}

```

Program ilk olarak **iobuf**'a çıkış gönderir, daha sonra da onu geri okur. İlk önce **getline()** fonksiyonunu kullanarak "This is a test" satırının tamamını okur. Sonra onluk tabandaki 100 ve onaltılık tabandaki 0x64 değerlerini okur.

## Alıştırmalar

1. Örnek 1'i sonlandırmadan önceye kadar **buf**'a gönderilen karakterlerin sayısını görüntüleyebilecek şekilde değiştirin.
2. Bir dizinin içeriğini diğerine kopyalamak için dizi tabanlı I/O kullanan bir uygulama yazın. (Bu tabi ki bu işlemin gerçekleştirilmesi için kullanılabilecek en verimli yol değildir.)
3. Dizi tabanlı I/O kullanarak kayan nokta içeren bir katarı kendi değerine çeviren bir program yazın.

## Pekiştirme Testi

Bu noktada aşağıdaki alıştırmaları yapabiliyor ve soruları cevaplayabiliyor olmalısınız.

1. **static** üye değişkenini diğer üye değişkeninden farklı kılan nedir?
2. Dizi tabanlı I/O kullanırken programınıza hangi başlığı dahil etmelisiniz?
3. Dizi tabanlı I/O belleği giriş ve/veya çıkış aygıtı olarak kullanmaktadır. Bu durumda C++'in "normal" I/O'su ile aralarında bir fark var mıdır?
4. Size **counter()** adında bir fonksiyon veriliyor. Derleyicinin bu fonksiyonu C için derlemesini sağlayan ifadeyi gösterin.
5. Dönüşüm fonksiyonu ne yapar?
6. **explicit**'in kullanım amacını açıklayın.
7. **const** türü fonksiyonuna konan temel sınıflarla nedir?
8. **namespace**'i açıklayın.
9. **mutable** ne yapar?

## Bütünleştirme Testi

Bu kısım sizin geçmiş bölümlerdeki konular da dahil olmak üzere, bu bölümde geçen konuları ne kadar kavradığınızı kontrol edecektir.

1. Sadece tek bir argüman gerektiren bir constructor otomatik olarak o argümanın tipinden kendi sınıf tipine bir dönüşüm sağlar. Bu özelliğin aşırı yüklenmiş atama operatörünün oluşturulma ihtiyacını ortadan kaldırdığı bir durum var mıdır?
2. Bir `const_cast`, `const` üye fonksiyonu içerisinde bu fonksiyonun kendisini uyanıranı değiştirmesine izin vermek için kullanılabilir mi?
3. Düşünce sorusu : Orijinal C++ kütüphanesi global namespace'de içermekteydi ve eski C++ programlarının tümü bu durumla ilgilenmişti. Bu duruma rağmen kütüphaneyi std namespace'ine taşımanın avantajı nedir?
4. İlk on iki bölümdeki alıştırmalara göz atın. Hangi üye fonksiyonlarının `const` veya `static` yapılabileceğini düşünün. Namespace'in doğru tanımlandığı örnekler var mı?

**BÖLÜM 14**

# **Standart Şablon Kütüphanesine Giriş**

**Standart Şablon Kütüphanesi**  
**Konteyner Sınıfları**  
**Vektörler**  
**Listeler**  
**Haritalar**  
**Algoritmalar**  
**string Sınıfı**

**TEBRİKLER!** Eğer önceki tüm bölümler üzerinde çalıştýrýsanız kendinize hiç şüphesiz C++ programcısı diyebilirsiniz. Kitabın bu son bölümünde C++’ın en can alıcı ve ileri yeniliklerinden bahsedeceğiz: Standart Şablon Kütüphanesi. Standart Şablon Kütüphanesi’nin (STL) C++’a katılması C++’ın standartlaştırılma zamanlarında meydana geldi. Standartlaştırma sürecinde STL C++’ın ilk tasarımlarında bulunmuyordu. STL, genel amaçlı bilinen algoritmaları ve veri yapılarını tanımlayan, şablonlaştırılmış sınıfları ve fonksiyonları hizmetinize sunar. Örneğin, vektörleri, sıraları, listeleri ve yiğinları destekler. Ayrıca onlara erişebilen çeşitli rutinler içerir. STL şablonlaştırılmış sınıflardan oluştugundan, algoritmalar ve veri yapıları her tipten veriye uygulanabilir.

STL’nin karmaşık yazılım tekniklerinden bazılarını ve C++’en ilginç yeniliklerini kullandığını ayrıca belirtmek gereklidir. STL’yi anlayıp kullanabilmek için, önceki konuları iyiye kavramış olmalısınız. Özellikle şablonlarla kendinizi kardeş gibi görmelisiniz. STL’yi tanımlayan şablon şekli, olduğundan daha karmaşık göründüğü için size biraz karışık gelebilir. Bu bölümde kitapta anlatılan konulardan daha zor bir şey olmasa da, ilk bakışta bu konular size anlaşılmaz ve caydırıcı görünecektir. Sabırlı olun ve örnekleri çalışın. Asla yabancı olduğunuz notasyonların sizi STL’nin basit yapısından uzaklaştırmasına izin vermeyin. STL büyük bir kütüphanedir ve bu bölümde tüm içeriği anlatılmayacaktır. Aslında STL’nin tam bir tanımı, yenilikleri, ayrıntıları ve programlama teknikleri koca bir kitabı doldurur. Buradaki giriş size temel işlemleri gösterecek ve tasarım düşüncesinden, programlamadan bahsedecektir. Bu bölümden sonra STL’nin geri kalanını kendi başınıza sorunsuz araştırabilirsiniz.

Bu bölüm aynı zamanda C++’ın en önemli sınıflarından biri olan **string** sınıfını ele alacaktır. **string** sınıfı operatörler kullanarak, diğer veri tipleriyle çalışabildiğiniz kadar size karakter katarlarıyla çalışma imkanı veren, katar veri yapısını tanımlar.

## Gözden Geçirme Testi

Devam etmeden önce alıştırmaları yapabiliyor ve soruları çözebiliyor olmalıdır.

1. **namespace**’in C++’a neden eklendiğini açıklayın.
2. **const** üye fonksiyonunu nasıl belirtirsiniz.
3. **mutable** düzenleyicileri size bir kütüphane fonksiyonunun programınızın kullanıcı tarafından değiştirilebilmesini sağlar. Doğru mu Yanlış mı?
4. Size şu sınıf veriliyor,

```
class X {
 int a, b;
public:
 X(int i, int j) { a = i, b = j; }
 //int'e dönüşümü burada yapın.
};
```

a ile b’nin toplamını döndüren bir tamsayı dönüşümü oluşturun.

5. Bir **static** üye değişkeni kendi bulunduğu sınıfın bir nesneden önce kullanılabilir.
6. Size aşağıdaki sınıf veriliyor,

```
class Demo {
 int a;
public:
 explicit Demo(int i) { a = i; }
 int geta() { return a; }
};
```

Aşağıdaki bildirim geçerli midir?

```
Demo o = 10;
```

## 14.1. Standart Şablon Dosyasına Giriş

Standart şablon dosyası büyük ve şekli karmaşık gözükse de, nasıl oluşturulduğunu ve hangi elemanları içerdigini öğrendiğinizde kullanması kolaylaşır. Bu sebeple kod örneklerinden herhangi birine bakmadan önce STL'ye bir giriş yapılmalıdır.

Standart şablon kütüphanesinin merkezinde üç temel birim vardır: Konteynırlar, algoritmalar ve iteratörler... Bu birimler bir çok yazılım problemlerini amada çözüm sağlamak için birbirleriyle ilişkili olarak çalışırlar.

**Konteynırlar** diğer nesneleri barınduran nesnelerdir. Çeşitli tiplerde konteynırlar vardır. Örneğin **vector** sınıfı dinamik bir diziyi tanımlar, **queue** bir sırayı oluşturur ve **list** lineer bir liste sağlar. Temel konteynirlara ek olarak STL, **birleşik konteynırlar** da tanımlar. Bunlar, anahtarları taban alan geri getirilebilen değişken imkanı verir. Örneğin **map** sınıfı belli anahtarlarla değerlere erişmeyi sağlayan haritayı tanımlar. Böylece, bir harita anahtar/değer çiftini saklar ve anahtar verildiğinde, değerin getirilmesine imkan verir.

Her konteynır sınıfı, konteynır uygulanan bir grup fonksiyon tanımlar. Örneğin, bir liste konteynır fonksiyon olarak **insert**, **delete** ve **merge** elamanlarını içerir. Yiğin, değerleri iten ve çeken (**push** ve **pop**) fonksiyonları barındırır.

**Algoritmalar** konteynırlar üzerinde rol oynar. Bazı servis algoritmaları konteynırların içini hazırlar, sıralar, arar ve dönüştürür. Birçok algoritma konteynırının içinde mevcut lineer eleman listeleri olan **sequence**'ların üzerinde işler.

**Iteratörler** işaretçi gibi nesnelerdir. Bir dizinin üzerinde gezinmek için nasıl işaretçi kullanılıyorsa, aynı şekilde iteratörler de, bir konteynırın içeriği üzerinde gezinme imkanı verir. Aşağıdaki tabloda 5 tip iteratör belirtilmiştir:

| iteratör                       | İzin verilen erişim                                    |
|--------------------------------|--------------------------------------------------------|
| Rasgele erişim (Random access) | Değerleri saklar ve okur. Elemanlara rasgele erişilir. |
| Çift yönlü(bi-directional)     | Değerleri saklar ve okur. İleri ve geri hareket.       |
| İleri (Forward)                | Değerleri saklar ve okur. Sadece ileri hareket         |
| Giriş (Input)                  | Değer okur ama saklamaz. Sadece ileri hareket.         |
| Çıkış (output)                 | Değer saklar ama okumaz. Sadece ileri hareket.         |

Genelde, bir iteratör daha az erişim hakkı olan diğer bir iteratörün yerine kullanılabilir. Örneğin, bir ileri (forward) iteratör bir giriş (input) iteratörünün yerine kullanılabilir.

Iteratörler aynı işaretçiler gibi ele alınır. Onları artırabilir veya azaltabilirsiniz. Ayrıca `*` operatörünü uygulayabilirsiniz. `iterator` tipi ile bildirilen iteratörler çeşitli konteynırlar tarafından tanımlanmıştır.

STL aynı zamanda *ters iteratörleri* (*reverse iterators*) destekler. Ters iteratörler, bir *sequence* (sıra) üzerinde ters yönde hareket eden, çift yön (*bidirectional*) veya rasgele erişim (*random-access*) iteratörleridir. Böylelikle eğer bir ters operatör bir sıranın sonuna işaret ediyorsa, iteratörü bir artırmak onun sondan bir önceki elemanı göstermesine sebep olur. Şablon tanımlamalarındaki çeşitli iteratör tiplerine başvururken, bu kitapta aşağıdaki tablo da listelenen terimler kullanılacaktır:

| Terim    | Iteratör tipi                                    |
|----------|--------------------------------------------------|
| Biliter  | Çift Yönlü (Bidirectional iterator)              |
| ForIter  | İleri iteratör (Forward iterator)                |
| InIter   | Giriş iteratör (Input iterator)                  |
| OutIter  | Çıkoş Iteratör (Output iterator)                 |
| RandIter | Rasgele erişim iteratör (Random-access iterator) |

Konteynirlara, algoritmalar ve iteratörlere ek olarak STL, destek vermek amacıyla birkaç farklı standart bileşeni daha kapsar. Bunlar kısaca, yer ayırcılar (*allocator*), predicate'ler (*lojik sonuç döndüren*) ve karşılaştırma fonksiyonlarıdır.

Her konteynının kendisi için tanımlanmış bir yer ayırcısı (*allocator*) vardır. Yer ayırcılar konteynirlar için bellek ayırma işini yürütür. Varsayılan yer ayırcı **allocator** sınıfından bir nesnedir fakat siz özel uygulamalar için ihtiyaç duyuyorsanız kenarlı yer ayırcılarınıza tanımlayabilirsiniz. Birçok durumda normal yer ayırcılar yeterli olacaktır.

Bazı algoritmalar ve konteynırlar *predicate* (*lojik sonuç döndüren, doğrulayıcı*) denen özel bir tip fonksiyon kullanır. Predicate'lerin iki tipi vardır: Tekli ve İkili ... Tekli *predicate* bir argüman alır ve ikili *predicate* iki argüman alır. Bu fonksiyonlar *true* (doğru) veya *false* (yanlış) değerini döndürürler. Bu bölümde tekli predicate kullanıldığında **UnPred** ile, ikili kullanıldığına ise **BinPred** ile belirtilecektir. Tekli *predicate* argümanlarının sırası her zaman *birinci*, *ikinci*'nin sırasında verilmelidir. Tekli ve ikili predicate için argümanlar konteynır tarafından saklanan nesnelerle aynı tipte değerler saklayacaklardır.

Bazı algoritmalar ve sınıflar iki elemanı karşılaştıran ve özel bir tip ikili *predicate* kullanır. Bunlara *karşılaştırma fonksiyonu* denir. Bu predicate'ler, ilk argüman diğerinden küçükse *true* değerini döndürür. Karşılaştırma fonksiyonları **Comp** tipinde belirtilecektir.

Ceşitli STL sınıflarının ihtiyaç duyukları başlıkların dışında, C++ standart kütüphanesi, STL'ye destek sağlamak için, `<utility>` ve `<functional>` başlıklarını içerir. `<utility>`, bir çift değeri depolayan şablon sınıf `pair`'in tanımını içerir. `pair`'i bu bölümün ileri-deki kısımlarında kullanacağız.

**<functional>** içerisindeki şablonlar, **operator()** tanımlayan nesneleri oluşturmanız için size yardım eder. Bunlara *fonksiyon nesneleri* (*function objects*) denir ve bir çok yerde fonksiyon işaretçilerinin yerine kullanılabilirler. **<functional>** içerisinde önceden tanımlanan fonksiyon nesneleri vardır. Bunlardan bazıları aşağıdaki tabloda verilmiştir.

|        |            |              |            |               |
|--------|------------|--------------|------------|---------------|
| plus   | minus      | multiplies   | divides    | modulus       |
| negate | equal_to   | not_equal_to | greater    | greater_equal |
| less   | less_equal | logical_and  | logical_or | logical_not   |

Belki en çok kullanılan fonksiyon nesnesi, bir nesnenin değerinin diğerinden az olduğunu tespit eden **less**'tir. Fonksiyon nesneleri, daha sonra anlatılacak olan STL algoritmaları içerisinde gerçek fonksiyon işaretçilerinin yerine kullanılabilirler. Fonksiyon işaretçilerinin yerine fonksiyon nesneleri kullanmak STL'nin daha etkin bir kod oluşturmasını sağlar. Ancak, bu bölümün amaçları açısından fonksiyon nesnelerine pek ihtiyacımız yok ve onları doğrudan kullanmayacağız. Fonksiyon nesnelerini kavramak zor olmasa da, onların detaylı bir açıklamasını vermek hem uzun sürer hem de bu kitabın içeriğinin dışında kahr. Bu konulara STL'den en iyi verimi almak istediğiniz zaman ihtiyaç duyabilirsiniz.

## Aliştırmalar

1. STL ile alakalı olarak konteynırlar, algoritmalar ve iteratörler nedir?
2. Predicate'lerin iki çeşidi nedir?
3. Iterator'lerin beş çeşidi nelerdir?

## 14.2. Konteynır (Container) Sınıfları

Önceki açıklamalarda ki, konteynırlar, gerçekten veri saklayan STL nesneleridir. STL tarafından tanımlanan konteynırları Tablo 14.1'de görebilirsiniz. Bu konteynırları kullanmak için dahil etmeniz gereken başlıklar da bu tabloda bulabilirsiniz. Karakter katarlarını kontrol eden **string** sınıfı da bir konteynirdir, fakat onu daha sonra ele alacağız.

| Konteynır      | Tanımı                                                                    | Gereken Başlık |
|----------------|---------------------------------------------------------------------------|----------------|
| bitset         | Bit kümesi                                                                | <bitset>       |
| deque          | Double sonlu kuyruk                                                       | <deque>        |
| list           | Lineer liste                                                              | <list>         |
| map            | Her tuşu sadece tek bir değerle bağlı olduğu tuş/değer çiftlerini saklar. | <map>          |
| multimap       | Bir tuşun birden fazla değerle bağlı olduğu tuş/değer çiftlerini saklar.  | <map>          |
| multiset       | Elemanların birbirinden değişik, tek olması gerekmeyen küme.              | <set>          |
| priority_queue | Öncelik kuyruğu                                                           | <queue>        |
| queue          | Kuyruk                                                                    | <queue>        |
| set            | Her elemanın tek olduğu küme                                              | <set>          |
| stack          | Yığın                                                                     | <stack>        |
| vector         | Dinamik dizi                                                              | <vector>       |

Tablo 14-1 STL tarafından tanımlanan konteynırlar

Şablon sınıf bildirimindeki taşıyıcıların isimleri keyfi olduğundan, konteynır sınıfları bu tiplerin **typedef** yapılmış sürümlerini deklare eder. Bu, tip isimlerini somut hale getirir. En yaygın kullanılan **typedef** isimlerinden bazlarını aşağıdaki tabloda verdik.

| <b>typedef İsmi</b>           | <b>Tanımı</b>                              |
|-------------------------------|--------------------------------------------|
| <b>size_type</b>              | <b>size_t</b> 'e eşdeğer bir integral tip  |
| <b>reference</b>              | Bir elemana referans                       |
| <b>const_reference</b>        | Bir elemana <b>const</b> referans          |
| <b>iterator</b>               | Bir iteratör                               |
| <b>const_iterator</b>         | Bir <b>const</b> iteratör                  |
| <b>reverse_iterator</b>       | Bir ters iteratör                          |
| <b>const_reverse_iterator</b> | Bir ters <b>const</b> iteratör             |
| <b>value_type</b>             | Konteynırda saklanan bir değerin tipi      |
| <b>allocator_type</b>         | Allocator (yer ayırcı) tipi                |
| <b>key_type</b>               | Anahtarın tipi                             |
| <b>key_compare</b>            | İki anahtar karşılaştırın fonksiyonun tipi |
| <b>value_compare</b>          | İki değer karşılaştırın fonksiyonun tipi   |

Bu bölümde konteynırların hepsini açıklamamız mümkün olmasa da bunlardan üç tanesini ilerdeki konularда inceleyeceğiz: **vector**, **list** ve **map**. Bu konteynırların nasıl çalıştığını bir kere anladığımızda, diğerlerini kullanırken de zorlanmayacaksınız.

### 14.3. Vektörler

Belki de konteynırların kullanılmasının en genel nedeni vektörlerdir. **vector** sınıfı dinamik dizileri destekler. Bu, gerektiğinde büyüyebilen bir dizidir. Bildiğiniz gibi C++ da dizinin büyüklüğü derleme amında sabittir. Bu her ne kadar dizilerle çalışırken en verimli yol olsada, bir o kadar da kısıtlayıcıdır. Çünkü dizinin büyülüğu değişen program koşullarına göre çalışma sırasında ayarlanamaz. Vektörler, bu problemi bellekte gerektiği kadar yer ayırtarak çözerler. Vektör dinamiktir, fakat elemanlarına ulaşmak için standart dizi notasyonunu kullanabilirsiniz.

**vector**'ün şablon belirtimi şu şekildedir:

```
template <class T, class Allocator = allocator<T>> class vector
```

Burada T saklanan verinin tipidir. **Allocator** yer ayırcayı belirler. Varsayılan değeri "standart allocator"dur. **vector** aşağıdaki constructor'lara (yapilandırıcı) sahiptir.

```
explicit vector(const Allocator &a = Allocator());
explicit vector(size_type num, const T &val = T(),
 const Allocator &a = Allocator());
vector(const vector<T, Allocator> &ob);
template <class InIter> vector(InIter start, InIter end,
 const Allocator &a = Allocator());
```

Bunlardan ilki boş bir vektör oluşturur. İkinci form *val* değerinde, *num* sayısındaki elemanlara sahip vektörü yapılandırır. *val* değeri varsayılan olabilir. Üçüncü form **ob** ile aynı elemanları içeren bir vektör oluşturur. Dördüncü form ise *start* ve *end* tarafından belirlenmiş alan-daki elemanları içeren bir vektör oluşturur.

Vektör içerisinde saklanacak her nesne varsayılan bir constructor tamamlamalıdır. Aynı zamanda `<` ve `==` işlemlerini tamamlamalıdır. Bazı derleyiciler diğer karşılaştırma operatörlerin de tamamlanmasını gerektirir.

(Tanımlamaların farklı olmasından dolayı kesin bilgi için derleyicinizin dokümanlarına başvurun.) Mevcut tiplerin tamamı bu gereksinimleri kendiliğinden karşılar.

Şablonun yazımı karışık görünse de vektörün deklare edilmesi zor değildir. Size şimdi birkaç örnek veriyorum:

```
vector<int> iv; // sıfır uzunluğunda bir int vektörü oluşturur
vector<char> cv(5); // 5 elemanlı bir char vektörü oluşturur
vector<char> cv(5, 'x'); // 5 elemanlı bir char vektörü için ilk değer verir
vector<int> iv2(iv); // bir int vektöründen başka bir int vektörü oluşturur.
```

Aşağıdaki karşılaştırma operatörleri `vector` için tanımlanmıştır:

```
, <, <=, !=, >, >=
```

Dizi başvuru operatörü `[ ]` de `vector` için tanımlanmıştır. Bu, standart dizi başvuru notasyonunu kullanarak vektörün elemanlarına erişmenizi sağlar.

`vector` tarafından tanımlanan üye fonksiyonlarını Tablo 14-2'de görebilirsiniz. (Yazım yine gözünüze korkutmasın) Üye fonksiyonların en önemlilerinden bazıları şunlardır: `size()`, `begin()`, `end()`, `push_back()`, `insert()` ve `erase()`. `size()` fonksiyonu vektörün o anki değerini döndürür. Bu fonksiyon oldukça kullanışlıdır, çünkü çalışma sırasında vektörün büyüklüğünü öğrenmenizi sağlar. Vektörlerin gerekli olduğunda büyülüklerinin artacağını unutmayın. Bu nedenle vektörün büyülüğüne çalışma sırasında karar verilir, derleme sırasında değil.

`begin()` fonksiyonu vektörün başlangıcını gösteren bir iteratör döndürür. `end()` fonksiyonu ise vektörün sonunu gösteren bir iteratör döndürür. Daha önce belirttiğimiz gibi iteratörler işaretçilere benzerler ve vektörün başlangıcını ve sonunu gösteren iteratörü elde etmeniz için `begin()` ve `end()` fonksiyonlarını kullanmalısınız.

#### Oye fonksiyon

```
template <class InIter>
void assign(InIter start, InIter end);
template <class Size, class T>
void assign(Size num,
 const T &val = T());
reference at(size_type i);
const_reference at(size_type i) const;
reference back();
const_reference back() const;
```

#### Açıklaması

`start` ve `end` tarafından tanımlanan vektörü sürece atar

Vektörü *val* değerindeki *num* elemanlarına atar

*i* tarafından belirtilen elemana referans döndürülür.

Vektördeki son elemana referans döndürür.

```

iterator begin();
const_iterator begin() const;
size_type capacity() const;

void clear();
bool empty() const;
iterator end();
const_iterator end() const;
iterator erase(iterator i);

iterator erase(iterator start, iterator end);

reference front();
const_reference front() const;
allocator_type get_allocator() const;
iterator insert(iterator i,
 const T &val = T());
void insert(iterator i, size_type num,
 const T &val)
template <class InIter>
void insert(iterator i, InIter start,
 InIter end);
size_type max_size() const;

reference operator[](size_type i) const;
const_reference operator[](size_type i)
 const;
void pop_back();
void push_back(const T &val);

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
void reserve(size_type num);

void resize(size_type num, T val = T());

size_type size() const;
void swap(vector<T, Allocator> &ob)

```

Vektördeki ilk elemana iteratör döndürür.

Vektörün o anki kapasitesini döndürür. Bu, bellekte daha fazla yer ayırmadan kaldırabileceğiniz eleman sayısıdır.

Vektörden tüm elemanları siler.

Uyancı vektör boşsa true aksi takdirde false döndürür.

Vektörün sonunu döndüren bir iteratör döndürür.

$i$  tarafından işaret edilen elemani siler. Silinen elemandan sonraki elemani gösteren bir iteratör döndürür.

$i$  ile  $end$  arasındaki elemanları siler. Son silinen elemandan sonraki elemani gösteren bir iteratör döndürür.

Vektörün ilk elemanın göstergesi bir referans döndürür.

Vektörün allocator'unu (yer ayıcı) döndürür.

$i$  tarafından belirlenen elemanın hemen öncesine val ekler. Elemani gösteren bir işaretçi döndürür.

$i$  tarafından belirtilen elemanın hemen öncesine num sayıda val ekler.

$i$  tarafından belirlenen elemanın hemen öncesine start ve end tarafından bir süreci(sequence) ekler.

Vektörün tutabileceği maksimum eleman sayısını döndürür.

$i$  tarafından belirlenen elemani gösteren bir referans döndürür.

Vektörün son elemanını atar.

Vektörün sonuna val tarafından belirlenen değere sahip bir eleman ekler.

Vektörün sonunu gösteren ters bir iteratör döndürür.

Vektörün başını gösteren ters bir iteratör döndürür.

Vektörün kapasitesini en az num'a eşit olacak şekilde ayarlar.

Vektörün büyüklüğünü num tarafından belirlenen değere göre değiştirir. Vektörün uzatılması gerekirse num tarafından belirtilen değere sahip elemanlar sona eklenir.

Vektörde o anda bulunan eleman sayısını döndürür.

Uyancı vektörde saklanan elemanları ob'un içindelerle değiştirir.

Tablo 14-2 Vektör üye fonksiyonları

**push\_back()** fonksiyonu vektörün sonuna bir değer koyar. Eğer gereklirse vektörün büyülüğu yeni elemana uyacak şekilde artırır. **insert'**i kullanarak araya da eleman ekleyebilirsiniz. Vektöre ilk değer de verilebilir. Her durumda vektörün elemanları varsa, bu elemanlara erişmek ve onları değiştirmek için dizi operatörünü kullanabilirsiniz. **erase( )**'i kullanarak vektörden elemanlar silebilirsiniz.

## Örnekler

- Vektörlerin temel işlemlerini gösteren kısa bir örnek veriyorum.

```
// Vektor temelleri
#include <iostream>
#include <vector>
using namespace std;

int main()
{
 vector<int> v; // sıfır uzunluğunda bir vektor oluştur.
 int i;

 // v'nin ilk büyüklüğünü göster.
 cout << "Size = " << v.size() << endl;

 /* vektörün sonuna değerler koy - vektor gerektiği kadar büyüyecek */
 for(i=0; i<10; i++) v.push_back(i);
 // v'nin şimdiki büyüklüğünü görüntüle
 cout << "Size now = " << v.size() << endl;

 // vektörün içeriğini göster
 cout << "Current contents:\n";
 for(i=0; i<v.size(); i++) cout << v[i] << " ";
 cout << endl;

 /* sonuna daha fazla değer koy - vektor yine gerektiği kadar büyüyecek */
 for(i=0; i<10; i++) v.push_back(i+10);

 // v'nin şimdiki büyüklüğünü göster
 cout << "Size now = " << v.size() << endl;

 // vektörün içeriğini göster
 cout << "Current contents:\n";
 for(i=0; i<v.size(); i++) cout << v[i] << " ";
 cout << endl;

 // vektörün içeriğini değiştir
 for(i=0; i<v.size(); i++) v[i] = v[i] + v[i];
 // vektörün içeriğini göster
 cout << "Contents doubled:\n";
 for(i=0; i<v.size(); i++) cout << v[i] << " ";
 cout << endl;

 return 0;
}
```

Programın çıktısı şöyledir:

```
Size = 0
Size now = 10
Current contents:
0 1 2 3 4 5 6 7 8 9
Size now = 20
Current contents:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Contents doubled:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

Programı dikkatlice inceleyelim. **main( )**'in içerisinde **v** isminde bir tamsayı vektörü oluşturulur. İlk değeri olmadığından boş bir vektördür. Yani 0 uzunluğundadır. Program **size( )** üye fonksiyonunu çağırarak bunu onaylar. Daha sonra **push\_back( )** fonksiyonu ile **v**'nin sonuna 10 eleman eklenir. Bu **v**'nin yeni elemanları karşılayabilecek kadar büyümeye yol açar. Çıkışın gösterdiği gibi, bu eklerden sonra büyülüğu 10 oluyor. Daha sonra **v**'nin içeriği görüntülenir. Standart dizi erişim notasyonunun kullanıldığından dikkat edin. Daha sonra 10 eleman daha ekleniyor ve **v**'nin büyülüğu kendiliğinden elemanları taşıyabilecek seviyeye geliyor. Son olarak **v** elemanlarının değerleri dizilerle aynı şekilde değiştirilir.

Bu programda ilginç olan bir nokta daha var. **v**'nin içeriğini gösteren döngüler hedef olarak **v.size( )**'ı kullanmaktadır. Vektörlerin dizilere olan avantajlarından biri de vektörün o anki büyülüğünün bulunmasının mümkün olmasıdır. Siz de tahmin edersiniz ki, bu pek çok durumda oldukça faydalıdır.

2. Bildiğiniz gibi diziler ve işaretçiler C++'a sıkça bağlıdır. Dizilere, dizi başvuru operatöryle ya da bir işaretçi üzerinden erişebilirsiniz. STL'de de buna paralel olarak vektörler ve iteratörler arasında bir bağ vardır. Vektörün üyelerine yine başvuru operatörleriyle veya iteratörlerle erişebilirsiniz. Aşağıdaki örnekte her iki yöntemi de görebilirsiniz.

```
// iterator kullanarak vektöre erişim
#include <iostream>
#include <vector>
using namespace std;

int main()
{
 vector<int> v; // sıfır uzunluğunda vektör oluştur.
 int i;

 // vektöre değerler koy
 for(i=0; i<10; i++) v.push_back(i);

 // Vektörün içeriğine başvuru operatörlü ile erişim
 for(i=0; i<10; i++) cout << v[i] << " ";
 cout << endl;

 // iterator ile erişim
 vector<int>::iterator p = v.begin();
 while(p != v.end()) {
 cout << *p << " ";
 p++;
 }

 return 0;
}
```

Programın çıktısı aşağıdaki gibidir.

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Bu programda vektör sıfır uzunluğunda oluşturulur. `push_back()` fonksiyonu vektörün sonuna değerler koyar ve vektörün büyütüğünü gereği kadar artırır. Vektörün sonuna değerler koyar ve vektörün büyütüğünü gereği kadar artırır.

`p` iteratörünün nasıl deklare edildiğine dikkat edin. Iteratör tipi konteynır sınıflar tarafından tanımlanır. Böylece, belirli bir konteynır için iteratör elde etmek amacıyla örnekte gösterilen benzer bir bildirim kullanacaksınız. Programda `begin()` üye fonksiyonu ile `p` vektörün başına getirilir. Bu fonksiyon vektörün başına bir iteratör döndürür. Bu iteratör gerçeklikçe bir artırmak suretiyle vektörün elemanlarına erişmek için kullanılabilir. Bu, bir dizinin elemanlarına işaretçi ile erişime paralel bir yoldur. Vektörün sonuna gelinip gelinmediğini anlamak için `end()` kullanılır. Bu fonksiyon, son elemandan bir sonraki elemani gösteren bir iteratör döndürür. Böylece, `p, v.end()`'e eşit olduğunda vektörün sonuna erişilmiş olur.

- Vektörün sonuna elemanın yanı sıra, `insert()` fonksiyonu ile vektörün arasına da eleman eklenebilir. `erase()`'i kullanarak işe elemanları silebilirsınız. Aşağıdaki programda `insert()` ve `erase()` fonksiyonlarını inceleyelim.

```
// insert ve erase'ı gösterir.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
 vector<int> v(5, 1); // create 5-element vector of 1s
 int i;

 // vektörün ilk içeriğini gösterir.
 cout << "Size = " << v.size() << endl;
 cout << "Original contents:\n";
 for(i=0; i<v.size(); i++) cout << v[i] << " ";
 cout << endl << endl;

 vector<int>::iterator p = v.begin();
 p += 2; // 3. Elemani işaret eder

 // 9 değerinde 10 eleman aray sokar.
 v.insert(p, 10, 9);

 // araya koyma işleminden sonraki içeriği gösterir.
 cout << "Size after insert = " << v.size() << endl;
 cout << "Contents after insert:\n";
 for(i=0; i<v.size(); i++) cout << v[i] << " ";
 cout << endl << endl;

 // bu elemanları sil
 p = v.begin();
 p += 2; // 3. Elemani göster
 v.erase(p, p+10); // sonraki 10 elemanı siler

 // silme işleminden sonraki içeriği gösterir.
 cout << "Size after erase = " << v.size() << endl;
 cout << "Contents after erase:\n";
 for(i=0; i<v.size(); i++) cout << v[i] << " ";
}
```

```

 cout << endl;
 return 0;
}

```

Program şu çıktıyi üretir.

```

size = 5
Original contents:
1 1 1 1 1
Size after insert = 15
Contents after insert:
1 1 9 9 9 9 9 9 9 9 9 1 1 1
Size after erase = 5
Contents after erase:
1 1 1 1 1

```

4. Programcı tarafından tanımlanmış sınıfın nesnelerini saklamak için vektör kullanan bir program inceleyelim. Sınıfın varsayılan constructor'ü tanımladığına ve aşırı yüklenmiş < ve == operatörleri sağladığına dikkat edin. Derleyicinin STL'yi nasıl tanımladığına dayanarak diğer karşılaştırma operatörlerinin de tanımlamasının gerçekleştirileceğini hatırlayın.

```

// Sınıf nesnesini vektörde saklar.
#include <iostream>
#include <vector>
using namespace std;

class Demo {
 double d;
public:
 Demo() { d = 0.0; }
 Demo(double x) { d = x; }
 Demo &operator=(double x)
 {
 d = x; return *this;
 }
 double getd() { return d; }
};

bool operator<(Demo a, Demo b)
{
 return a.getd() < b.getd();
}

bool operator==(Demo a, Demo b)
{
 return a.getd() == b.getd();
}

int main()
{
 vector<Demo> v;
 int i;

 for(i=0; i<10; i++)
 v.push_back(Demo(1/3.0));
 for(i=0; i<v.size(); i++)
 cout << v[i].getd() << " ";
}

```

```

 cout << endl;

 for(i=0; i<v.size(); i++)
 v[i] = v[i].getd() * 2.1;

 for(i=0; i<v.size(); i++)
 cout << v[i].getd() << " ";

 return 0;
}

```

Programın çıkışı aşağıdaki gibidir.

```

0 0.333333 0.666667 1 1.33333 1.66667 2 2.33333 2.66667 3
0 0.7 1.4 2.1 2.8 3.5 4.2 4.9 5.6 6.3

```

## Aliştırmalar

- Verilen örnekleri biraz değiştirerek üzerinde çalışın. Sonuçları gözleyin.
- Örnek 4'te **Demo** için hem varsayılan hem de parametreleştirilmiş yapılandırıcı tanımlanmıştır. Bunu neden önemli olduğunu açıklayın.
- Size basit, **Coord** adında bir sınıf veriyorum. **Coord** sınıfından nesneleri bir vektor içerisinde saklayan bir program hazırlayın. (İpucu: **Coord**'a bağlı olarak < ve == operatörleri tanımlamayı unutmayın)

```

class Coord {
public:
 int x, y;
 Coord() { x = y = 0; }
 Coord(int a, int b) { x = a; y = b; }
};

```

## 14.4. Listeler

**list** sınıfı iki yönlü lineer bir liste sağlar. Rasgele erişilebilen vektörlerin aksine, listelere sadece sırayla erişilebilir. Listeler iki yönlü oldukları için önden arkaya veya arkadan öne erişilebilirler. **list** sınıfının şablon bildirimi şu şekildedir:

```
template <class T, class Allocator = allocator<T>> class list
```

Burada **T** listede depolanan verinin tipidir. Yer ayırcı **allocator** ile belirlenir. Varsayılanı *standart allocator*'dur (yer ayırcı). Bu sınıf aşağıdaki yapılandırıcılara sahiptir.

```

explicit list(const Allocator &a = Allocator());
explicit list(size_type num, const T &val = T(),
 const Allocator &a = Allocator());
list(const list<T, Allocator> &ob);
template <class InIter> list(InIter start, InIter end,
 const Allocator &a = Allocator());

```

Bunlardan ilki boş bir liste oluşturur. İkincisi varsayılan olmasına izin verilen, *val* değerini içeren *num* sayıda elemanı olan bir liste oluşturur. Üçüncü form *ob*'daki elemanlarla aynı elemanları içeren bir liste oluşturur. Dördüncüsü ise *start* ile *end* iteratörleri tarafından belirlenen aralıkta eleman içeren bir liste oluşturur.

Aşağıdaki karşılaştırma operatörleri **list** için tanımlanmıştır:

`==, <, <=, !=, >, >=`

**list** için tanımlanmış üye fonksiyonlar Tablo 14.3'de gösterilmiştir. Vektörler gibi listelerde **push\_back()** fonksiyonu ile eleman eklenebilir. **push\_front()** ile listenin önünden, **insert()** ile de ortasına eleman ekleyebilirsiniz. **splice()** ile iki listeyi birleştirebilir, **merge()** ile birbirine ekleyebilirsiniz.

Liste içerisinde tutulacak herhangi bir veri tipi varsayılan yapılandırıcıyı tanımlamalıdır. Aynı zamanda çeşitli karşılaştırma operatörlerini de tanımlama zorunluluğu vardır. Ben bu kitabı hazırlayıorken, listede saklanacak nesnelerin ön gereksinimleri derleyiciiden derleyiciye değişebilir. Bu nedenle derleyicinizin dokümanlarına bakmanız gerekiyor.

#### Üye fonksiyon

```
template <class InIter>
void assign(InIter start, InIter end);
template <class Size, class T>
void assign(Size num,
const T &val = T());
reference back();
const_reference back() const;
iterator begin();
const_iterator begin() const;
void clear();
bool empty() const;
iterator end();
const_iterator end() const;
iterator erase(iterator i);

iterator erase(iterator start, iterator end);

reference front();
const_reference front() const;
allocator_type get_allocator() const;
iterator insert(iterator i,
const T &val = T());
void insert(iterator i, size_type num,
const T &val)
template <class InIter>
void insert(iterator i
InIter start, InIter end);
size_type max_size() const;
```

#### Açıklaması

start ve end tarafından belirlenen liste sürecini atar.

val değerinde num elemanlı bir liste atar.

Listedeki son elemana bir referans döndürür.

Listedeki ilk elemanı gösteren bir iteratör döndürür.

Listedeki tüm elemanları siler.

Uyarlan liste boşsa true aksi takdirde false döndürür.

Listenin sonunu gösteren bir iteratör döndürür.

i tarafından işaret edilen elemanı siler.

Çıkarılan elemandan sonrakine bir iteratör döndürür.

start ile end arasındaki elemanları siler. Çıkarılan son elemandan sonrakine bir iteratör döndürür.

Listedeki ilk elemanı gösteren bir referans döndürür.

Listenin yer ayıncısını döndürür.

i tarafından belirlenen elemandan hemen bir öncekine val değerini sokar. Elemanı gösteren bir iteratör döndürür.

i tarafından belirlenen elemandan hemen önce val değerinde num kadar kopya sokar.

i tarafından gösterilen elemandan hemen önce start ve end tarafından belirlenen süreci sokar.

Listenin taşıyabileceğini maksimum eleman sayısını

|                                          |                                                                                                                                                                                                                                                                                           |
|------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void merge(list<T, Allocator> &ob);      | döndürür.                                                                                                                                                                                                                                                                                 |
| template <class Comp>                    | Uyarı: sıralı liste ile ob içerisinde bulunan sıralanmış listeyi biribirine ekler. Sonuçta sıralanmıştır. Eklemeden sonra ob içerisindeki liste silinmiştir. İkinci formda, bir elemanın değerinin diğerinden z olup olmadığını anlamak için bir karşılaştırma fonksiyonu belirtilebilir. |
| void merge(<list<T, Allocator> &ob,      |                                                                                                                                                                                                                                                                                           |
| Comp cmpfn);                             | Listedeki son elemanı atar.                                                                                                                                                                                                                                                               |
| void pop_back();                         | Listedeki ilk elemanı atar.                                                                                                                                                                                                                                                               |
| void pop_front();                        | Val ile belirlenmiş bir elemanı liste sonuna ekler.                                                                                                                                                                                                                                       |
| void push_back(const T &val);            | Val ile belirlenmiş bir elemanı liste başına ekler.                                                                                                                                                                                                                                       |
| void push_front(const T &val);           | Listenin sonuna ters iteratör döndürür.                                                                                                                                                                                                                                                   |
| reverse_iterator rbegin();               | Val değerinde elemanları listeden çıkarır.                                                                                                                                                                                                                                                |
| const_reverse_iterator rbegin() const;   | Bir mantıksal ifade pr'si true olanları listeden atar.                                                                                                                                                                                                                                    |
| void remove(const T &val);               | Listenin başına ters iteratör döndürür.                                                                                                                                                                                                                                                   |
| template <class UnPred>                  | Num tarafından belirlenen derecede listenin uzunluğunu değiştirir. Eğer liste genişleyecekse val değerinde elemanlar listenin sonuna eklenir.                                                                                                                                             |
| void remove_if(UnPred pr);               | Uyarı: listeyi tersine çevirir.                                                                                                                                                                                                                                                           |
| reverse_iterator rend();                 | Listedeki toplam eleman sayısını döndürür.                                                                                                                                                                                                                                                |
| const_reverse_iterator rend() const;     | Listeyi sıralar. İkinci tıpte, bir değerin diğerinden küçük olup olmadığını saptamak için cmpfn karşılaştırma fonksiyonu kullanılarak sıralama yapılır.                                                                                                                                   |
| void resize(size_type num, T val = T()); | Ob'nin içini i tarafından gösterilen noktada uyarın listeye sokar. İşlemden sonra ob boşaltılır.                                                                                                                                                                                          |
| void reverse();                          | Ob listesi içerisinde ei tarafından gösterilen elemanı atar ve uyarın listeye i tarafından gösterilen noktada sokar.                                                                                                                                                                      |
| size_type size() const;                  | Start ve end tarafından belirlenen aralığı ob içerisindeki elemanları değiştirecektir.                                                                                                                                                                                                    |
| void sort();                             | Uyarı: listedeki elemanları ob içerisindeki elemanlarla değiştirir.                                                                                                                                                                                                                       |
| template <class Comp>                    | Uyarı: listedeki aynı elemanları çıkarır. İkinci tipi ise pr'yi kullanarak tekliği saptar.                                                                                                                                                                                                |
| void sort(Comp cmpfn);                   |                                                                                                                                                                                                                                                                                           |
| void splice(iterator i,                  |                                                                                                                                                                                                                                                                                           |
| list<T, Allocator> &ob);                 |                                                                                                                                                                                                                                                                                           |
| void splice(iterator i,                  |                                                                                                                                                                                                                                                                                           |
| list<T, Allocator> &ob,                  |                                                                                                                                                                                                                                                                                           |
| iterator el);                            |                                                                                                                                                                                                                                                                                           |
| void splice(iterator i,                  |                                                                                                                                                                                                                                                                                           |
| list<T, Allocator> &ob,                  |                                                                                                                                                                                                                                                                                           |
| iterator start, iterator end);           |                                                                                                                                                                                                                                                                                           |
| void swap(list<T, Allocator> &ob)        |                                                                                                                                                                                                                                                                                           |
| void unique();                           |                                                                                                                                                                                                                                                                                           |
| template <class BinPred>                 |                                                                                                                                                                                                                                                                                           |
| void unique(BinPred pr);                 |                                                                                                                                                                                                                                                                                           |

Tablo 14.3. list Üye fonksiyonları

## Örnekler

1. Listelere basit bir örnek verelim:

```
// liste temelleri
#include <iostream>
#include <list>
using namespace std;

int main()
```

```

{
 list<char> lst; // create an empty list
 int i;

 for(i=0; i<10; i++) lst.push_back('A'+i);

 cout << "Size = " << lst.size() << endl;

 list<char>::iterator p;

 cout << "Contents: ";
 while(!lst.empty()) {
 p = lst.begin();
 cout << *p;
 lst.pop_front();
 }

 return 0;
}

```

Program çıktısı aşağıdaki gibidir:

```

Size = 10
Contents: ABCDEFGHIJ

```

Bu program bir karakter listesi oluşturur. İlk önce boş bir `list` nesnesi oluşturulur. Sonra listeye A'dan J'ye kadar on karakter koyar. Bu işlem `push_back()` fonksiyonu ile yapılır. Bu fonksiyon mevcut listenin sonuna yeni değerler koyar. Daha sonra listenin büyüklüğü görüntülenir. Daha sonra işlem, listenin içeriği tekrar tekrar elde edilerek, görüntülenerek ve ilk elemanı listeden atarak, liste boşalana kadar devam eder.

2. Bir önceki örnekte tersine çevrildiği zaman boşaltılmıştı. Elbette bu gerekli değildir. Örneğin, listeyi gösteren döngü kodu şu şekilde değiştirilebilir:

```

list<char>::iterator p = lst.begin();
while(p != lst.end()) {
 cout << *p;
 p++;
}

```

Burada `p` iteratörünün ilk değer listenin başını gösteresi için verilmiştir. Döngüdeki her adımda `p` artırılır ve bu şekilde bir sonraki elemanı göstermesi sağlanır. `p` listenin sonuna geldiğinde döngü sona erer.

3. Listeler iki yönlü olduğundan elemanlar listenin başına veya sonuna eklenebilir. Örneğin aşağıdaki program iki liste oluşturmaktadır. Bunlardan biri diğerinin tersidir.

```

// elemanlar listenin başına ve sonuna konabilir.
#include <iostream>
#include <list>
using namespace std;

```

```

int main()
{
 list<char> lst;
 list<char> revlst;
 int i;

 for(i=0; i<10; i++) lst.push_back('A'+i);

 cout << "Size of lst = " << lst.size() << endl;
 cout << "Original contents: ";

 list<char>::iterator p;

 /* Elemanları 1'ten itibaren itibaren sil ve ters sırada revlst'e koy. */
 while(!lst.empty()) {
 p = lst.begin();
 cout << *p;
 lst.pop_front();
 revlst.push_front(*p);
 }
 cout << endl << endl;

 cout << "Size of revlst = ";
 cout << revlst.size() << endl;
 cout << "Reversed contents: ";
 p = revlst.begin();
 while(p != revlst.end()) {
 cout << *p;
 p++;
 }

 return 0;
}

```

Program bize şu çıktıyi verir:

```

Size of lst = 10
Original contents: ABCDEFGHIJ

Size of revlst = 10
Reversed contents: JIHGFEDCBA

```

Programda, liste, elemanları lst'in başından silerek ve onları revlst'nin önünden itibaren iterek ters çevirir. Bu sayede elemanlar revlst'de ters sırada saklanırlar.

4. Bir listeyi `sort()` üye fonksiyonunu çağırarak sıralayabilirsiniz. Aşağıdaki program, rasgele karakterler içeren bir liste oluşturur, daha sonra onları sıralayarak diğer bir listeye yerleştirir.

```

// Liste sıralar
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
 list<char> lst;

```

```

int i;

// rastgele karakterler içeren liste oluştur
for(i=0; i<10; i++)
 lst.push_back('A'+(rand()%26));

cout << "Original contents: ";
list<char>::iterator p = lst.begin();
while(p != lst.end()) {
 cout << *p;
 p++;
}
cout << endl << endl;

// listeyi sırala
lst.sort();

cout << "Sorted contents: ";
p = lst.begin();
while(p != lst.end()) {
 cout << *p;
 p++;
}

return 0;
}

```

Program şu çıktıının benzerini üretir.

```

Original contents: PRQGHOMEAY
Sorted contents: AEGHJMOPQUY

```

- Sıralı bir liste bir dğeriyle eklenebilir. Sonuçta her iki listenin içeriğine sahip sıralı bir liste oluşur. Yeni liste uyaran listede depolanır ve ikinci listede boşaltılır. Bu örnekte iki liste birleştiriliyor. Listelerin ilki ACEGI harflerini ikincisi BDFHJ harflerini içermeye. Bu listeler daha sonra ABCDEFGHIJ. Sürecini oluşturmak için birleştiriliyor.

```

// İki listeyi birleştirir.
#include <iostream>
#include <list>
using namespace std;

int main()
{
 list<char> lst1, lst2;
 int i;

 for(i=0; i<10; i+=2) lst1.push_back('A'+i);
 for(i=1; i<11; i+=2) lst2.push_back('A'+i);

 cout << "Contents of lst1: ";
 list<char>::iterator p = lst1.begin();
 while(p != lst1.end()) {
 cout << *p;
 p++;
 }
 cout << endl << endl;

```

```

cout << "Contents of lst2: ";
p = lst2.begin();
while(p != lst2.end()) {
 cout << *p;
 p++;
}
cout << endl << endl;

// Şimdi birleştirir
lst1.merge(lst2);
if(lst2.empty())
 cout << "lst2 is now empty\n";

cout << "Contents of lst1 after merge:\n";
p = lst1.begin();
while(p != lst1.end()) {
 cout << *p;
 p++;
}

return 0;
}

```

Programın çıktısı şu şekildedir.

```

Contents of lst1: ACEGI
Contents of lst2: BOFHJ
lst2 is now empty
Contents of lst1 after merge:
ABCDEFGHIJ

```

- Bu örnek, yazılım projelerini denetlemeyi sağlayan **Project** tipinde nesneleri saklamak için bir liste kullanmaktadır. **<**, **>**, **!=** ve **==** operatörlerin **Project** tipinden nesneler için aşırı yüklediğine dikkat edin. Bu operatörler Microsoft Visual C++ 5 tarafından gerek duyulan operatörlerdir (bu bölümdeki STL örneklerini derlemek için kullanılan derleyici). Diğer derleyiciler, ek operatörleri aşırı yüklemenize ihtiyaç duyabilirler. STL, bu fonksiyonları konteynır içerisindeki nesnelerin sırasını ve eşitliğini saptamak için kullanır. Liste sıralı bir konteynır olmasa bile, ararken, sıralarken ve birleştirirken elemanları karşılaştırmak için bir yola ihtiyaç duyar.

```

#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class Project {
public:
 char name[40];
 int days_to_completion;
 Project() {
 strcpy(name, "");
 days_to_completion = 0;
 }
}

```

```
Project(char *n, int d) {
 strcpy(name, n);
 days_to_completion = d;
}

void add_days(int i) {
 days_to_completion += i;
}

void sub_days(int i) {
 days_to_completion -= i;
}

bool completed() { return !days_to_completion; }

void report() {
 cout << name << ":" ;
 cout << days_to_completion;
 cout << " days left.\n";
}

bool operator<(const Project &a, const Project &b)
{
 return a.days_to_completion < b.days_to_completion;
}

bool operator>(const Project &a, const Project &b)
{
 return a.days_to_completion > b.days_to_completion;
}

bool operator==(const Project &a, const Project &b)
{
 return a.days_to_completion == b.days_to_completion;
}

bool operator!=(const Project &a, const Project &b)
{
 return a.days_to_completion != b.days_to_completion;
}

int main()
{
 list<Project> proj;

 proj.push_back(Project("Compiler", 35));
 proj.push_back(Project("Spreadsheet", 190));
 proj.push_back(Project("STL implementation", 1000));

 list<Project>::iterator p = proj.begin();

 /* display projects */
 while(p != proj.end()) {
 p->report();
 p++;
 }

 // 1. Projeye 10 gün ekle
 p = proj.begin();
 p->add_days(10);
}
```

```

// tamamlamak için ilk projeyi gönder
do {
 p->sub_days(5);
 p->report();
} while (!p->completed());

return 0;
}

```

Programın çıktısı aşağıdaki gibidir.

```

Compiler: 35 days left.
Spreadsheet: 190 days left.
STL Implementation: 1000 days left.
Compiler: 40 days left.
Compiler: 35 days left.
Compiler: 30 days left.
Compiler: 25 days left.
Compiler: 20 days left.
Compiler: 15 days left.
Compiler: 10 days left.
Compiler: 5 days left.
Compiler: 0 days left.

```

## Aliştırmalar

1. Küçük değişiklikler yaparak alıştırmalar üzerinde çalışın.
2. Örnek 1'de liste gösterme işlemi sırasında boşaltılmaktadır. Örnek 2'de listeyi yok etmeden tersine çevirmenin başka bir yolunu gördünüz. Bir listeyi silmeden tersine çevirmenin başka bir yolu var mı? Çözümünün çalıştığını, örnek 1'deki programa katarak gösterin.
3. Aliştırmalar 6'ya verdığınız cevabı aşağıdakilerden oluşan, başka bir projeler listesi oluşturarak geliştirin. Daha sonra her iki listeyi de sıralayın ve onları birleştirin. Sonucu gösterin.

| Project     | Days to Completion |
|-------------|--------------------|
| Database    | 780                |
| Mail merge  | 50                 |
| COM objects | 300                |

## 14.5. Haritalar

**map** sınıfı farklı anahtarların değerlerle haritalandığı birleşik bir konteynırı destekler. Eşsiz bir anahtar bir değere verdığınız isimdir. Değer saklandığında onu, anahtarını kullanarak elde edebilirsiniz. Böylece en genel anlamda harita anahtar/değer çiftlerinin listesidir. Haritanın gücü anahtarları verilmiş bir değeri arayabilmenizdir. Örneğin, bir kişinin ismini anahtar olarak kullanan ve kişinin telefon numarasını da değer olarak saklayan bir harita kullanabileirdiniz. Birleşik konteynlar programlamada gittikçe daha popüler hale gelmektedir.

Daha önce belirttiğimiz gibi harita sadece farklı değerler saklayabilir. Aynı anahtar birden fazla kullanılamaz. Farklı olmayan anahtarlarla izin veren bir harita oluşturmak için **multimap** kullanın. **map** konteynırının şablon gösterimi şu şekildedir:

```
template <class Key, class T, class Comp = less<Key>,
 class Allocator = allocator<T>> class map;
```

Burada **Key** anahtarların veri tipidir. **T** saklanmakta olan (haritalanan) değerlerin veri tipidir. **Comp** ise iki anahtarı karşılaştırın bir fonksiyondur. Varsayılanı standart **less()** hizmet fonksiyon nesnesidir. **Allocator** (yer ayırcı) yer ayırcıdır. (varsayılanı **allocator**'dır). **map** sınıfının aşağıdaki gösterilen yapılandırıcıları (constructor) vardır.

```
explicit map(const Comp &compfn = Comp(),
 const Allocator &a = Allocator());
map(const map<Key, T, Comp, Allocator> &ob);
template <class Iterator> map(Iterator start, Iterator end,
 const Comp &compfn = Comp(), const Allocator &a = Allocator());
```

Bunlardan ilki boş bir harita oluşturur. İkincisi ise **ob** ile aynı elemanları içeren bir harita oluşturur. Üçüncü form, **start** ve **end** iteratörleri tarafından belirlenen alandaki elemanları içeren bir harita oluşturur. **compfn** tarafından belirlenen fonksiyon mevcutsa, bu fonksiyon haritanın sıralmasını düzenler.

Genelde anahtar olarak kullanılan her nesne varsayılan bir constructor (yapıldırıcı) tanımlamalı ve gerekli tüm karşılaştırma operatörlerini aşırı yüklemelidir.

Aşağıdaki karşılaştırma operatörler **map** için tanımlanmıştır.

```
==, <, <=, !=, >, >=
```

**map** tanımlanan üye fonksiyonları Tablo 14.4'te bulabilirsiniz. Açıklamalarda **key\_type** anahtarın tipidir, **value\_type** ise **pair<Key, T>**'yi temsil eder.

#### Üye Fonksiyon

```
iterator begin();
const_iterator begin() const;
void clear();
size_type count(const key_type &k) const;
bool empty() const;
iterator end();
const_iterator end() const;
pair<iterator, iterator>
equal_range(const key_type &k);
pair<const_iterator, const_iterator>
equal_range(const key_type &k) const;
void erase(iterator i);
void erase(iterator start, iterator end);
```

#### Açıklaması

Haritadaki ilk elemani gösteren bir iteratör döndürür.

Haritadaki tüm elemanları siler.

Haritada k'nın kaç kere meydana geldiği döndürür.( 1 veya 0 )

Uyarı harita bossa true aksi takdirde false döndürür.  
Haritanın sonunu gösteren bir iteratör döndürür.

Belirli anahtarı içeren haritadaki ilk ve son elemanları gösteren iteratör çiftini döndürür.

i tarafından işaret edilen elemani siler.  
start ile end arasındaki elemanları siler.

## BÖLÜM 14: Standart Şablon Kütüphanesine Giriş

|                                          |                                                                                                                                                                                                                                        |
|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| size_type erase(const key_type &k)       | K değerine sahip anahtarın olan elemanları siler.                                                                                                                                                                                      |
| iterator find(const key_type &k);        | Belirli anahtarın gösteren bir işaretçi döndürür. Eğer anahtar bulunmuyorsa haritanın sonunu gösteren bir iteratör döndürülür.                                                                                                         |
| const_iterator find(const key_type &k)   |                                                                                                                                                                                                                                        |
| const;                                   |                                                                                                                                                                                                                                        |
| allocator_type get_allocator( ) const;   |                                                                                                                                                                                                                                        |
| iterator insert(iterator i,              | Haritanın yer ayırcısını döndürür.                                                                                                                                                                                                     |
| const value_type &val);                  | i tarafından belirlenen elemana veya ondan bir sonrakine val değerini sokar.                                                                                                                                                           |
| template <class InIter>                  |                                                                                                                                                                                                                                        |
| void insert(InIter start, InIter end)    | Bir aralıktaki elemanları araya sokar                                                                                                                                                                                                  |
| pair<iterator, bool>                     |                                                                                                                                                                                                                                        |
| insert(const value_type &val);           |                                                                                                                                                                                                                                        |
| key_compare key_comp( ) const;           | Uyarı haritaya val değerini sokar. Bu elemanı gösteren bir iteratör döndürür. Bu eleman eğer mevcut değilse haritaya sokulur. Eğer eleman araya konmuşsa pair<iterator, true> döndürülür. Aksi halde pair<iterator, false> döndürülür. |
| iterator lower_bound(const key_type &k); | Anahtarları karşılaştırılan fonksiyon nesnesi döndürür.                                                                                                                                                                                |
| const_iterator                           | k'ya eşit veya k'dan daha büyük anahtarın olan haritadaki ilk elemanı gösteren bir iteratör döndürür.                                                                                                                                  |
| lower_bound(const key_type &k) const;    |                                                                                                                                                                                                                                        |
| size_type max_size( ) const;             |                                                                                                                                                                                                                                        |
| reference operator[ ](const key_type &i) | Haritanın kaldırabileceği maksimum eleman sayısını döndürür.                                                                                                                                                                           |
| reverse_iterator rbegin( );              | i tarafından belirlenen elemanı gösteren bir referans döndürür. Eğer bu eleman yoksa araya eklenir.                                                                                                                                    |
| const_reverse_iterator rbegin( ) const;  |                                                                                                                                                                                                                                        |
| reverse_iterator rend( );                |                                                                                                                                                                                                                                        |
| const_reverse_iterator rend( ) const;    | Haritanın başını gösteren ters bir iteratör döndürür.                                                                                                                                                                                  |
| size_type size( ) const;                 |                                                                                                                                                                                                                                        |
| void swap(map<Key, T, Comp,              | Haritadaki mevcut eleman sayısını döndürür.                                                                                                                                                                                            |
| Allocator> &ob)                          | Uyarı haritada saklanan elemanları ob'un içersindeki değişti.                                                                                                                                                                          |
| iterator upper_bound(const key_type &k); | k'dan daha büyük anahtarın olan haritadaki ilk elemanı gösteren iteratörü döndürür.                                                                                                                                                    |
| const_iterator                           |                                                                                                                                                                                                                                        |
| upper_bound(const key_type &k) const;    |                                                                                                                                                                                                                                        |
| value_compare value_comp( ) const;       | Değerleri karşılaştırılan fonksiyon nesnesini döndürür.                                                                                                                                                                                |

Tablo 14.4 map üye fonksiyonları

Anahtar/değer çiftleri pair tipinde nesneler gibi haritada saklanır. Şablon gösterimi de şu şekildedir.

```
template <class Ktype, class Vtype> struct pair {
 typedef Ktype first_type; // anahtarın tipi
 typedef Vtype second_type; // değer tipi
 Ktype first; // Anahtarı içerir
 Vtype second; // Değeri içerir

 // yapılandırıcılar
 pair();
 pair(const Ktype &k, const Vtype &v);
 template<class A, class B> pair<const<A, B> &ob>;
}
```

Açıklamalarında gösterdiği gibi `first`'in içerisindeki değer anahtarı içerir ve `second`'ın içerisindeki değerde bu anahtarla ilişkili değeri içerir.

`pair`'in constructor'larından birini kullanarak, ya da parametre olarak kullanılan veri tipine dayanmış bir `pair` nesnesi oluşturan `make_pair()`'ı kullanarak, bir `pair`(çift) oluşturabilirsiniz. `make_pair()` aşağıdaki prototipe sahip sosyal bir fonksiyondur.

```
template <class Ktype, class Vtype>
pair<Ktype, Vtype> make_pair(const Ktype &k, const Vtype &v);
```

Bu fonksiyon, sizinde görebileceğiniz gibi *Ktype* ve *Vtype* tarafından belirlenen tip değerlerinden oluşan bir `pair` nesnesi döndürür. `make_pair()`'ın avantajı şudur: Saklanmakta olan nesnelerin tiplerini sizin tarafınızdan açıkça belirtilmesi yerine derleyici tarafından otomatik olarak belirlenmesini sağlar.

## Örnekler

1. Aşağıdaki programda harita kullanmanın temellerini görebilirsiniz. Bu program 10 adet anahtar/değer çifti saklar. Anahtar bir karakterdir ve değer bir tamsayıdır. Saklanan anahtar/değer çiftleri şu şekildedir:

|   |   |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |

ve böyle gider... Çiftler saklandığında kullanıcıdan bir anahtar istenir. (Ör: A ile J arasından bir harf) ve bu anahtarla ilişkili değer gösterilir.

```
// Basit bir harita örneği
#include <iostream>
#include <map>
using namespace std;

int main()
{
 map<char, int> m;
 int i;

 // çiftleri haritaya koyar
 for(i=0; i<10; i++) {
 m.insert(pair<char, int>('A'+i, i));
 }

 char ch;
 cout << "Enter key: ";
 cin >> ch;

 map<char, int>::iterator p;

 // verilen anahtarın değerini bul
 p = m.find(ch);
 if(p != m.end())
 cout << p->second;
```

```

 else
 cout << "Key not in map.\n";
 return 0;
}

```

**pair** şablon sınıfının anahtar/değer çiftlerinin oluşturulmasında kullanıldığına dikkat edin. **pair** tarafından belirlenen veri tipleri, çiftlerin arasına eklendikleri haritadakilerle uymalıdır.

Haritaya anahtar ve değerlerle ilk değerler verildiğinde, **find()** fonksiyonunu kullanarak anahtarı verilmiş değeri arayabilirsiniz. **find()** uyan elemanı gösteren veya anahtar bulunmadıysa haritanın sonunu gösteren bir iteratör döndürür. Uyan bulunduğuanda anahtarla ilişkili değer **pair**'ın **second** üyesinde saklanır.

2. Bir önceki örnekte anahtar/değer çiftleri **pair<char, int>** kullanılarak açıkça oluşturuluyordu. Bu yöntemde yanlış bir şey olmasa da çoğunlukla, parametre olarak kullanılan veri tipine dayanarak bir **pair** nesnesi oluşturan **make\_pair()**'ı kullanmak daha kolaydır. Örneğin, bir önceki programı düşünecek olursak bu kod satırı anahtar/değer çiftlerini **m**'in içine ekleyecektir:

```
m.insert(make_pair((char)('A'+i), i));
```

Burada **char**'a yapılan tip dönüşümü **i**, "A"ya eklendiğinde **int**'e otomatik olarak yapılan dönüşümün etkisini yok etmek için gereklidir. Aksi halde hangi tipte olduğuna otomatik olarak karar verilir.

3. Tüm konteynırlar gibi haritalarda sizin oluşturduğunuz tipte nesneleri saklamak için kullanılabilir. Örneğin burada verilen program kelimeler ve kelimelerin tersini içeren bir harita oluşturur. Bunu yapmak için **word** ve **opposite** isminde iki sınıf oluşturur. Harita anahtarlarının sıralı bir listesini sakladığından, program aynı zamanda **word** tipinde nesneler için < operatörünü de tanımlar. Genelde < operatörünü anahtar olarak kullanacağınız tüm sınıflarda tanımlamalısınız. (Bazı derleyiciler bu ek karşılaştırma operatörlerinin tanımlanmasını gerektirebilir.)

```

// Terslerin haritası
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class word {
 char str[20];
public:
 word() { strcpy(str, ""); }
 word(char *s) { strcpy(str, s); }
 char *get() { return str; }
};

// word nesneleriyle ilişkili olarak küçuktur operatörü tanımlanmalıdır.
bool operator<(word a, word b)
{
 if (a.get() < b.get())
 return true;
 else
 return false;
}

```

```
 return strcmp(a.get(), b.get()) < 0;
}

class opposite {
 char str[20];
public:
 opposite() { strcmp(str, ""); }
 opposite(char *s) { strcpy(str, s); }
 char *get() { return str; }
};

int main()
{
 map<word, opposite> m;

 // haritaya word'ler ve opposite'ler konur
 m.insert(pair<word, opposite>(word("yes"), opposite("no")));
 m.insert(pair<word, opposite>(word("good"), opposite("bad")));
 m.insert(pair<word, opposite>(word("left"), opposite("right")));
 m.insert(pair<word, opposite>(word("up"), opposite("down")));

 // kelime verildi tersini bul
 char str[80];
 cout << "Enter word: ";
 cin >> str;

 map<word, opposite>::iterator p;

 p = m.find(word(str));
 if(p != m.end())
 cout << "Opposite: " << p->second.get();
 else
 cout << "Word not in map.\n";

 return 0;
}
```

Bu örnekte haritadaki her girdi boş ile sonlandırılmış karakter dizileri taşır. Bu bölümde daha sonra programı, **string** standart tipini kullanan, daha kolay bir yolla yeniden yazıldığını göreceksiniz.

## Aliştırmalar

1. Örnekler üzerinde küçük değişiklikler yaparak çalışınız.
2. İsimleri ve telefon numaralarını içeren bir harita oluşturun. İsimlere ve numaralara programınız tarafından girilebilsin, değiştirilebilsin ve istenen bir isim bulunabil sin. (İpucu: Model olarak örnek 3'ü alın)
3. Harita içerisinde anahtar olarak kullanılan nesneler için < operatörünü tanımlamanız gereklidir mi?

## 14.6. Algoritmalar

Daha önce anlatıldığı gibi, algoritmalar konteynırlar üzerinde işlerler. Her konteynır kendi temel işlemleri için destek sağlasa da standart algoritmalar daha genişletilmiş ve karışık hizmetler sunar. Aynı zamanda size iki farklı tipte konteynırla aynı anda çalışma şansı verir. STL algoritmalarına erişmek için programınıza `<algorithm>` başlığını eklemelisiniz. STL, Tablo 14-5'te kısaltılmış olarak verilen çok sayıda algoritma tanımlar. Bütün algoritmalar şablon sınıflarıdır. Yani bu demektir ki her tip konteynırda uygulanabilirler. Tabloyu takip eden örneği inceleyin:

| Algoritma               | Amaç                                                                                                         |
|-------------------------|--------------------------------------------------------------------------------------------------------------|
| adjacent_find           | Birbirine yakın uyumlu elemanları arar ve ilk uyumluya bir iteratör döndürür.                                |
| binary_search           | Sıralanmış bir süreçte ikili arama gerçekleştirir.                                                           |
| copy                    | Süreci kopyalar.                                                                                             |
| copy_backward           | copy( ) ile aynıdır ancak ilk önce sürecin sonundan elemanları atar.                                         |
| count                   | Sürec içerisindeki eleman sayısını döndürür.                                                                 |
| count_if                | Bazi doğrulamalardan geçen elemanların sayısını döndürür.                                                    |
| equal                   | İki aralığın aynı olduğunu saptar.                                                                           |
| equal_range             | Sürecin sırasını bozmadan bir elemanın sürece girebileceği aralığı döndürür.                                 |
| fill                    | Bir aralığı belirtilen değerle doldurur.                                                                     |
| fill_n                  |                                                                                                              |
| find                    | Bir aralığı bir değer için arar ve bulunduğu ilk elemana bir iteratör döndürür.                              |
| find_end                | Alt süreç için bir aralığı arar. Bu fonksiyon aralık içerisindeki alt sürecin sonuna iteratör döndürür.      |
| find_first_of           | Sürec içerisinde, aralık içerisindeki elemanla uyusan ilk elemani bulur.                                     |
| find_if                 | Kullanıcı tarafından tanımlanmış mantık ifadesinin true döndürmesi için bir elemani aralık içerisinde arar.  |
| for_each                | Aralıktaki elemlara fonksiyonları uygular.                                                                   |
| generate                | Üretici fonksiyondan gelen değerleri aralıktaki elemlara atar.                                               |
| generate_n              |                                                                                                              |
| includes                | Bir sürecin elemlerinin tümünü diğer sürecin içerişip içermeyebine bakar.                                    |
| inplace_merge           | Bir aralığı diğeryle birleştirir. Her iki aralıkta artan sıradı olmalıdır. Sonuçta oluşan süreçte sıralıdır. |
| iter_swap               | İki iteratör argümanı tarafından gösterilen değerleri değiştirir.                                            |
| lexicographical_compare | Bir süreci diğeryle alfabetik olarak karşılaştırır.                                                          |
| lower_bound             | Belirtilen değerden az olmayan ilk noktayı bir süreç içerisinde bulur.                                       |
| make_heap               | Bir süreçten heap oluşturur.                                                                                 |
| max                     | İki değerden maksimumu döndürür.                                                                             |
| max_element             | Aralık içerisindeki maksimum değerdeki elemana bir iteratör döndürür.                                        |
| merge                   | İki sıralanmış süreci üçüncü bir süreç oluşturarak birleştirir.                                              |
| min                     | İki değerden en küçüğünü döndürür.                                                                           |
| min_element             | Aralık içerisindeki en küçük elemana bir iteratör döndürür.                                                  |

|                                       |                                                                                                                                                                                                        |
|---------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>mismatch</code>                 | İki süreç içerisinde, elemanlarda ilk uyumsuzluğu bulur. İki elemana iteratörler döndürür.                                                                                                             |
| <code>next_permutation</code>         | Sürecin bir sonraki permutasyonunu döndürür.                                                                                                                                                           |
| <code>nth_element</code>              | Süreci, belirtilen E elemanından küçükleri öne, büyükleri sona koyarak sıralar.                                                                                                                        |
| <code>partial_sort</code>             | Aralığı sıralar                                                                                                                                                                                        |
| <code>partial_sort_copy</code>        | Bir aralığı sıralar ve sonuç süreçte siğabilecek bütün elemanları kopyalar.                                                                                                                            |
| <code>partition</code>                | Bir süreçteki, mantık ifadesinde true olarak belirlenen elemanları başa, false olarak belirlenenleri sona koymak suretiyle süreci sıralar.                                                             |
| <code>pop_heap</code>                 | İlk ve son elemanları değiştirir ve heap'i yeniden yapılandırır.                                                                                                                                       |
| <code>prev_permutation</code>         | Bir sürecin önceki permutasyonunu yapılandırır.                                                                                                                                                        |
| <code>push_heap</code>                | Heap'in sonuna bir elemanı iter.                                                                                                                                                                       |
| <code>random_shuffle</code>           | Süreci karıştırır.                                                                                                                                                                                     |
| <code>remove</code>                   | Belirtilen aralıktaki elemanları çıkarır.                                                                                                                                                              |
| <code>remove_if</code>                |                                                                                                                                                                                                        |
| <code>remove_copy</code>              |                                                                                                                                                                                                        |
| <code>remove_copy_if</code>           |                                                                                                                                                                                                        |
| <code>replace</code>                  | Belirtilen aralıktaki elemanları değiştirir.                                                                                                                                                           |
| <code>replace_copy</code>             |                                                                                                                                                                                                        |
| <code>replace_if</code>               |                                                                                                                                                                                                        |
| <code>replace_copy_if</code>          |                                                                                                                                                                                                        |
| <code>reverse</code>                  | Aralığın sırasını tersine çevirir.                                                                                                                                                                     |
| <code>reverse_copy</code>             |                                                                                                                                                                                                        |
| <code>rotate</code>                   | Aralıktaki elemanları sola döndürür.                                                                                                                                                                   |
| <code>rotate_copy</code>              |                                                                                                                                                                                                        |
| <code>search</code>                   | Bir süreçteki alt süreci arar.                                                                                                                                                                         |
| <code>search_n</code>                 | Bir süreci belirtilen sayıdaki benzer elemanlar için arar.                                                                                                                                             |
| <code>set_difference</code>           | İki sıralanmış grup arasındaki farkı içeren bir süreç oluşturur.                                                                                                                                       |
| <code>set_symmetric_difference</code> | İki sıralanmış grup arasındaki simetrik farkı içeren bir süreç oluşturur.                                                                                                                              |
| <code>set_union</code>                | İki sıralanmış grubun birleşimini içeren bir süreç oluşturur.                                                                                                                                          |
| <code>sort</code>                     | Bir aralığı sıralar.                                                                                                                                                                                   |
| <code>sort_heap</code>                | Belirtilen aralıktaki heap'i sıralar.                                                                                                                                                                  |
| <code>stable_partition</code>         | Bir süreçteki, mantık ifadesinde true olarak belirlenen elemanları başa, false olarak belirlenenleri sona koymak suretiyle süreci sıralar. Bölümleme durağandır. Sürecin ilişkili sıralaması saklanır. |
| <code>stable_sort</code>              | Bir aralığı sıralar. Sıralama durağandır. Eşdeğer elemanlar yeniden sıralanmaz.                                                                                                                        |
| <code>swap</code>                     | İki değeri değiştirir                                                                                                                                                                                  |
| <code>swap_ranges</code>              | Aralıktaki elemanları değiştirir.                                                                                                                                                                      |
| <code>transform</code>                | Bir aralıktaki elemanlara bir fonksiyon uygular ve geri geleni bir süreç içerisinde saklar.                                                                                                            |
| <code>unique</code>                   | Aralıktaki çift olan değerleri eler.                                                                                                                                                                   |
| <code>unique_copy</code>              |                                                                                                                                                                                                        |
| <code>upper_bound</code>              | Sürec içerisinde bir değerden daha büyük olmayan son noktayı bulur.                                                                                                                                    |

Tablo 14-5 STL algoritmaları

## Örnekler

- En basit algoritmaların ikisi **count( )** ve **count\_if( )**'dır. Genel formlarını aşağıda veriyorum.

```
template <class InIter, class T>
size_t count(InIter start, InIter end, const T &val);

template <class InIter, class T>
size_t count(InIter start, InIter end, UnPred pfn);
```

**count( )** algoritması *start* ve *end* ile sınırlanmış süreçte, *val* değeri ile uyusan elemanların toplam sayısını döndürür. **count\_if( )** algoritması *start* ve *end* ile sınırlanmış süreçte mantıksal ifade *pfn*'nin true döndürdüğü elemanların toplam sayısını döndürür.

Aşağıdaki program **count( )** ve **count\_if( )** fonksiyonlarının kullanımı hakkında size bir fikir verecektir:

```
// count ve count_if
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

/* This is a unary predicate that determines if a value is even. */
bool even(int x)
{
 return !(x%2);
}

int main()
{
 vector<int> v;
 int i;

 for(i=0; i<20; i++) {
 if(i%2) v.push_back(1);
 else v.push_back(2);
 }

 cout << "Sequence: ";
 for(i=0; i<v.size(); i++) cout << v[i] << " ";
 cout << endl;

 int n;
 n = count(v.begin(), v.end(), 1);
 cout << n << " elements are 1\n";

 n = count_if(v.begin(), v.end(), even);
 cout << n << " elements are even.\n";

 return 0;
}
```

Program aşağıdaki çıktıyi verecektir:

```
Sequence: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
10 elements are 1
10 elements are even.
```

Program 1'ler ve 2'ler içeren 20 elemanlı bir vektör oluşturarak işe başlayacak. Daha sonra, **count( )** 1'lerin sayısını ve **count\_if( )** de çift olan elemanların sayısını bulacak. **Even( )** fonksiyonun nasıl kodlandığında dikkat edin. Bütün birli mantıksal fonksiyonlar (predicate, boolean), üzerinde çalıştığı konteynırın tipi ile aynı tipte nesneleri parametre olarak alır. Mantıksal ifade daha sonra bu nesneye göre true veya false değeri döndürmelidir.

2. Bazen ilk süreçten bazı seçilmiş elemanlarla yeni bir süreç oluşturmak yararlı bir iş olabilir. Bunu gerçekleştiren bir algoritma da **remove\_copy( )** algoritmasıdır. Genel formu aşağıdaki gibidir:

```
template <class InIter, class OutIter, class T>
OutIter remove_copy(InIter start, InIter end,
OutIter result, const T &val);
```

**remove\_copy( )** algoritması *val*'e eşit olan elemanları, *result* tarafından gösterilen süreçce belirtilen aralıkta kopyalar. Sonucun sonuna bir iteratör döndürür. Çıkış konteynırı sonucu taşıyabilecek kadar büyük olmalıdır.

Aşağıdaki program **remove\_copy( )** ile ilgilidir. 1'ler ve 2'lerle dolu bir süreç oluşturur. Daha sonra bütün 1'leri süreçten atar.

```
// remove_copy.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
 vector<int> v, v2(20);
 int i;

 for(i=0; i<20; i++) {
 if(i%2) v.push_back(1);
 else v.push_back(2);
 }

 cout << "Sequence: ";
 for(i=0; i<v.size(); i++) cout << v[i] << " ";
 cout << endl;

 // 1'leri at
 remove_copy(v.begin(), v.end(), v2.begin(), 1);
 cout << "Result: ";
 for(i=0; i<v2.size(); i++) cout << v2[i] << " ";
 cout << endl << endl;

 return 0;
}
```

Program tarafından üretilen çıktı aşağıdaki gibidir:

```
Sequence: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
Result: 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0
```

- Bazı durumlarda bir süreci tersine çeviren **reverse()** algoritması da kullanışlı olabilir. Genel formu,

```
template <class BiIter> void reverse(BiIter start, BiIter end);
```

şeklindedir. **reverse()** algoritması *start* ve *end* tarafından belirtilen aralıktaki elemanların sırasını ters çevirir. Aşağıdaki program **reverse()** ile ilgilidir:

```
// reverse.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
 vector<int> v;
 int i;

 for(i=0; i<10; i++) v.push_back(i);

 cout << "Initial: ";
 for(i=0; i<v.size(); i++) cout << v[i] << " ";
 cout << endl;

 reverse(v.begin(), v.end());

 cout << "Reversed: ";
 for(i=0; i<v.size(); i++) cout << v[i] << " ";
}

return 0;
}
```

Program tarafından üretilen çıktı aşağıdaki gibidir:

```
Initial: 0 1 2 3 4 5 6 7 8 9
Reversed: 9 8 7 6 5 4 3 2 1 0
```

- Diger ilginç algoritmalarlardan biri de belirttiğiniz bir fonksiyona göre verilen bir aralıktaki her elemanı değiştiren **transform()** algoritmasıdır. **transform()** algoritmasının iki genel formu vardır:

```
template <class InIter, class OutIter, class Func>
OutIter transform(InIter start, InIter end, OutIter result, Func unaryfunc);

template <class InIter1, class InIter2, class OutIter, class Func>
OutIter transform(InIter1 start1, InIter1 end1, InIter2 start2,
OutIter result, Func binaryfunc);
```

**transform( )** algoritması aralıktaki elemanlara bir fonksiyon uygular ve sonucu *result*'a koyar. İlk formda aralık *start* ve *end* ile, uygulanacak fonksiyon *unaryfunc* ile belirtilir. Bu fonksiyon parametresindeki elemanın bir değerini alır. Elemanın dönüşümünü döndürmek zorundadır. İkinci formda ise dönüşüm ikili operatör fonksiyonu kullanarak uygulanır. İlk parametresinde süreçten, dönüştürülecek elemanın değerini alır, ikinci parametre olarak ikinci süreçten elemanı alır. Her iki fonksiyonda sonuç sürecinin sonuna bir iteratör döndürür.

Aşağıdaki program bir listedeki değerlerin karesini alan **xform( )** adında basit bir dönüşüm fonksiyonu kullanır. Sonuç sürecinin, ilk süreç tarafından sağlanan aynı listede saklandığında dikkat edin.

```
// Dönüşüm algoritması
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

// Basit bir dönüşüm fonksiyonu
int xform(int i) {
 return i * i; // İlk değerin karesini al
}

int main()
{
 list<int> xl;
 int i;

 // put values into list
 for(i=0; i<10; i++) xl.push_back(i);

 cout << "Original contents of xl: ";
 list<int>::iterator p = xl.begin();
 while(p != xl.end()) {
 cout << *p << " ";
 p++;
 }
 cout << endl;

 // xl'yi dönüştür.
 p = transform(xl.begin(), xl.end(), xl.begin(), xform);

 cout << "Transformed contents of xl: ";
 p = xl.begin();
 while(p != xl.end()) {
 cout << *p << " ";
 p++;
 }
 return 0;
}
```

Program aşağıdaki çıktıyi verir:

```
Original contents of xl: 0 1 2 3 4 5 6 7 8 9
Transformed contents of xl: 0 1 4 9 16 25 36 49 64 81
Gordüğünüz gibi xl'deki her elemanın karesi alınmıştır.
```

## Aliştırmalar

- sort( ) algoritmasının şu formları vardır:

```
template <class RandIter> void sort(RandIter start, RandIter end);
template <class RandIter, class Comp>
void sort(RandIter start, RandIter end, Comp cmpfn);
```

*start* ve *end* tarafından belirlenen aralığı sıralar. İkinci form bir elemanın diğerinden az olup olmadığını belirlemeniz için bir karşılaştırma fonksiyonu belirtmenize imkan verir. Sort( )'un işleyişini gösteren bir program hazırlayın. (İstediğiniz formu kullanabilirsiniz.)

- merge( ) algoritması sıralanmış iki süreci birleştirip sonucu bir üçüncü sürece koyar. Genel formlarından biri aşağıdaki gibidir.

```
template <class InIter1, class InIter2, class OutIter>
OutIter merge(InIter1 start1, InIter1 end1, InIter2 start2, InIter2 end2,
OutIter result);
```

Birleştirilecek süreçler *start1*, *end1* ve *start2*, *end2* ile belirtilmiştir. Sonuç *result* tarafından gösterilen bir sürec konur. Sonuç sürecine bir iteratör döndürülür. Bu algoritmayı kullanan bir program hazırlayın.

## 14.7. string Sınıfı

Bildiğiniz gibi C++, mevcut katar tiplerini desteklemez. Ancak katarları işlemek için iki yol sunar. Birincisi aşina olduğunuz, geleneksel olan, boş ile sonlandırılmış karakter dizilerini kullanabilmenizdir. Bu bazı zamanlar C katarı olarak da bilinir. İkinci yol ise string tipinden bir sınıf nesnesi kullanmanızdır. Burada incelenen yaklaşım budur.

Normalde **string** sınıfı, **basic\_string** adında daha genel şablon sınıfının özelleştirilmiş halidir. Aslında **basic\_string** sınıfının iki özelleştirilmiş hali vardır. Bunlardan biri 8 bitlik karakterleri destekleyen **string**, diğeri ise geniş karakter katarlarını destekleyen **wstring**'tir. Normal programlamada en genel şekilde 8 bitlik karakter tipleri kullanıldığından, burada **basic\_string**'in **string** sürümünü inceleyeceğiz.

**string** sınıfını incelemeden önce, neden C++ kütüphanesinin bir parçası olduğunu anlamanız gereklidir. Standart sınıflar C++'a sebepsiz yere eklenmemiştir. Her yeni eklenti için uzun çalışmalar yapılmış ve fikirler ortaya atılmıştır. C++ boş sonlu karakter dizileri olarak, katarlar için zaten destek sağlar. **string** sınıfının dahil edilmesi ilk başta size bu kurallın geçersiz olduğu imajını uyandırabilir. Ancak, bu gerçekten çok uzaktır. Sebep şudur: Boş sonlu katarlar herhangi bir C++ operatörü tarafından işlenemez. Ya da normal C++ ifadelerinde yerlerini alamaz. Örneğin şu kodu dikkatlice inceleyin.

```
char s1[80], s2[80], s3[80];
s1 = "one"; // yapamaz
s2 = "two"; // yapamaz
s3 = s1 + s2; // hata ixin verilmeyaz
```

Açıklamaların gösterdiği gibi C++'da bir karakter dizisine değer atamak için atama operatörünü, (Başlangıç değeri atama durumu hariç). Ya da iki katarı birleştirmek için + operatörünü kullanamazsınız. Bu işlemler aşağıda gösterildiği gibi kütüphane fonksiyonları ile gerçekleştirilmelidir.

```
strcpy(s1, "one");
strcpy(s2, "two");
strcpy(s3, s1);
strcat(s3, s2);
```

Boş sonlu karakter dizileri kendilerince teknik olarak veri tipleri olmadıklarından, C++ operatörleri onlara uygulanamaz. Bu durum en rahat katar işlemlerini bile zora sokar. Her şeyden öte, standart **string** sınıfının gelişimi ile ortaya çıkan standart C++ operatörlerini kullanarak boş sonlu katarlar üzerinde çalışılabilme imkanı doğar. Şunu unutmayın ki C++'da bir sınıf tanımladığınız da, siz, C++ ortamına tamamen entegre olabilen yepyeni bir veri tipi oluşturmuş olursunuz. Tabi ki bu operatörlerin yeni sınıfa bağlı olarak aşırı yüklenemeleri manasına gelir. Böylece standart **string** sınıfının eklenmesi ile operatörleri kullanarak diğer veri tiplerinin yönetimi ile aynı şekilde katarları işlemek mümkün olmuştur.

Ancak, standart **string** sınıfı aynı zamanda da güvenlidir. Tecrübesiz veya dikkatsiz bir programcı boş sonlu katar taşıyan karakter dizisinin sonunu aşabilir (sınırlarının dışına çıkabilir). Örneğin, standart katar kopyalama fonksiyonu olan **strcpy()**'yi düşünün. Bu fonksiyon hedef dizî için herhangi bir sınır kontrolü gerçekleştirmez. Eğer kaynak dizî hedef dizinin taşıyabileceğinden daha fazla karakter içeriyorsa bir program hatası veya sistem çökmesi meydana gelir(bir istimal) Göreceğiniz gibi, standart **string** sınıfı bu tip hataları önler.

Son olarak **string** sınıfını dahil etmek için 3 temel sebep var: kalıcılık( bir katar artık bir veri tipi tanımlıyor), uygunluk (standart C++ operatörlerini artık kullanabilirsiniz) ve güvenlik (sınırlar aşılmıyor). Şunu da aklınızda bulundurun ki normal boş sonlandırılmış katarların tümünden vazgeçmeniz için bir sebep yok. Onlar hala katarları belirtmek için en verimli yol. Ancak, hız problemi yoksa yeni **string** sınıfı size katarları yönetmek için güvenli ve tam bir entegrasyon sağlar.

Genellikle STL'nin bir parçası gibi düşünülmese de **string** sınıfı, C++'da tanımlanmış başka bir konteynördür. Bu onun önceki konuda anlatılan algoritmaları desteklediği anlamına gelir. Ancak katarların daha farklı yetenekleri de vardır. **string** sınıfına erişmek için **<string>** başlığını programınıza eklemelisiniz.

**string** sınıfı bir çok yapılandırıcı ve üye fonksiyonları ile çok genişir. Ayrıca çok sayıda üye fonksiyon çoklu aşırı yüklenmiş formları içerir. bu sebeple, **string**'in tüm içeriğine bu bölümde bakmak mümkün değildir. Onun yerine en önemli ve sık kullanılan birkaç özelligidenden bahsedeceğiz. **string**'in genel olarak nasıl işlediğini kavramışsanız gerisini kolayca kendi başınıza anlayabilirsiniz.

**string** sınıfı birkaç constructor (yapılendirici) destekler. En sık kullanılan ilk üçünün prototipi aşağıdaki gibidir:

```
string();
string(const char *str);
string(const string &str);
```

İlk form boş bir **string** nesnesi oluşturur. İkincisi *str* tarafından gösterilen boş sonlu katarlardan **string** nesnesi oluşturur. Bu form boş sonlu katarlardan **string** nesnelerine dönüşüm sağlar. Üçüncü form ise bir **string** nesnesinden diğer bir **string** nesnesini oluşturur.

**string** nesneleri için tanımlanan ve katarlara uygulanan operatörler aşağıdaki tabloda verilmiştir:

| Operator | Anlamı                                   |
|----------|------------------------------------------|
| =        | Atama                                    |
| ++       | Ekleme(Concatenation)                    |
| +=       | Ekleme ataması(Concatenation assignment) |
| ==       | Eşitlik                                  |
| !=       | Eşit değil                               |
| <        | ...dan az                                |
| <=       | ...dan az veya ona eşit                  |
| >        | ...dan büyük                             |
| >=       | ...dan büyük veya ona eşit               |
| []       | Dizi başvuru operatörü                   |
| <<       | Çıkar                                    |
| >>       | Giriş                                    |

Bu operatörler **string** nesnelerinin normal ifadelerde kullanımına izin verir. **strcpy()** ve **strcat()** gibi fonksiyonlara çağrı yapılması gerektiğini ortadan kaldırır. Genelde **string** nesnelerini ifadelerde normal boş sonlandırılmış katarlarla karıştırabilirsiniz. Örneğin bir **string** nesnesi boş sonlandırılmış katara atanabilir.

+ operatörü kullanarak, bir **string** nesnesini diğeriley veya bir C tipi katarla birleştirebilirsınız. Bu sayede aşağıdaki ifade çeşitleri desteklenmektedir.

```
string + string
string + C-string
C-string + string
```

+ operatörü bir karakteri bir katarın sonuna eklemek için de kullanılabilir.

**string** sınıfı genelde -1 olan **npos** sabitini tanımlamaktadır. Bu sabit mümkün olan en uzun katarın uzunluğunu temsil eder.

Basit katar işlemlerinin çoğu katar operatörleri ile yapılabilir. Fakat daha karmaşık ve kapsamlı olanları **string** sınıfının üye fonksiyonları tarafından gerçekleştiriliyor. Bu fonksiyonların hepsini bu bölümde ele almamız imkansız da olsa birkaç tanesini burada inceleye-

ceğiz. Bir katarı diğerine atamak için `assign( )` fonksiyonunu kullanın. Bu fonksiyonun kullanım şekillerinden ikisi şunlardır:

```
string &assign(const string &strb, size_type start, size_type num);
string &assign(const char *str, size_type num);
```

İlk formda, `strb`'tan, `start` tarafından belirlenen indeksten `num` kadar karakter başlayarak uyaran nesneci atanır. İkincisinde boş sonlandırılmış `str` katarının ilk `num` sayıda karakteri uyarıcı nesneye atanır. Her iki durumda da uyarıcı nesneyi gösteren bir referans döndürülür. Elbette, bir katarın tamamını diğerine = operatörünü kullanarak atamak çok daha kolaydır. `assign( )` fonksiyonunu sadece katarın bir parçasını atarken kullanmanız gerekecek.

`append( )` üye fonksiyonunu kullanarak bir katarın bir parçasını diğerine ekleyebilirsiniz. Bu formların ikisi aşağıda gösterilmiştir:

```
string &append(const string &strb, size_type start, size_type num);
string &append(const char *str, size_type num);
```

İlkinde `strb`'tan `start` tarafından belirlenen indeksten başlayarak `num` sayıda karakter uyarıcı nesneye eklenir. İkincisinde boş sonlandırılmış `str` katarının ilk `num` sayıda karakter, uyarıcı nesneye eklenir. Her durumda uyaran nesneyi gösteren bir referans döndürülür. Tabii ki, katarın tamamını bir diğerine eklemek için + operatörünü kullanmak çok daha kolaydır. `append( )` fonksiyonunu sadece katarın bir parçasını atarken kullanmanız gerekecek.

`insert( )` ve `replace( )`'yi kullanarak karakterleri katar içine sokabilirsiniz veya bunları değiştirebilirsiniz. En bilinenlerinin prototiplerini aşağıda veriyorum :

```
string &insert(size_type start, const string &strb);
string &insert(size_type start, const string &strb,
size_type insStart, size_type num);
string &replace(size_type start, size_type num, const string &strb);
string &replace(size_type start, size_type orgNum, const string &strb,
size_type replaceStart, size_type replaceNum);
```

`insert( )`'in ilk formu uyaran katara `start` tarafından belirlenen indeksinde `strb`'u sokar. İkincisinde ise `insStart`'dan başlayacak şekilde `strb`'tan, `num` sayıda karakteri, `start` tarafından belirtilen indeks noktasında, uyaran katara sokar. Her durumda uyaran nesneyi gösteren bir referans döndürülür. Tabii ki, katarın tamamını bir diğerine eklemek için + operatörünü kullanmak çok daha kolaydır. `append( )` fonksiyonunu sadece katarın bir parçasını atarken kullanmanız gerekecek. `replace()`'in ilk formunda ise `start`'dan başlayarak uyaran katardan `num` sayıda karakteri `strb` ile değiştirir. İkinci formda, uyaran katarın `start`'dan başlayarak, `orgNum` sayıda karakteri, `strb` tarafından belirlenen katardaki `replaceNum` sayıda karakterle değiştirir. Her iki durumda da uyaran nesneci referans döndürülür.

`erase( )`'yi kullanarak bir katardan karakterler atabilirsiniz. Bu formlardan biri şu şekildedir :

```
string gerase(size_type start = 0, size_type num = npos);
```

Yukarıdaki ifade *num* sayıda karakteri *start*'tan başlayarak uyaran katarдан atar. Uyaran nesneci bir referans döndürülür.

**string** sınıfı katar arayan **find()** ve **rfind()** gibi bazı üye fonksiyonları sağlar. Bu iki fonksiyonun en sık kullanılan prototipleri şunlardır:

```
size_type find(const string &strob, size_type start=0) const;
size_type rfind(const string &strob, size_type start=npos) const;
```

*start*'tan başlayarak *strob*'ta saklanan katarı bulmak için **find()** uyaran katarı arar. Eğer uyaran katarın katarda aranan bulunursa, bulunduğu noktanın indeksi **find()** tarafından döndürülür. Eğer bir şey bulunamazsa **npos** döndürülür. **rfind()** ise **find()**'ın tersidir. *Start*'tan başlayarak uyaran katarı, *strob*'ta saklanan katarı bulmak için ters yönde arar. (Böylece *strob*'ta bulunanın uyaran katarın en son görüldüğü noktayı bulur.) Eğer aranan bulunursa, **rfind()** uyaran katarın uyusun katarın indeksini döndürür. Eğer bir şey bulunamazsa **npos** döndürülür.

Bir katar nesnesindeki tüm içeriği bir diğerinin içeriğiyle karşılaştırmak için normalde daha önce anlatılan aşırı yüklenmiş ilişkili operatörleri kullanacaksınız. Ancak katarın bir parçasını diğeriyle karşılaştırmak istiyorsanız aşağıda gösterilen **compare()** üye fonksiyonunu kullanmanız gerekecek:

```
int compare(size_type start, size_type num, const string &strob) const;
```

Burada *start*'tan başlayarak *strob*'taki *num* kadar karakter, uyaran katarla karşılaştırılacaktır. Eğer uyaran katar *strob*'dan daha azsa, **compare()** 0'dan küçük bir değer, daha çoksa 0'dan büyük bir değer döndürür. Eğer *strob* uyaran katarla eşitse **compare()** 0 döndürür.

**string** nesneleri kendilerince kullanışlı olsa da, katarların boş sonlu karakter dizileri sürümünü kullanmanız gereken durumlar da ortaya çıkacaktır. Örneğin bir **string** nesnesini dosya ismi oluşturmak için kullanabilirsiniz. Ancak dosyayı açarken standart boş sonlu bir katarı gösteren işaretçi belirtmeniz gerekir. Bu problemi çözmek için **c\_str()** üye fonksiyonu hazırlanmıştır. Prototipi aşağıdaki gibidir :

```
const char *c_str() const;
```

Bu fonksiyon uyaran **string** nesnesinin içerisindeki katarın boş sonlu sürümünü gösteren bir işaretçi döndürür. Boş sonlandırılmış katar değiştirilmemelidir. Aynı zamanda **string** nesnesi üzerinde herhangi bir işlem gerçekleştiğten sonra geçerliliği garanti edilemez.

Bu bir konteynır olduğundan **string**, sırasıyla katarın başını ve sonunu gösteren iteratörler döndüren **begin()** ve **end()** fonksiyonlarını destekler. Katardaki mevcut karakter sayısını döndüren **size()** fonksiyonu da bunlara dahildir.

## Örnekler

1. Geleneksel C tipi katarların kullanımı her zaman olsa da C++ `string` sınıfı katar yönetimini çok daha basite indirger. Örneğin `string` nesneleri ile tırnaklı bir katar bir `string`'e atama operatörü kullanarak atayabilirsiniz. `+` operatörü kullanarak katarları birleştirebilir, karşılaştırma operatörleri ile de katar karşılaştırması yapabilirsiniz. Aşağıdaki program bu işlemleri gerçekleştiriyor.

```
// Katarlar için kısa bir program.
#include <iostream>
#include <string>
using namespace std;

int main()
{
 string str1("Demonstrating Strings");
 string str2("String Two");
 string str3;
 // katar atama
 str3 = str1;
 cout << str1 << "\n" << str3 << "\n";

 // iki katarı birleştirme
 str3 = str1 + str2;
 cout << str3 << "\n";

 // katar karşılaştırma
 if(str3 > str1) cout << "str3 > str1\n";
 if(str3 == str1+str2)
 cout << "str3 == str1+str2\n";

 /* string nesnesi normal bir katara da atanabilir. */
 str1 = "This is a normal string.\n";
 cout << str1;

 // bir string nesnesi kullanarak bir diğerini oluşturur
 string str4(str1);
 cout << str4;
 // katar girişi
 cout << "Enter a string: ";
 cin >> str4;
 cout << str4;

 return 0;
}
```

Bu program aşağıdaki çıktıyi verir :

```
Demonstrating Strings
Demonstrating Strings
Demonstrating StringsString Two
str3 > str1
str3 == str1+str2
This is a normal string.
This is a normal string.
Enter a string: Hello
Hello
```

Gördüğünüz gibi **string** tipinden nesneler C++'ın mevcut veri tipleri ile çalışan tekniklerin benzerleriyle yönetilebilir. Aslında katar sınıfının en büyük avantajı budur.

Katarlarla çalışmanın ne kadar basit olduğunu dikkat edin. Örneğin +, katarları birleştirmek için, >, iki katarı karşılaştırmak için kullanılır. Bu işlemleri C tipi boş sonlu katarlarla gerçekleştirmek için daha az uyumlu olan **strcat()** ve **strcmp()** fonksiyonlarını çağırmanız gereklidir. C++ **string** nesneleri, rahat bir şekilde C tipi boş sonlandırılmış katarlarla birlikte kullanılabilir. Bu nedenle onları programınızda kullanmanın bir zararı yoktur, hatta dikkate değer faydaları da vardır.

Bir önceki programda dikkat etmemiz gereken bir başka nokta da katar uzunluğunun belirtilmemiş olmasıdır. Bir **string** nesnesi verilen katarı tutmak için kendiliğinden boyutlandırılır. Böylelikle atama yaparken veya katarları birleştirirken hedef katar yeni katarın boyutunu karşılayabilecek kadar büyür. Katarın sınırının dışına çıkmak mümkün değildir. **string** nesnelerinin bu dinamik yönü standart boş sonlandırılmış katarlardan daha iyi olduğunun düşünülmesinin nedenidir (Sınır taşmaları konusunda).

2. Aşağıdaki program **insert()**, **erase()** ve **replace()** fonksiyonlarıyla ilgiliidir.

```
// insert(), erase() ve replace()'ı gösterir.
#include <iostream>
#include <string>
using namespace std;

int main()
{
 string str1("This is a test");
 string str2("ABCDEFG");
 cout << "Initial strings:\n";
 cout << "str1: " << str1 << endl;
 cout << "str2: " << str2 << "\n\n";
 // insert()
 cout << "Insert str2 into str1:\n";
 str1.insert(5, str2);
 cout << str1 << "\n\n";
 // erase()
 cout << "Remove 7 characters from str1:\n";
 str1.erase(5, 7);
 cout << str1 << "\n\n";
 // replace
 cout << "Replace 2 characters in str1 with str2:\n";
 str1.replace(5, 2, str2);
 cout << str1 << endl;
 return 0;
}
```

Programın çıktısı aşağıdaki gibidir :

```
str1: This is a test
str2: ABCDEFG

Insert str2 into str1:
```

```

This ABCDEFG is a test

Remove 7 characters from str1:
This is a test

Replace 2 characters in str1 with str2:
This ABCDEPG a test

```

3. **string** bir veri tipi tanımladığından **string** tipinden nesneleri taşıyan konteynırı oluşturmak mümkündür. Örnek olarak Konu 14.5, Örnek 3'te gösterilen kelimeyi tersine döndürme programın daha iyi hazırlanmış bir sürümünü verelim.

```

// katarlar kullanarak kelimeyi tersine çevirme.
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
 map<string, string> m;
 int i;
 m.insert(pair<string, string>("yes", "no"));
 m.insert(pair<string, string>("up", "down"));
 m.insert(pair<string, string>("left", "right"));
 m.insert(pair<string, string>("good", "bad"));
 string s;
 cout << "Enter word: ";
 cin >> s;
 map<string, string>::iterator p;
 p = m.find(s);
 if(p != m.end())
 cout << "Opposite: " << p->second;
 else
 cout << "Word not in map.\n";
 return 0;
}

```

## ALIŞTIRMALAR

1. **string** tipinden nesneler kullanarak aşağıdaki katarları **list'e** depolayın.

|     |       |       |      |      |
|-----|-------|-------|------|------|
| one | two   | three | Four | five |
| six | seven | eight | Nine | ten  |

Daha sonra listeyi sıralayın. Son olarak da sıralı listeyi gösterin.

2. **string** bir konteynır olduğundan standart algoritmalar ile birlikte kullanılabilir. Kullanıcıdan katar girişi isteyen bir program oluşturun. Daha sonra **count()**'u kullanarak katarda kaç "e" harfinin olduğunu görüntüleyin.
3. 2. alıştırmaya verdığınız cevabı kaç tane küçük harf olduğunu gösterecek şekilde değiştirin. (İpucu: **count\_if()**'i kullanın.)
4. **string** sınıfı hangi şablon sınıfının özelleştirilmiş halidir?

## Pekiştirme Testi

Bu noktada aşağıdaki alıştırmaları yapabiliyor ve soruları cevaplayabiliyor olmalısınız.

1. STL daha güvenilir programları oluşturmanızı nasıl kolaylaştırır?
2. Bir konteynır, iteratör ve algoritmanın STL ile olan ilişkisini tanımlayın.
3. 1'den 10'a kadar rakamları içerecek şekilde on elemanlı bir vektör oluşturan bir program hazırlayın. Daha sonra bu vektörden bir listeye sadece çift elemanları kopyalayın.
4. `string` veri tipi kullanmanın avantajlarından ve dezavantajlarından birini söyleyin.
5. predicate (mantıksal ifade) nedir?
6. Kendi başınıza Konu 14.5, Aşağıda 2'deki çözümü `string` kullanacak şekilde yeniden kodlayın.
7. STL fonksiyon nesnelerini araştırmaya başlayın. İşe başlarken fonksiyon nesnelerini yapılandırmakla görevli `unary_function` ve `binary_function` standart fonksiyonlarını inceleyin.
8. Derleyiciniz ile birlikte gelen STL dokümanlarını inceleyin. İlginç yenilikleri ve teknikleri orada bulabilirsiniz.

## Bütünleştirme Testi

Bu kısım sizin geçmiş böölümdeki konular da dahil olmak üzere, bu bölümde geçen konuları ne kadar kavradığınızı kontrol edecektir.

1. Bölüm 1'den bu yana çok uzun bir yol aldınız. Kitabı bir kere gözden geçirin. Eğer bunu yaparsanız örnekleri nasıl geliştirebileceğinizi düşünün (özellikle ilk altı böölümdeki) ve onların C++'ın öğrendiğiniz avantajlarından yararlanmazlığını sağlayın.
2. Programlama en iyi uygulama yaparak öğrenilir. Çok sayıda C++ programı yazın. Öğrendiğiniz C++'a özel yenilikleri ile alıştırmalar yapın.
3. STL'yi araştırmaya devam edin. Gelecekte birçok programlama işlemleri kavramları ile bölümlenecektir. Çünkü bir program algoritmalar tarafından konteynır işlevlerine dönüştürülebilir.
4. Son olarak C++'ın size sınırsız bir güç verdiği unutmayın. Bu gücü kullanmayı öğrenmeniz önemlidir. Bu güç sayesinde C++ programlama yeteneğinizin sınırlarını geliştirir. Ancak eğer bu güç eksik kullanılırsa anlaşılması zor, takibi ve saklanması imkansız olan programlar da oluşturulabilirsiniz. C++ güçlü bir araçtır. Fakat her araç gibi iyiliği onu kullanan kişiye bağlıdır.

www.bergkun.com  
www.bergkun.com

www.bergkun.com  
www.bergkun.com

# C ve C++ Arasındaki Diğer Farklar

C++, ANSI standarı C'nin geliştirilmiş halidir ve sanal olarak tüm C programları aynı zamanda C++ programlarıdır. Fakat, aralarında bazı küçük farklılıklar da mevcuttur. Bunlardan birkaçını Bölüm 1'de incelemiştik. Burada da bilmeniz gereken birkaç tanesinden de bahsedeceğiz.

C++, ANSI standarı C'nin geliştirilmiş halidir ve sanal olarak tüm C programları aynı zamanda C++ programlarındır. Fakat, aralarında bazı küçük farklılıklar da mevcuttur. Bunlardan birkaçını Bölüm 1'de incelemiştik. Burada da bilmeniz gereken birkaç tanesinden de bahsedeceğiz:

- C ve C++ arasında küçük, fakat potansiyel olarak önemli bir fark şudur : C'de bir karakter sabit otomatik olarak bir tamsayıya yükseltilir, C++'da ise bu şekilde olmaz.
- C'de, global bir değişkenin birkaç defa deklare edilmesi, her ne kadar kötü bir programlama uygulaması olsa da, hata değildir. C++'da bu bir hatadır.
- C'de bir belirticinin en azından 32 önemli karakteri olacaktır. C++'da tüm karakterler önemli kabul edilir. Fakat, olaya pratik açıdan bakacak olursak aşırı uzun belirticiler hantaldırlar ve seyrek kullanılırlar.
- C'de pek normal bir şey olmasa da **main()**'i bir program içerisinde çağırabilirsiniz. C++'da buna izin verilmez.
- C'de **register** değişkeninin adresini alamazsınız. C++'da bunu yapabilirsiniz.
- C'de **wchar\_t** tipi **typedef** ile tanımlanmıştır. C++'da **wchar\_t** bir anahtar kelime dir.

E K B

# Cevaplar

## 1.3. Alıştırmalar

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

int main()
{
 double hours, wage;

 cout << "Çalışılan saatleri girin: ";
 cin >> hours;

 cout << "Saat başı ücreti girin: ";
 cin >> wage;

 cout << "Ödenecek: $" << wage * hours;

 return 0;
}

2. #include <iostream>
using namespace std;

int main()
{
 double feet;

 do {
 cout << "Fiti girin (çıkış için 0): ";
 cin >> feet;

 cout << feet * 12 << " inç\n";
 } while (feet != 0.0);

 return 0;
}

3. /* Bu program en küçük ortak paydayı hesaplar. */
#include <iostream>
using namespace std;

int main()
{
 int a, b, d, min;

 cout << "İki sayı girin: ";
 cin >> a >> b;
 min = a > b ? b : a;

 for(d = 2; d<min; d++)
 if(((a%d)==0) && ((b%d)==0)) break;
 if(d==min) {
 cout << "Ortak payda yok\n";
 return 0;
 }
 cout << "En küçük ortak payda " << d << ".\n";
 return 0;
}
```

## 1.4. Alıştırmalar

- Açıklama her ne kadar garip olsa da geçerlidir.

## 1.5. Alıştırmalar

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

class card {
    char title[80]; // kitap adı
    char author[40]; // yazar
    int number; // kütüphane numarası
public:
    void store(char *t, char *name, int num);
    void show();
};

void card::store(char *t, char *name, int num)
{
    strcpy(title, t);
    strcpy(author, name);
    number = num;
}

void card::show()
{
    cout << "Kitap adı: " << title << "\n";
    cout << "Yazar: " << author << "\n";
    cout << "Eldeki sayı: " << number << "\n";
}

int main()
{
    card book1, book2, book3;

    book1.store("Dune", "Frank Herbert", 2);
    book2.store("The Foundation Trilogy", "Isaac Asimov", 2);
    book3.store("The Rainbow", "D. H. Lawrence", 1);

    book1.show();
    book2.show();
    book3.show();

    return 0;
}
```
- ```
#include <iostream>
using namespace std;

#define SIZE 100

class q_type {
 int queue[SIZE]; // kuyruğu tutuyor
 int head, tail; // head (kafa) ve tail(kuyruk) 'in indisleri
public:
 void init(); // hazırlık
```

```

void q(int num); // kayıt
int deq(); // tekrar alma
//

// Hazırlık
void q_type::init()
{
 head = tail = 0;
}

// Kuyruğa değer veriliyor.
void q_type::q(int num)
{
 if(tail+1==head || (tail+1==SIZE && !head)) {
 cout << "Kuyruk dolu\n";
 return;
 }
 tail++;
 if(tail==SIZE) tail = 0; // çevrim
 queue[tail] = num;
}

// Kuyruktan bir değer çıkarılıyor.
int q_type::deq()
{
 if(head == tail) {
 cout << "Kuyruk boş\n";
 return 0; // veya başka bir hata gösterici
 }
 head++;
 if(head==SIZE) head = 0; // çevrim
 return queue[head];
}

int main()
{
 q_type q1, q2;
 int i;

 q1.init();
 q2.init();
 for(i=1; i<=10; i++) {
 q1.q(i);
 q2.q(i*i);
 }
 for(i=1; i<=10; i++) {
 cout << "Dequeue 1: " << q1.deq() << "\n";
 cout << "Dequeue 2: " << q2.deq() << "\n";
 }
 return 0;
}

```

## 1.6. Alıştırmalar

1. f() fonksiyonunun prototipi yok.

## 1.7. Alistirmalar

1. 

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

// Tamsayılar, long ve double için sroot() fonksiyonunu aşırı yükleyin.

int sroot(int i);
long sroot(long i);
double sroot(double i);

int main()
{
 cout << "90.34'ün karekökü : " << sroot(90.34);
 cout << "\n";
 cout << "90L'nin karekökü : " << sroot(90L);
 cout << "\n";
 cout << "90'in karekökü : " << sroot(90);

 return 0;
}

// Tamsayıının karekökünü döndürür.
int sroot(int i)
{
 cout << "Tamsayıının kökü hesaplanıyor \n";
 return (int) sqrt((double) i);
}

// long'un karekökünü döndürür.
long sroot(long i)
{
 cout << " long'un kökü hesaplanıyor \n";
 return (long) sqrt((double) i);
}

// double'in karekökünü döndürür.
double sroot(double i)
{
 cout << " double'in kökü hesaplanıyor \n";
 return sqrt(i);
}
```

2. atof( ), atoi( ) ve atol( ) fonksiyonları aşırı yüklenemezler, çünkü sadece döndürdükleri veri tipleri farklıdır. Fonksiyonların aşırı yüklenmesi için argümanlarının sayısının ya da tipinin farklı olması gereklidir.

3. // min() fonksiyonunun aşırı yüklenmesi.

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

char min(char a, char b);
int min(int a, int b);
double min(double a, double b);

int main()
{
```

```
cout << "Min : " << min('x', 'a') << "\n";
cout << "Min : " << min(10, 20) << "\n";
cout << "Min : " << min(0.2234, 99.2) << "\n";
return 0;
}

// min() for chars
char min(char a, char b)
{
 return tolower(a)<tolower(b) ? a : b;
}

// min() for ints
int min(int a, int b)
{
 return a<b ? a : b;
}

// double'lar için min()
double min(double a, double b)
{
 return a<b ? a : b;
}

4. #include <iostream>
using namespace std;

// sleep'in tamsayı ve char * argümanını kabul edecek şekilde aşırı yüklenmesi
void sleep(int n);
void sleep(char *n);

// bu değeri işlemcinizin hızına uyacak şekilde değiştirin.
#define DELAY 100000

int main()
{
 cout << '.';
 sleep(3);
 cout << '.';
 sleep("2");
 cout << '.';
 return 0;
}

// Tamsayı argümanı için sleep().
void sleep(int n)
{
 long i;
 for(; n; n--)
 for(i=0; i<DELAY; i++)
}

// char * argümanı için sleep().
void sleep(char *n)
{
 long i;
 int j;
 j = atoi(n);
 for(; j; j--)
 for(i=0; i<DELAY; i++)
}
```

## Pekiştirme Testi: Bölüm 1

1. Polimorfizm çeşitli ortamlara erişebilmek için genel bir arabirimin kullanıldığı mekanizmadır. Depolama, kod ile ona ait verisi arasında korunan bir bağ sağlar. Depolanan rutinlere erişim kontrolü sıkı bir şekilde kontrol edilebilir, böyledikle istenmeyen kurcalamaları önler. Kalıtım, bir nesnenin diğer nesnelerin özelliklerini alabildiği bir süreçtir. Kalıtım ile hiyerarşik sınıflandırma sisteminin desteklenmesi için kullanılır.
2. Gerek normal C stili ile gerekse C++'a özel tek satır açıklamaları ile bir C++ programına açıklama ekleyebilirsiniz.
3. 

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

int main()
{
 int b, e, r;
 cout << "Tabanı girin: ";
 cin >> b;
 cout << "Üssünü girin: ";
 cin >> e;

 r = 1;
 for(; e; e--) r = r * b;

 cout << "Sonuç: " << r;

 return 0;
}
```
4. 

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

// katar çevirme fonksiyonunun aşırı yüklenmesi.
void rev_str(char *s); // katar ters çevriliyor
void rev_str(char *in, char *out); // çıkışa ters katar veriliyor

int main()
{
 char s1[80], s2[80];

 strcpy(s1, "Bu bir deneme");

 rev_str(s1, s2);
 cout << s2 << "\n";

 rev_str(s1);
 cout << s1 << "\n";

 return 0;
}

// Katar ters çevriliyor ve sonucu s'e koyn.
void rev_str(char *s)
{
```

```

 char temp[80];
 int i, j;

 for(i=strlen(s)-1, j=0; i>=0; i--, j++)
 temp[j] = s[i];

 temp[j] = '\0'; // null-terminate sonucu
 strcpy(s, temp);
}

// Katarı ters çevirin ve sonucu çıkışa verin.
void rev_str(char *in, char *out)
{
 int i, j;

 for(i=strlen(in)-1, j=0; i>=0; i--, j++)
 out[j] = in[i];

 out[j] = '\0'; // null-terminate sonucu
}

5. #include <iostream.h>

int f(int a);

int main()
{
 cout << f(10);

 return 0;
}

int f(int a)
{
 return a * 3.1416;
}

```

6. **bool** veri tipi Boolean değerleri saklar. **bool** tipinde nesnelerin sahip olabileceği değerler sadece **true** ve **false**'dur.

## Gözden Geçirme Testi: Bölüm 2

```

1. #include <iostream>
#include <cstring>
using namespace std;

int main()
{
 char s[80];

 cout << "Katar girin: ";
 cin >> s;

 cout << "Uzunluk: " << strlen(s) << "\n";

 return 0;
}

```

```
2. #include <iostream>
#include <cstring>
using namespace std;

class addr {
 char name[40];
 char street[40];
 char city[30];
 char state[3];
 char zip[10];
public:
 void store(char *n, char *s, char *c, char *t, char *z);
 void display();
};

void addr::store(char *n, char *s, char *c, char *t, char *z)
{
 strcpy(name, n);
 strcpy(street, s);
 strcpy(city, c);
 strcpy(state, t);
 strcpy(zip, z);
}

void addr::display()
{
 cout << name << "\n";
 cout << street << "\n";
 cout << city << "\n";
 cout << state << "\n";
 cout << zip << "\n\n";
}

int main()
{
 addr a;

 a.store("C. B. Turkle", "11 Pinetree Lane", "Mausau", "In", "46576");
 a.display();

 return 0;
}

3. #include <iostream>
using namespace std;

int rotate(int i);
long rotate(long i);
int main()
{
 int a;
 long b;

 a = 0x8000;
 b = 8;

 cout << rotate(a);
 cout << "\n";
 cout << rotate(b);
```

```
 return 0;
}

int rotate(int i)
{
 int x;

 if(i & 0x8000) x = 1;
 else x = 0;

 i = i << 1;
 i += x;

 return i;
}

long rotate(long i)
{
 int x;

 if(i & 0x80000000) x = 1;
 else x = 0;
 i = i << 1;
 i += x;

 return i;
}
```

4. *i* tamsayısı *myclass*'a private'dır ve *main( )*'in içerisinde ona erişilemez.

## 2.1. Alıştırmalar

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

#define SIZE 100

class q_type {
 int queue[SIZE]; // kuyruğu saklar.
 int head, tail; // head(kafa) ve tail(kuyruk) indisleri
public:
 q_type(); // constructor
 void q(int num); // saklanıyor
 int deq(); // tekrar alma
};

// Constructor
q_type::q_type()
{
 head = tail = 0;
}

// Kuyruğa değer konuluyor.
void q_type::q(int num)
{
 if(tail+1==head || (tail+1==SIZE && !head)) {
 cout << "Kuyruk dolu\n";
 return;
 }
```

```

 tail++;
 if(tail==SIZE) tail = 0; // çevrim
 queue[tail] = num;
 }

 // Kuyruktan değer çıkarılıyor.
 int q_type::deq()
 {
 if(head == tail) {
 cout << "Kuyruk boş \n";
 return 0; // veya başka hata gösterici
 }
 head++;
 if(head==SIZE) head = 0; // çevrim
 return queue[head];
 }

 int main()
 {
 q_type q1, q2;
 int i;

 for(i=1; i<=10; i++) {
 q1.q(i);
 q2.q(i*i);
 }

 for(i=1; i<=10; i++) {
 cout << "Dequeue 1: " << q1.deq() << "\n";
 cout << "Dequeue 2: " << q2.deq() << "\n";
 }
 return 0;
 }
}

2. // Kronometre emulatoru
#include <iostream>
#include <ctime>
using namespace std;

class stopwatch {
 double begin, end;
public:
 stopwatch();
 ~stopwatch();
 void start();
 void stop();
 void show();
};

stopwatch::stopwatch()
{
 begin = end = 0.0;
}
stopwatch::~stopwatch()
{
 cout << "Stopwatch nesnesi yok ediliyor...";
 show();
}
void stopwatch::start()
{
 begin = (double) clock() / CLOCKS_PER_SEC;
}

```

```

void stopwatch::stop()
{
 end = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::show()
{
 cout << "Geçen zaman: " << end - begin;
 cout << "\n";
}

int main()
{
 stopwatch watch;
 long i;

 watch.start();
 for(i=0; i<320000; i++) ; // for döngüsünü kur
 watch.stop();

 watch.show();

 return 0;
}

```

3. constructor'ların döndürme tipi olamaz.

## 2.2. Alıştırmalar

1. // Yiğin için bellekte dinamik olarak yer ayrılıyor.

```

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

// Karakterler için bir yiğin sınıfı bildiriliyor,
class stack {
 char *stck; // yiğini tutuyor
 int tos; // yiğinin indisı
 int size; // yiğinin büyüklüğü
public:
 stack(int s); // constructor
 ~stack(); // destructor
 void push(char ch); // yiğine karakter itiliyor
 char pop(); // yiğinden karakter çekiliyor
};

// Yiğini hazırlama
stack::stack(int s)
{
 cout << "Bir yiğin oluşturuluyor\n";
 tos = 0;
 stck = (char *) malloc(s);
 if(!stck) {
 cout << "Bellekte yer ayırma hatası\n";
 exit(1);
 }
 size = s;
}

```

```

stack::~stack()
{
 free(stck);
}

// Bir karakter iter.
void stack::push(char ch)
{
 if(tos==size) {
 cout << "Yığın dolu\n";
 return;
 }
 stck[tos] = ch;
 tos++;
}

// Bir karakter iter.
char stack::pop()
{
 if(tos==0) {
 cout << "Yığın boş\n";
 return 0; // yığın boşsa null döndürülüyor.
 }
 tos--;
 return stck[tos];
}

int main()
{
 // Hazırlığı otomatik olarak yapılan iki yığın oluşturun.
 stack s1(10), s2(10);
 int i;

 s1.push('a');
 s2.push('x');
 s1.push('b');
 s2.push('y');
 s1.push('c');
 s2.push('z');

 for(i=0; i<3; i++) cout << "s1'i çek: " << s1.pop() << "\n";
 for(i=0; i<3; i++) cout << "s2'yi çek: " << s2.pop() << "\n";

 return 0;
}

2. #include <iostream>
#include <ctime>
using namespace std;

class t_and_d {
 time_t systime;
public:
 t_and_d(time_t t); // constructor
 void show();
};

t_and_d::t_and_d(time_t t)
{
 systime = t;
}

```

```

void t_and_d::show()
{
 cout << ctime(&sys time);
}

int main()
{
 time_t x;

 x = time(NULL);
 t_and_d ob(x);
 ob.show();

 return 0;
}

3. #include <iostream>
using namespace std;

class box {
 double l, w, h;
 double volume;
public:
 box(double a, double b, double c);
 void vol();
};

box::box(double a, double b, double c)
{
 l = a;
 w = b;
 h = c;
 volume = l * w * h;
}
void box::vol()
{
 cout << "Hacim : " << volume << "\n";
}

int main()
{
 box x(2.2, 3.97, 8.09), y(1.0, 2.0, 3.0);

 x.vol();
 y.vol();
 return 0;
}

```

## 2.3. Alıştırma

- #include <iostream>
using namespace std;

class area\_cl {
public:
 double height;
 double width;
};

```

class rectangle : public area_cl {
public:
 rectangle(double h, double w);
 double area();
};

class isosceles : public area_cl {
public:
 isosceles(double h, double w);
 double area();
};

rectangle::rectangle(double h, double w)
{
 height = h;
 width = w;
}

isosceles::isosceles(double h, double w)
{
 height = h;
 width = w;
}

double rectangle::area()
{
 return width * height;
}

double isosceles::area()
{
 return 0.5 * width * height;
}

int main()
{
 rectangle b(10.0, 5.0);
 isosceles i(4.0, 6.0);
 cout << "Dikdörtgen: " << b.area() << "\n";
 cout << "Üçgen: " << i.area() << "\n";

 return 0;
}

```

## 2.5. Alıştırmalar

1. // Yapı kulanınan stack sınıfı.  

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

#define SIZE 10

// Yapı kulanarak karakterler için stack sınıfı deklare edin.
struct stack {
 stack(); // constructor
 void push(char ch); // yiğine karakter itin
 char pop(); // yiğinden karakter çekin.
private:
 char stck[SIZE]; // yiğin tutuluyor
```

```

 int tos; // yiğinin indisi
}

// Yiğinin hazırlığı yapılıyor.
stack::stack()
{
 cout << "Yiğin oluşturuluyor\n";
 tos = 0;
}

// Karakter itin.
void stack::push(char ch)
{
 if(tos==SIZE) {
 cout << "Yiğin dolu\n";
 return;
 }
 stck[tos] = ch;
 tos++;
}

// Karakter çekin.
char stack::pop()
{
 if(tos==0) {
 cout << "Yiğin boş\n";
 return 0; // Yiğin boşsa null döndürün
 }
 tos--;
 return stck[tos];
}

int main()
{
 // Hazırlığı otomatik olarak yapılan iki yiğini oluşturun.
 stack s1, s2;
 int i;

 s1.push('a');
 s2.push('x');
 s1.push('b');
 s2.push('y');
 s1.push('c');
 s2.push('z');

 for(i=0; i<3; i++) cout << "s1'i çekin: " << s1.pop() << "\n";
 for(i=0; i<3; i++) cout << "s2'yi çekin: " << s2.pop() << "\n";

 return 0;
}

2. #include <iostream>
using namespace std;

union swapbytes {
 unsigned char c[2];
 unsigned i;
 swapbytes(unsigned x);
 void swp();
};


```

```

swapbytes::swapbytes(unsigned x)
{
 i = x;
}
void swapbytes::swp()
{
 unsigned char temp;
 temp = c[0];
 c[0] = c[1];
 c[1] = temp;
}
int main()
{
 swapbytes ob(1);
 ob.swp();
 cout << ob.i;
 return 0;
}

```

3. Anonim bileşim, iki değişkenin aynı bellek yerini paylaşmasını sağlayan mekanizmadır. Anonim birleşimin üyelerine doğrudan erişilir, nesneye bir referans gerekmmez. Birleşimlerle aynı alan seviyesindedirler.

## 2.6. Alıştırmalar

- #include <iostream>  
 using namespace std;  
  
 // abs() üç şekilde aşırı yükleniyor :  
  
 // abs() for ints  
 inline int abs(int n)  
 {  
 cout << "Tamsayı abs()\n";
 return n<0 ? -n : n;
 }
  
 // abs() for longs  
 inline long abs(long n)  
 {  
 cout << "long abs()\n";
 return n<0 ? -n : n;
 }
  
 // abs() for doubles  
 inline double abs(double n)  
 {  
 cout << "double abs()\n";
 return n<0 ? -n : n;
 }
  
 int main()
 {
 cout << "-10'un mutlak değeri: " << abs(-10) << "\n";
 cout << "-10L'nın mutlak değeri: " << abs(-10L) << "\n";
 cout << "-10.01'un mutlak değeri: " << abs(-10.01) << "\n";

 return 0;
 }

2. Fonksiyon in-line yapılamayabilir, çünkü bir `for` döngüsü içermektedir. Bazı derleyiciler döngü içeren fonksiyonları in-line yapamazlar.

## 2.7. Alıştırmalar

1. 

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

#define SIZE 10

// Karakterler için bir stack sınıfı deklare edin.
class stack {
 char stck[SIZE]; // yiğini tutuluyor
 int tos; // yiğinin indisi
public:
 stack() { tos = 0; }
 void push(char ch)
 {
 if(tos==SIZE) {
 cout << "yiğın dolu\n";
 return;
 }
 stck[tos] = ch;
 tos++;
 }
 char pop()
 {
 if(tos==0) {
 cout << "yiğın boş\n";
 return 0; // yiğin boşsa null döndürün
 }
 tos--;
 return stck[tos];
 }
};

int main()
{
 // Hazırlığı otomatik olarak yapılan iki yiğin oluşturun.
 stack s1, s2;
 int i;

 s1.push('a');
 s2.push('x');
 s1.push('b');
 s2.push('y');
 s1.push('c');
 s2.push('z');

 for(i=0; i<3; i++) cout << "s1'i çekin: " << s1.pop() << "\n";
 for(i=0; i<3; i++) cout << "s2'yi çekin: " << s2.pop() << "\n";

 return 0;
}
```
2. 

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

```

class strtype {
 char *p;
 int len;
public:
 strtype(char *ptr)
 {
 len = strlen(ptr);
 p = (char *) malloc(len+1);
 if(!p) {
 cout << "Belirkte yer ayirma hatasi\n";
 exit(1);
 }
 strcpy(p, ptr);
 }
 ~strtype() { cout << "p serbest bırakiliyor\n"; free(p); }

 void show()
 {
 cout << p << " - uzunluk: " << len;
 cout << "\n";
 }
};

int main()
{
 strtype s1("Bu bir deneme."), s2("Ben C++'i seviyorum.");
 s1.show();
 s2.show();

 return 0;
}

```

## Pekiştirme Testi: Bölüm 2

1. Constructor, nesne oluşturulurken çağrılan fonksiyondur. destructor ise nesne yok edilirken çağrılan fonksiyondur.
2. #include <iostream>  
using namespace std;

```

class line {
 int len;
public:
 line(int l);
};
line::line(int l)
{
 len = l;
 int i;
 for(i=0; i<len; i++) cout << "*";
}

int main()
{
 line l(10);
 return 0;
}

```

```

3. 10 1000000 -0.0009

4. #include <iostream>
using namespace std;

class area_cl {
public:
 double height;
 double width;
};

class rectangle : public area_cl {
public:
 rectangle(double h, double w) { height = h; width = w; }
 double area() { return height * width; }
};

class isosceles : public area_cl {
public:
 isosceles(double h, double w) { height = h; width = w; }
 double area() { return 0.5 * width * height; };
};

class cylinder : public area_cl {
public:
 cylinder(double h, double w) { height = h; width = w; }
 double area()
 {
 return (2 * 3.1416 * (width/2) * (width/2)) + (3.1416 * width * height);
 }
};
int main()
{
 rectangle b(10.0, 5.0);
 isosceles i(4.0, 6.0);
 cylinder c(3.0, 4.0);

 cout << "Dikdörtgen: " << b.area() << "\n";
 cout << "Üçgen: " << i.area() << "\n";
 cout << "Silindir: " << c.area() << "\n";

 return 0;
}

```

5. in-line fonksiyonlarının kodu satır içinde genişletilir. Bu, fonksiyon gerçekten çağrılmaması anlamına geliyor. Bu şekilde fonksiyonun çağrılması ve döndürme mekanizması ile ilgili sorunlar önlenmiş olur. Avantajı çalışma hızını artırmasıdır. Dezavantajı ise programın büyülüüğünü artırmamasıdır.

```

6. #include <iostream>
using namespace std;

class myclass {
 int i, j;
public:
 myclass(int x, int y) { i = x; j = y; }
 void show() { cout << i << " " << j; }
};

int main()

```

```
 myclass count(2, 3);
 count.show();
 return 0;
}
```

7. Bir sınıf içerisinde, üyeleri varsayılan olarak private'dır. Yapılar içerisinde ise, üyeleri varsayılan olarak public'dır.
8. Evet. Bir anonim birleşim tanımlar.

## Bütünleştirme Testi: Bölüm 2

1. 

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

class prompt {
 int count;
public:
 prompt(char *s) { cout << s; cin >> count; }
 ~prompt();
};

prompt::~prompt() {
 int i, j;

 for(i=0; i<count; i++) {
 cout << '\a';
 for(j=0; j<32000; j++); // gecikme
 }
}

int main()
{
 prompt ob("Bir sayı girin: ");
 return 0;
}
```
2. 

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

class ftoi {
 double feet;
 double inches;
public:
 ftoi(double f);
};
ftoi::ftoi(double f)
{
 feet = f;
 inches = feet * 12;
 cout << feet << inches << "inc.\n";
}
int main()
{
 ftoi a(12.0), b(99.0);
 return 0;
}
```

```

3. #include <iostream>
#include <cstdlib>
using namespace std;

class dice {
 int val;
public:
 void roll();
};

void dice::roll()
{
 val = (rand() % 6) + 1; // 1 ile 6 arasında sayı üretin
 cout << val << "\n";
}

int main()
{
 dice one, two;

 one.roll();
 two.roll();
 one.roll();
 two.roll();
 one.roll();
 two.roll();
 return 0;
}

```

## Gözden Geçirme Testi: Bölüm 3

- constructor **widgit()**, destructor ise **~widgit()** şeklinde adlandırılıyor.
- constructor fonksiyonu bir nesne oluşturulurken çağrılır (yani, nesne var olurken). destructor ise nesne yok edilirken çağrılır.
- ```
class Mars : public planet {
// ...
};
```
- Bir fonksiyonu satır içinde genişletmek için ya tanımının önüne **inline** belirticisi koyulur ya da tanımı bir sınıf bildirimini içerisinde koyulur.
- in-line fonksiyonlar ilk kullanımından önce tanımlanmalıdır. Diğer bilinen kısıtlamalar şunlardır: Herhangi bir döngü içeremez. İçe dönük olamaz. **goto** veya **switch** ifadelerini içeremez ve herhangi bir **static** değişken içeremez.
- ```
sample ob(100, "X") ;
```

### 3.1. Alıştırmalar

- x=y** atama ifadesi yanlıştır çünkü **cl1** ve **cl2** iki ayrı tip sınıfı. Hal böyle olunca iki farklı sınıfa ait nesneler birbirine atanamaz.
- ```
#include <iostream>
```

```

using namespace std;

#define SIZE 100

class q_type {
    int queue[SIZE]; // kuyruğu tutar
    int head, tail; // head ve tail'in indisleri
public:
    q_type(); // yapılandırıcı
    void q(int num); // saklama
    int deq(); // geri alma
};

// yapılandırıcı
q_type::q_type()
{
    head = tail = 0;
}
// Değeri kuyruğa koy
void q_type::q(int num)
{
    if(tail+1==head || (tail+1==SIZE && !head)) {
        cout << "Queue is full\n";
        return;
    }
    tail++;
    if(tail==SIZE) tail = 0; // çevrim
    queue[tail] = num;
}
// Değeri sıradan çıkar
int q_type::deq()
{
    if(head == tail) {
        cout << "Queue is empty\n";
        return 0; // veya diğer bir hata belirteci
    }
    head++;
    if(head==SIZE) head = 0; // çevrim
    return queue[head];
}

int main()
{
    q_type q1, q2;
    int i;

    for(i=1; i<=10; i++) {
        q1.q(i);
    }

    //bir kuyruğu diğerine ata
    q2 = q1;

    //her ikisinin de aynı değere sahip olduğunu göster
    for(i=1; i<=10; i++)
        cout << "Dequeue 1: " << q1.deq() << "\n";

    for(i=1; i<=10; i++)
        cout << "Dequeue 2: " << q2.deq() << "\n";
    return 0;
}

```

3. Eğer kuyruğu saklamak için gerekli olan bellek dinamik olarak ayrılsa, bir kuyruğu diğerine atmak, atama ifadesinde, sol taraftaki kuyruk için ayrılmış olan dinamik belleğin yok olmasını sebep olur. Ayrıca sağ taraftaki kuyruk için ayrılmış olan bellek nesneler yok edildiğinde iki defa serbest bırakılır. Bu iki durumdan biri kabul edilemeyen bir hatadır.

3.2. Alıştırmalar

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

#define SIZE 10

// karakterler için bir stack sınıfı bildir
class stack {
    char stck[SIZE]; // yiğini tutar
    int tos; // stack Üstünün indeksi
public:
    stack(); // yapılandırıcı
    void push(char ch); // yiğine karakter sok
    char pop(); // yiğinden karakter çek
};

// stack'i hazırla
stack::stack()
{
    cout << "Constructing a stack\n";
    tos = 0;
}

// karakter sok
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Stack is full\n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// karakter çek
char stack::pop()
{
    if(tos==0) {
        cout << "Stack is empty\n";
        return 0; // yiğin boşsa boş döndür
    }
    tos--;
    return stck[tos];
}

void showstack(stack o);
int main()
{
    stack s1;
    int i;
```

```

    sl.push('a');
    sl.push('b');
    sl.push('c');

    showstack(sl);

    // main içerisindeki sl halen mevcut
    cout << "sl stack still contains this: \n";
    for(i=0; i<3; i++) cout << sl.pop() << "\n";

    return 0;
}

//yiğindaki içeriği göster
void showstack(stack o)
{
    char c;

    //Bu ifade sona erdiğinde o yiğini boşaltır
    while(c=o.pop()) cout << c << "\n";
    cout << "\n";
}

```

Bu ifade aşağıdaki çıkışını verir.

```

Constructing a stack
c
b
a
Stack is empty
sl stack still contains this:
c
b
a

```

2. `neg()`'ı çağırmak için kullanılan `o` nesnesindeki `p` işaretçisi tarafından gösterilen tamsayı için kullanılan bellek, `o`'nun kopyası yok edildiğinde serbest bırakılır. Bu olay `neg()` sona erdiğinde, `main()` içerisindeki `o` tarafından ihtiyaç duyulsa bile gerçekleşir.

3.3. Alıştırmalar

```

1. #include <iostream>
using namespace std;

class who {
    char name;
public:
    who(char c) {
        name = c;
        cout << "Constructing who #";
        cout << name << "\n";
    }
    ~who() { cout << "Destructing who #" << name << "\n"; }
};

who makewho();

```

```

    who_temp('B');
    return temp;
}

int main()
{
    who ob('A');
    makewho();
    return 0;
}

```

2. Bir nesne döndürülmesinin uygun olmayacağı çeşitli durumlar mevcuttur. Örneğin; bir nesne yaratılırken diskte bir dosyayı açıyor ve yok-edilirken de kapatıyorsa, o nesne bir fonksiyondan döndürülürken, geçici nesne yok-edildiğinde dosya kapatılacaktır.

3.4. Alıştırma

```

1. #include <iostream>
using namespace std;

class pr2; // forward bildirim

class pri {
    int printing;
    // ...
public:
    pri() { printing = 0; }
    void set_print(int status) { printing = status; }
    // ...
    friend int inuse(pr1 o1, pr2 o2);
};

class pr2 {
    int printing;
    // ...
public:
    pr2() { printing = 0; }
    void set_print(int status) { printing = status; }
    // ...
    friend int inuse(pr1 o1, pr2 o2);
};

// Yazıcı kullanımdaysa true döndür
int inuse(pr1 o1, pr2 o2)
{
    if(o1.printing || o2.printing) return 1;
    else return 0;
}

int main()
{
    pri p1;
    pr2 p2;

    if(!inuse(p1, p2)) cout << "printer idle\n";
}

```

```

cout << "Setting p1 to printing...\n";
p1.set_print(1);
if(inuse(p1, p2)) cout << "Now, printer in use.\n";

cout << "Turn off p1...\n";
p1.set_print(0);
if(!inuse(p1, p2)) cout << "Printer idle\n";

cout << "Turn on p2...\n";
p2.set_print(1);
if(inuse(p1, p2)) cout << "Now, printer in use.\n";

return 0;
}

```

Pekiştirme Testi: Bölüm 3

1. Bir nesne diğerine atanacaksa iki nesnenin sınıfı tipi de aynı olmalıdır.
2. **ob1** ve **ob2** atamasındaki problem şudur: **ob2**'ye ait **p**'nin ilk değeri tarafından işaret edilen bellek şimdi yok-olmuştur. Çünkü bu değer atama ile değiştirilmiştir. Böylece bu belleğin serbest bırakılması imkansız hale gelmiştir ve **ob1**'e ait **p** tarafından işaret edilen bellek, yok edilme esnasında iki defa serbest bırakılmıştır. (Büyük ihtimalle dinamik yer ayırma sisteme zarar verecek)

3.

```
int light(planet p)
{
    return p.get_miles() / 186000;
}
```
4. Evet.
5.

```
// yiğini alfabe ile doldur
#include <iostream>
using namespace std;

#define SIZE 27

// Karakterler için bir stack sınıfı bildir.
class stack {
    char stck[SIZE]; // stack'i tut
    int tos; // stack üstünün indeks'i
public:
    stack(); // yapılandırıcı
    void push(char ch); // yiğine karakter sok
    char pop(); // yiğinden karakter çek
};

// stack'i hatırla
stack::stack()
{
    cout << "Constructing a stack\n";
    tos = 0;
}

// karakter sok
void stack::push(char ch)
{
```

```

    if(tos==SIZE) {
        cout << "Stack is full\n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Karakter çek
char stack::pop()
{
    if(tos==0) {
        cout << "Stack is empty\n";
        return 0; //bos yığında boş döndür
    }
    tos--;
    return stck[tos];
}

void showstack(stack o);
stack loadstack();

int main()
{
    stack s1;

    s1 = loadstack();
    showstack(s1);

    return 0;
}

// Stack içeriğini göster
void showstack(stack o)
{
    char c;

    // Bu komut bittiğinde stack artık boştur
    while(c=o.pop()) cout << c << "\n";
    cout << "\n";
}

//stack'i alfabetin harfleriyle yükle
stack loadstack()
{
    stack t;
    char c;
    for(c = 'a'; c <= 'z'; c++) t.push(c);
    return t;
}

```

6. Bir fonksiyona nesne geçirildiğinde veya ondan nesne döndürüldüğünde, fonksiyon sona erdiğinde yok edilecek, o nesnenin geçici kopyaları oluşturulur. Nesnenin geçici kopyası yok edildiğinde yok-edici fonksiyon programda gerekli olabilecek herhangi bir şeyi yok-edilebilir.
7. Arkadaş, üye olmadığı halde private üyelerle erişim iznine sahip bir fonksiyondur.

Bütünleştirme Testi: Bölüm 3

```
1. // Yiğine alfabe yükleniyor.  
#include <iostream>  
#include <cctype>  
using namespace std;  
  
#define SIZE 27  
  
// Karakterler için bir stack sınıfı deklare ediliyor.  
class stack {  
    char stck[SIZE]; // yiğini tutar  
    int tos; // yiğinin indisini  
public:  
    stack(); // constructor  
    void push(char ch); // yiğine karakter iter  
    char pop(); // yiğindan karakter çeker  
};  
  
// Yiğinin hazırlığını yapar.  
stack::stack()  
{  
    cout << "Yiğin oluşturuluyor\n";  
    tos = 0;  
}  
  
// Karakter çeker.  
void stack::push(char ch)  
{  
    if(tos==SIZE) {  
        cout << "Yiğin dolu\n";  
        return;  
    }  
    stck[tos] = ch;  
    tos++;  
}  
  
// Bir karakter çek.  
char stack::pop()  
{  
    if(tos==0) {  
        cout << "Yiğin boş\n";  
        return 0; // Yiğin boşsa null dondurur  
    }  
    tos--;  
    return stck[tos];  
}  
  
void showstack(stack o);  
stack loadstack();  
stack loadstack(int upper);  
  
int main()  
{  
    stack s1, s2, s3;  
  
    s1 = loadstack();  
    showstack(s1);  
  
    // Büyük harfleri al
```

```
s2 = loadstack(l);
showstack(s2);

// Küçük harfleri kullan
s3 = loadstack(0);
showstack(s3);

return 0;
}

// Yiğinin içeriğini göster.
void showstack(stack o)
{
    char c;

    // Bu ifade sona erdiğinde o yiğini boştur.
    while(c=o.pop()) cout << c << "\n";
    cout << "\n";
}

// Yiğini alfabetin harfleri ile doldur.
stack loadstack()
{
    stack t;
    char c;

    for(c = 'a'; c<='z'; c++) t.push(c);
    return t;
}

/* Yiğini alfabetin harfleri ile doldur.
   Eğer upper 1 ise harfleri büyüt; aksi halde küçült. */
stack loadstack(int upper)
{
    stack t;
    char c;

    if(upper) c = 'A';
    else c = 'a';
    for(; toupper(c)<='Z'; c++) t.push(c);
    return t;
}

2. #include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype();
    void show();
    friend char *get_string(strtype *ob);
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr);
}
```

```

p = (char *) malloc(len+1);
if(!p) {
    cout << "Bellekte yer ayırma hatası\n";
    exit(1);
}
strcpy(p, ptr);

strtype::~strtype()
{
    cout << "p serbest bırakılıyor\n";
    free(p);
}

void strtype::show()
{
    cout << p << " - length: " << len;
    cout << "\n";
}

char *get_string(strtype *ob)
{
    return ob->p;
}

int main()
{
    strtype s1("Bu bir deneme.");
    char *s;

    s1.show();
    // katara işaretçi al
    s = get_string(&s1);
    cout << "Bu s1'deki katar: ";
    cout << s << "\n";

    return 0;
}

```

3. Bu denemenin sonucu şu şekildedir : Evet, taban sınıfından gelen veri, türemiş sınıf'a ait bir nesneyi yine aynı sınıfın türemiş diğer bir nesneye atarken kopyalanır. Bu durumu aşağıdaki program göstermektedir.

```

#include <iostream>
using namespace std;

class base {
    int a;
public:
    void load_a(int n) { a = n; }
    int get_a() { return a; }
};

class derived : public base {
    int b;
public:
    void load_b(int n) { b = n; }
    int get_b() { return b; }
};

```

```

int main()
{
    derived ob1, ob2;

    ob1.load_a(5);
    ob1.load_b(10);

    // ob1'i ob2'ye ata
    ob2 = ob1;

    cout << "ob1'in a'sı ve b'sı: ";
    cout << ob1.get_a() << ' ' << ob1.get_b() << "\n";
    cout << " ob2'nin a'sı ve b'sı: ";
    cout << ob2.get_a() << ' ' << ob2.get_b() << "\n";

    /* Sizin de tahmin edeceğiniz gibi, sonuç her nesne için aynıdır. */

    return 0;
}

```

Gözden Geçirme Testi: Bölüm 4

1. Bir nesne aynı tipten başka bir nesneye atandığında, sağıdaki nesneye ait tüm veri üyelerinin mevcut değerleri soldaki nesnede onlara karşılık gelen üyelere atanır.
2. Sorun, nesne atamaları sırasında hedef nesnede mevcut olan önemli verilerin üzerinde yazılmış durumunda ortaya çıkar. Örneğin, dinamik belleği veya açık bir dosyayı işaret eden bir işaretçinin üzerine yazılabilir, bu nedenle de bir kayıp meydana gelir.
3. Nesne fonksiyona gönderildiğinde bir kopyası oluşturulur. Fakat kopyanın constructor fonksiyonu çağrılmaz. Kopyanın destructor'u fonksiyonun sona ermesiyle nesne yok edilirken çağrıılır.
4. Parametreye geçirildiği zaman bir argüman ve kopyasının ayrılma ihlali çeşitli durumlar nedeniyle ortaya çıkabilir. Örneğin bir dinamik bellek bir destructor tarafından serbest bırakıldığında bu bellek aynı zamanda argüman içinde kaybolur. Genelde eğer destructor fonksiyonu ilk argümanın ihtiyaç duyduğu her şeyi yokederse argüman zarar görür.

```

5. #include <iostream>
using namespace std;

class summation {
    int num;
    long sum; // num'un toplamı
public:
    void set_sum(int n);
    void show_sum() {
        cout << num << " toplam " << sum << "\n";
    }
};

void summation::set_sum(int n)
{
    int i;

```

```

num = n;

sum = 0;
for(i=1; i<=n; i++)
    sum += i;
}

summation make_sum()
{
    int i;
    summation temp;

    cout << "Bir sayı girin: ";
    cin >> i;

    temp.set_sum(i);
    return temp;
}
int main()
{
    summation s;

    s = make_sum();

    s.show_sum();

    return 0;
}

```

6. Bazı derleyicilerde in-line fonksiyonu döngü içeremez.

```

7. #include <iostream>
using namespace std;

class myclass {
    int num;
public:
    myclass(int x) { num = x; }
    friend int isneg(myclass ob);
};

int isneg(myclass ob)
{
    return (ob.num < 0) ? 1 : 0;
}

int main()
{
    myclass a(-1), b(2);

    cout << isneg(a) << ' ' << isneg(b);
    cout << "\n";

    return 0;
}

```

8. Evet, bir arkadaş fonksiyon, birden fazla sınıfla arkadaş olabilir.

4.1. Alıştırmalar

1.

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

class letters {
    char ch;
public:
    letters(char c) { ch = c; }
    char get_ch() { return ch; }
};

int main()
{
    letters ob[10] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };
    int i;

    for(i=0; i<10; i++)
        cout << ob[i].get_ch() << ' ';

    cout << "\n";

    return 0;
}
```
2.

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

class squares {
    int num, sqr;
public:
    squares(int a, int b) { num = a; sqr = b; }
    void show() {cout << num << ' ' << sqr << "\n"; }
};

int main()
{
    squares ob[10] = {
        squares(1, 1),
        squares(2, 4),
        squares(3, 9),
        squares(4, 16),
        squares(5, 25),
        squares(6, 36),
        squares(7, 49),
        squares(8, 64),
        squares(9, 81),
        squares(10, 100)
    };
    int i;

    for(i=0; i<10; i++) ob[i].show();

    return 0;
}
```
3.

```
#include <iostream>
using namespace std;
class letters {
    char ch;
```

```

public:
    letters(char c) { ch = c; }
    char get_ch() { return ch; }
};

int main()
{
    letters ob[10] = {
        letters('a'),
        letters('b'),
        letters('c'),
        letters('d'),
        letters('e'),
        letters('f'),
        letters('g'),
        letters('h'),
        letters('i'),
        letters('j')
    };
    int i;

    for(i=0; i<10; i++)
        cout << ob[i].get_ch() <<

    cout << "\n";
    return 0;
}

```

4.2. Alıştırmalar

- // Ters sırada göster.

```

#include <iostream>
using namespace std;

class samp {
    int a, b;
public:
    samp(int n, int m) { a = n; b = m; }
    int get_a() { return a; }
    int get_b() { return b; }
};

int main()
{
    samp ob[4] = {
        samp(1, 2),
        samp(3, 4),
        samp(5, 6),
        samp(7, 8)
    };
    int i;

    samp *p;

    p = &ob[3]; // son elemanın adresini al

    for(i=0; i<4; i++) {
        cout << p->get_a() << ' ';
    }
}
```

```

        cout << p->get_b() << "\n";
        p--; // bir önceki nesneye geç
    }

    cout << "\n";

    return 0;
}

2. /* İki boyutlu nesneler matrisi oluştur. İşaretçi ile eriş. */
#include <iostream>
using namespace std;

class samp {
    int a;
public:
    samp(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob[4][2] = {1, 2, 3, 4, 5, 6, 7, 8};
    int i;

    samp *p;

    p = (samp *) ob;
    for(i=0; i<4; i++) {
        cout << p->get_a() << " ";
        p++;
        cout << p->get_a() << "\n";
        p++;
    }

    cout << "\n";

    return 0;
}

```

4.3. Alıştırma

```

1. // Bu işaretçiyi kullan.
#include <iostream>
using namespace std;

class myclass {
    int a, b;
public:
    myclass(int n, int m) { this->a = n; this->b = m; }
    int add() { return this->a + this->b; }
    void show();
};

void myclass::show()
{
    int t;
    t = this->add(); // Üye fonksiyonu çağır
    cout << t << "\n";
}

```

```
int main()
{
    myclass ob(10, 14);
    ob.show();
    return 0;
}
```

4.4. Alistirmalar

1.

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

int main()
{
    float *f;
    long *l;
    char *c;

    f = new float;
    l = new long;
    c = new char;

    if(!f || !l || !c) {
        cout << "Bellekte yer ayirma hatasi.";
        return 1;
    }

    *f = 10.102;
    *l = 100000;
    *c = 'A';

    cout << *f << ' ' << *l << ' ' << *c;
    cout << '\n';

    delete f; delete l; delete c;

    return 0;
}
```
2.

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

class phone {
    char name[40];
    char number[14];
public:
    void store(char *n, char *num);
    void show();
};

void phone::store(char *n, char *num)
{
    strcpy(name, n);
    strcpy(number, num);
}

void phone::show()
{
```

```

        cout << name << ":" << number;
        cout << "\n";
    }

    int main()
    {
        phone *p;

        p = new phone;

        if(!p) {
            cout << "Bellekte yer ayırmaya hatası.";
            return 1;
        }

        p->store("Isaac Newton", "111 555-2323");
        p->show();

        delete p;

        return 0;
    }
}

```

- Hata durumunda **new**, ya boş işaretçi döndürecek ya da bir hata üretecektir. Hangi yaklaşımın kullanıldığını tespit etmek için derleyicinizin dokümanlarına bakın. Standart C++'da, **new** normalde bir hata oluşturur.

4.5. Alıştırmalar

- ```

char *p;

p = new char [100];

// ...
strcpy(p, "Bu bir denemedir");

```
- ```

#include <iostream>
using namespace std;

int main()
{
    double *p;

    p = new double (-123.0987);

    cout << *p << '\n';

    return 0;
}

```

4.6. Alıştırmalar

- ```

#include <iostream>
using namespace std;

void zneg(int *i); // başvuru sürümlü
void pneg(int *i); // işaretçi sürümlü

```

```

int main()
{
 int i = 10;
 int j = 20;

 rneg(i);
 pneg(&j);

 cout << i << " " << j << "\n";

 return 0;
}

// referans parametresi kullanma
void rneg(int &i)
{
 i = -i;
}

// işaretçi parametresi kullanma
void pneg(int *i)
{
 *i = - *i;
}

```

2. `triple()` çağrılığında `d`'nin adresi `&` operatörü ile açık şekilde elde edilir. Bu ne gereklidir, ne de geçerlidir. Bir referans parametresi kullanıldığında argümanın ö-nüne `&` operatörü getirilmez.
3. Referans parametresinin adresi kendiliğinden fonksiyona geçirilir. Adresi manuel olarak elde etmenize gerek yoktur. Referans ile göndermek değer ile göndermekten daha hızlıdır. Argümanın kopyası oluşturulmaz. Böylece herhangi bir yan etki o-luştması ihtimali ortada kalkar. Çünkü kopyanın destructor'u çağrılmıştır.

## 4.7. Alıştırma

1. Orijinal programda `show()`'a nesne değer ile geçirilmiştir. Bu nedenle bir kopyası yapılmıştır. `show()` döndüğünde kopya yok edilir ve destructor'u çağrılar. Bu da `p`'nin bırakılmasına neden olur. Fakat, ona işaret edilen belleğe `show()`'un argümanları tarafından hala ihtiyaç duyulur. Size fonksiyon çağrılığında kopyanın ya-pılması engellemek için referans parametresi kullanan programın doğru sürü-münü aşağıda veriyorum:

```

// Bu program artık düzeltildmiştir.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
 char *p;
public:
 strtype(char *s);
 ~strtype() { delete [] p; }
}

```

```

 char *get() { return p; }

}

strtype::strtype(char *s)
{
 int l;

 l = strlen(s)+1;

 p = new char [l];
 if(!p) {
 cout << "Bellekte yer ayirma hatası\n";
 exit(1);
 }
 strcpy(p, s);
}

// Referans parametresi kullanılarak düzeltme.
void show(strtype &x)
{
 char *s;

 s = x.get();
 cout << s << "\n";
}

int main()
{
 strtype a("Hello"), b("There");

 show(a);
 show(b);

 return 0;
}

```

## 4.8. Alistirmalar

1. // Basit bir iki boyutlu, sınırlı dizi örneği.

```

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

class array {
 int isize, jsize;
 int *p;
public:
 array(int i, int j);
 int &put(int i, int j);
 int get(int i, int j);
};

array::array(int i, int j)
{
 p = new int [i*j];
 if(!p) {
 cout << "Bellekte yer ayirma hatası\n";
 exit(1);
 }
}
```

```

 isize = i;
 jsize = j;
}

// Diziye bir şeyler koy.
int &array::put(int i, int j)
{
 if(i<0 || i>=isize || j<0 || j>=jsize) {
 cout << "Sınır hatası!!!\n";
 exit(1);
 }
 return p[i*jsize + j]; // p[i]'ya referans döndür
}

// Diziden bir şeyler al.
int array::get(int i, int j)
{
 if(i<0 || i>=isize || j<0 || j>=jsize) {
 cout << "Sınır hatası!!!\n";
 exit(1);
 }
 return p[i*jsize + j]; // karakter döndür
}

int main()
{
 array a{2, 3};
 int i, j;

 for(i=0; i<2; i++)
 for(j=0; j<3; j++)
 a.put(i, j) = i+j;

 for(i=0; i<2; i++)
 for(j=0; j<3; j++)
 cout << a.get(i, j) << ' ';

 // sınır hatası oluştur
 a.put(10, 10);

 return 0;
}

```

2. Hayır. Fonksiyon tarafından döndürülen referans işaretçiye atanamaz.

## Pekiştirme Testi: Bölüm 4

- #include <iostream>  
 using namespace std;  
  
 class a\_type {
 double a, b;
 public:
 a\_type(double x, double y) {
 a = x;
 b = y;
 }
 void show() { cout << a << ' ' << b << '\n'; }
 };

```
int main()
{
 a_type ob[2][5] = {
 a_type(1, 1), a_type(2, 2),
 a_type(3, 3), a_type(4, 4),
 a_type(5, 5), a_type(6, 6),
 a_type(7, 7), a_type(8, 8),
 a_type(9, 9), a_type(10, 10)
 };

 int i, j;

 for(i=0; i<2; i++)
 for(j=0; j<5; j++)
 ob[i][j].show();
 cout << '\n';

 return 0;
}

2. #include <iostream>
using namespace std;

class a_type {
 double a, b;
public:
 a_type(double a, double b) {
 a = a;
 b = b;
 }
 void show() { cout << a << " " << b << '\n'; }
};

int main()
{
 a_type ob[2][5] = {
 a_type(1, 1), a_type(2, 2),
 a_type(3, 3), a_type(4, 4),
 a_type(5, 5), a_type(6, 6),
 a_type(7, 7), a_type(8, 8),
 a_type(9, 9), a_type(10, 10)
 };

 a_type *p;

 p = (a_type *) ob;

 int i, j;

 for(i=0; i<2; i++)
 for(j=0; j<5; j++) {
 p->show();
 p++;
 }
 cout << '\n';

 return 0;
}
```

3. **this** işaretçisi, üye fonksiyona kendiliğinden gönderilen bir işaretcidir ve çağrıyı gerçekleştiren nesneyi işaret eder.
4. **new** ve **delete**'in genel formları şöyledir:

```
p-var = new type;
delete p-var;
```

**new**'i kullanırken tip dönüşümü uygulamak zorunda değilsiniz. Nesnenin uzunluğu kendiliğinden belirlenir. **sizeof**'u da kullanmak zorunda değilsiniz. Aynı zamanda **<cstdlib>** başlığını programınıza dahil etmek şart değil.

5. Bir referans, aslında diğer bir değişkenin veya argümanın farklı bir ismi olan kapalı bir işaretçi sabitidir. Referans parametresi kullanmanın bir avantajı argümanın kopyasının oluşturulmamasıdır.

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

void recip(double &d);

int main()
{
 double x = 100.0;

 cout << "x " << x << "\n";

 recip(x);

 cout << "Reciprocal is " << x << "\n";

 return 0;
}

void recip(double &d)
{
 d = 1/d;
}
```

## Bütünleştirme Testi: Bölüm 4

1. İşaretçi kullanarak bir nesnenin üyesine erişmek için ok (->) operatörünü kullanın.

```
2. #include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
 char *p;
 int len;
public:
 strtype(char *ptr);
 ~strtype();
 void show();
};
```

```

strtype::strtype(char *ptr)
{
 len = strlen(ptr);
 p = new char [len+1];
 if(!p) {
 cout << "Beliekte yer ayırma hatası\n";
 exit(1);
 }
 strcpy(p, ptr);
}

strtype::~strtype()
{
 cout << "p serbest bırakılıyor\n";
 delete [] p;
}

void strtype::show()
{
 cout << p << " - length: " << len;
 cout << "\n";
}

int main()
{
 strtype s1("Bu bir deneme."), s2("C++'yi seviyorum.");
 s1.show();
 s2.show();

 return 0;
}

```

## Gözden Geçirme Testi: Bölüm 5

1. Bir referans, otomatik olarak referanslığı kaldırılan ve işaret ettiği nesne ile dönüştürülebilir özel bir işaretçi tipidir. Üç tip referans vardır: parametre referansları, bağımsız referanslar ve fonksiyonlar tarafından döndürülen referanslar.

```

#include <iostream>
using namespace std;

int main()
{
 float *f;
 int *i;

 f = new float;
 i = new int;

 if(!f || !i) {
 cout << "Beliekte yer ayırma hatası\n";
 return 1;
 }

 *f = 10.101;
 *i = 100;
}

```

```
 cout << *f << ' ' << *i << '\n';

 delete f;
 delete i;

 return 0;
}
```

3. Bir hazırlayıcı içeren new'in genel şekli aşağıda gösteriliyor:

```
p-var = new type (initializer);
```

Örneğin, bu bir tam sayı için yer ayırır ve ona 10 değerini verir:

```
int *p;
p = new int (10);
```

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

class samp {
 int x;
public:
 samp(int n) { x = n; }
 int getx() { return x; }
};

int main()
{
 samp A[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
 int i;

 for(i=0; i<10; i++) cout << A[i].getx() << ' ';
 cout << "\n";
 return 0;
}
```

5. Avantajları: Referans parametresi, çağrı içinde kullanılan nesnenin kopyasının oluşturulmamasına imkan tanır. Referans genellikle değerden daha hızlı gönderilir. Referans parametresi başvuru yoluyla çağrıma notasyonunu ve yordamını akıcı hale getirir ve hata ihtimalini azaltır.

Dezavantajları: Referans parametresinde yapılan değişiklikler çağrıda kullanılan değişkeni de değiştirir. Bir referans parametresi çağrıma rutinindeki yan etkileri için bir ortam oluşturur.

6. Hayır.

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

void mag(long &num, long order);

int main()
{
 long num;
 long order;
 cout << "Enter a number: ";
 cin >> num;
 cout << "Enter an order: ";
 cin >> order;
 mag(num, order);
 cout << num << endl;
}
```

```

long n = 4;
long o = 2;
cout << "4 raised to the 2nd order of magnitude is ";
mag(in, o);
cout << n << '\n';

return 0;
}

void mag(long &num, long order)
{
 for(; order; order--) num = num * 10;
}

```

## 5.1. Alıştırmalar

- #include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
char \*p;
int len;
public:
strtype();
strtype(char \*s, int l);
char \*getstring() { return p; }
int getlength() { return len; }
};

strtype::strtype()
{
p = new char [255];
if(!p) {
cout << "Bellekte yer ayirma hatali\n";
exit(1);
}
\*p = '\0'; // bog katar
len = 255;
}

strtype::strtype(char \*s, int l)
{
if(strlen(s) >= l) {
cout << "Bellekte çok az yer ayırmalı\n";
exit(l);
}

p = new char [l];

if(!p) {
cout << "Bellekte yer ayırma hatalı\n";
exit(l);
}

strcpy(p, s);
len = l;
}

```
int main()
{
 strtype s1;
 strtype s2("Bu bir denemedir", 100);

 cout << "s1: " << s1.getstring() << " - Length: ";
 cout << s1.getLength() << '\n';
 cout << "s2: " << s2.getstring() << " - Length: ";
 cout << s2.getLength() << '\n';

 return 0;
}

2. // Kronometre emülatörü
#include <iostream>
#include <ctime>
using namespace std;

class stopwatch {
 double begin, end;
public:
 stopwatch();
 stopwatch(clock_t t);
 ~stopwatch();
 void start();
 void stop();
 void show();
};

stopwatch::stopwatch()
{
 begin = end = 0.0;
}

stopwatch::stopwatch(clock_t t)
{
 begin = (double) t / CLOCKS_PER_SEC;
 end = 0.0;
}

stopwatch::~stopwatch()
{
 cout << "Stopwatch(kronometre) nesnesi yok ediliyor...";
 show();
}

void stopwatch::start()
{
 begin = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::stop()
{
 end = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::show()
{
 cout << "Geçen zaman: " << end - begin;
 cout << "\n";
}
```

```

int main()
{
 stopwatch watch;
 long i;

 watch.start();
 for(i=0; i<320000; i++) ; // for döngüsünü kur
 watch.stop();
 watch.show();

 // ilk değeri kullanarak nesneyi oluştur
 stopwatch s2(clock());
 for(i=0; i<250000; i++) ; // for döngüsünü kur
 s2.stop();
 s2.show();

 return 0;
}

```

## 5.2. Alıştırmalar

1. **obj** ve **temp** nesneleri normal olarak oluşturulur. Fakat, **temp**, **f( )** tarafından döndürüldüğünde geçici nesne oluşturulur ve kopya constructor'ına çağrıyı gerçekleştiren bu geçici nesnedir.
2. Programın yazılması sırasında, bir nesne **getval( )**'e gönderilirken birebir kopyası yapılır. **getval( )** döndüğünde bu kopya yok edilir ve bu nesne için ayrılan bellek (**p** tarafından işaretçi edilen) serbest bırakılır. Ancak bu, **getval( )**'in çağrılmasında kullanılan nesnenin gerek duyduğu bellektir. Programın doğru sürümü aşağıdadır. Bu problemden kurtulmak için constructor kopyası kullanılıyor.

```

// Bu program düzeltilmiştir.
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
 int *p;
public:
 myclass(int i);
 myclass(const myclass &o); // kopya constructor
 ~myclass() { delete p; }
 friend int getval(myclass o);
};

myclass::myclass(int i)
{
 p = new int;

 if(!p) {
 cout << "Bellekte yer ayırma hatası\n";
 exit(1);
 }
 *p = i;
}

```

```

// Kopya constructor
myclass::myclass(const myclass &o)
{
 p = new int; // kopyanın kendi belleğini ayırac
 if(!p) {
 cout << "Beldekte yer ayırma hatası\n";
 exit(1);
 }
 *p = *o.p;
}

int getval(myclass o)
{
 return *o.p; // değer al
}

int main()
{
 myclass a(1), b(2);

 cout << getval(a) << " " << getval(b);
 cout << "\n";
 cout << getval(a) << " " << getval(b);

 return 0;
}

```

3. Bir kopya constructor, nesne bir diğer nesneyi hazır hale getirmek için kullanılırken uyarılır. Normal constructor nesne oluşturulduğunda çağrıılır.

## 5.4. Alıştırmalar

- #include <iostream>  
# include <cstdlib>  
using namespace std;

long mystrtol(const char \*s, char \*\*end, int base = 10)
{
 return strtol(s, end, base);
}

int main()
{
 long x;
 char \*sl = "100234";
 char \*p;

 x = mystrtol(sl, &p, 16);
 cout << "Base 16: " << x << '\n';

 \*
 x = mystrtol(sl, &p, 10);
 cout << "Base 10: " << x << '\n';

 x = mystrtol(sl, &p); // Normalde 10'luk taban kullan
 cout << "Base 10 by default: " << x << '\n';

 return 0;
}

2. Varsayılan argümanları alan tüm parametreler, almayanların sağında bulunmalıdır. Yani, bir kere parametre varsayılanlarını verdiğinde başladığınızda bütün alt parametreler de varsayılanlara sahip olmak zorundadır. Soruda **q** verilmiş bir varsayılan değildir.
3. İmleci hareket ettiren fonksiyonlar derleyiciden ve ortamdan ortama fark etse de sadece bir çözüm gösterilmiştir. Aşağıdaki program komut istemi ortamında Borland C++'da çalışır:

```
// Not: Bu bir Borland C++ programıdır.
#include <iostream>
#include <conio.h>
using namespace std;

void myclreol(int len = -1);

int main()
{
 int i;

 gotoxy(1, 1);
 for(i=0; i<24; i++)
 cout << "abcdefghijklmnopqrstuvwxyz1234567890!`~";

 gotoxy(1, 2);
 myclreol();
 gotoxy(1, 4);
 myclreol(20);

 return 0;
}

// len parametresi belirlenen e kadar satır sonuna doğru sil.
void myclreol(int len)
{
 int x, y;

 x = wherex(); // x pozisyonunu al
 y = wherex(); // y pozisyonunu al

 if(len == -1) len = 60-x;

 int i = x;
 for(; i<len; i++) cout << ' ';
 gotoxy(x, y); // imleci sıfırla
}
```

4. Varsayılan argüman başka bir parametre veya yerel değişken alamaz.

## 5.6. Alıştırma

1. 

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

int dif(int a, int b)
```

```

 {
 return a-b;
 }

 float dif(float a, float b)
 {
 return a-b;
 }

 int main()
 {
 int (*p1)(int, int);
 float (*p2)(float, float);

 p1 = dif; // dif(int, int)'in adresi
 p2 = dif; // dif(float, float)'un adresi;

 cout << p1(10, 5) << ' ';
 cout << p2(10.5, 8.9) << '\n';

 return 0;
 }
}

```

## Pekiştirme Testi: Bölüm 5

- // time\_t için date()'i aşırı yükle.

```

#include <iostream>
#include <cstdio> // sscanf() için dahil edilmiştir
#include <ctime>
using namespace std;

class date {
 int day, month, year;
public:
 date(char *str);
 date(int m, int d, int y) {
 day = d;
 month = m;
 year = y;
 }
 // type time_t tipinin parametresi için aşırı yükle
 date(time_t t);
 void show() {
 cout << month << '/' << day << '/';
 cout << year << '\n';
 }
};

date::date(char *str)
{
 sscanf(str, "%d-%d-%d", &month, &day, &year);
}

date::date(time_t t)
{
 struct tm *p;

 p = localtime(&t); // bulunduğuunuz zamana çevirin
 day = p->tm_mday;
}

```

```

month = p->tm_mon;
year = p->tm_year;
}

int main()
{
 // katar kullanarak date nesnesini oluşturun
 date sdate("12/31/99");

 // tamsayılar kullanarak date nesnesini oluşturun
 date idate(12, 31, 99);
 /* time_t kullanarak date nesnesini oluşturun
 bu sistem saatini kullanarak bir nesne oluşturur */
 date tdate(time(NULL));

 sdate.show();
 idate.show();
 tdate.show();

 return 0;
}

```

- samp** sınıfı sadece bir constructor tanımlar ve bu constructor bir hazırlayıcıya ihtiyaç duyar. Bu yüzden bir constructor olmadan **samp** tipinde bir nesneyi deklare etmek doğru olmaz. (Yani, **samp x** geçersiz bir bildirimdir.)
- Bir constructor'ın aşırı yüklenmesinin nedenlerinden biri, esneklik sağlamaktır. Bu sayede size en uygun constructor'u seçme imkanı verir. Bir değeri ise hazırlanmış veya hazırlanmış nesnelerin deklare edilmesine imkan tanımıştır. Bir constructor'u aşırı yüklemek isteyebilirsiniz, böylece dinamik diziler için yer ayrılabilir.
- Kopya constructor'ın en genel şekli aşağıdaki gibidir:

```

sınıf-isim (const sınıfisamı &obj) {
 // constructor'ın gövdesi
}

```

- Bir kopya constructor'i, hazırlama olayı meydana geldiğinde çağrılr. Özellikle bir nesne diğerini hazırlamak için açıkça kullanıldığında ve bir nesne bir fonksiyona parametre olarak gönderildiğinde ve de nesne bir fonksiyon tarafından döndürüürken, geçici nesne oluşturulduğunda kopya constructor'i çağrılr.
- overload** anahtar kelimesi eskidir. C++'in ilk sürümlerinde bu, derleyiciye bir fonksiyonunun aşırı yükleneceğini haber vermek için kullanılırdı. Yeni derleyiciler tarafından artık desteklenmemektedir.
- Varsayılan argüman, fonksiyon çağrılrken uyuşmayan argüman ortaya çıktığında fonksiyon parametresine verilen değerdir.
- #include <iostream>
#include <cstring>
using namespace std;

void reverse(char \*str, int count = 0);

```

int main()
{
 char *s1 = "Bu bir deneme.";
 char *s2 = "C++'i seviyorum.";

 reverse(s1); // katarın tamamını tersine çevir
 reverse(s2, 7); // ilk 7 karakteri tersine çevir

 cout << s1 << '\n';
 cout << s2 << '\n';

 return 0;
}

void reverse(char *str, int count)
{
 int i, j;
 char temp;

 if(!count) count = strlen(str)-1;

 for(i=0, j=count; i<j; i++, j--) {
 temp = str[i];
 str[i] = str[j];
 str[j] = temp;
 }
}

```

9. Varsayılan argümanların tamamını alan parametreler almayanların sağında bulunmalıdır.
10. İki anlamlı varsayılan tip dönüşümleri, referans parametreleri ve varsayılan argümanlar ile tanıtolabilir.
11. İki anlamı vardır, çünkü derleyici **compute()**'un hangi sürümünü çağıracağını bilmemek. **divisor** normalleştiricisi ile olan ilk sürüm müdür? Veya sadece bir parametre alan ikinci sürüm müdür?
12. Aşırı yüklenmiş bir fonksiyonun adresini elde ederken, hangi fonksiyonun kullanılacağına karar veren, işaretçinin tipidir.

## Bütünleştirme Testi: Bölüm 5

```

1. #include <iostream>
using namespace std;

void order(int sa, int sb)
{
 int t;

 if(a<b) return;
 else { // a ile b'yi değiştir
 t = a;
 a = b;
 b = t;
 }
}

```

```

int main()
{
 int x=10, y=5;

 cout << "x: " << x << ", y: " << y << "\n";
 order(x, y);

 cout << "x: " << x << ", y: " << y << "\n";
 return 0;
}

```

- Referans parametresi alan bir fonksiyonun çağrılmış notasyonu ile değer parametresi alan bir fonksiyonunki ile aynıdır.
- Varsayılan argüman, fonksiyonları aşırı yüklemenin kısa bir yoludur, çünkü net sonuç aynıdır. Örneğin,

```
int f(int a, int b = 0);
```

fonksiyonel olarak aşağıdaki aşırı yüklenmiş fonksiyonlarla aynıdır:

```

int f(int a):s
int f(int a, int b):i

4. #include <iostream>
using namespace std;

class samp {
 int a;
public:
 samp() { a = 0; }
 samp(int n) { a = n; }
 int get_a() { return a; }
};

int main()
{
 samp ob(88);
 samp obarray[10];
 // ...
}

```

- Kopya constructor'ları, sizin, yani programcınızı nesne kopyasının nasıl oluşturulduğunu kesin olarak kontrol etmeniz gerektiğinde kullanılır. Bu sadece varsayılan birebir kopya istenmeyen yan etkiler oluşturduğunda önemlidir.

## Gözden Geçirme Testi: Bölüm 6

- class myclass {
 int x, y;
public:
 myclass(int i, int j) { x=i; y=j; }
 myclass() { x=0; y=0; }
};

- ```
2. class myclass {
    int x, y;
public:
    myclass(int i=0, int j=0) { x=i; y=j; }
}
```
3. Normal argümanlar başlatıldığında normalleştirilmeyen parametreler kullanılmaz.
4. Sürümleri arasındaki tek fark bir sürümünün değer parametresi olması ve diğer sürümünün referans parametresi olması olan bir fonksiyon asırı yüklenemez. (Derleyici onları ayırt edemez.)
5. Sıkça ortaya çıkacak bir veya daha fazla normal değerin olması durumunda normal argümanları kullanmak doğrudur. Ortaya çıkma ihtimali olan değer veya değerler yoksa bu doğru değildir.
6. Hayır, çünkü dinamik bir dizinin hazırlığını yapmanın bir yolu yoktur. Bu sınıfın sadece bir constructor'ı vardır ve bu constructor'un hazırlayıcılara ihtiyacı vardır.
7. Bir kopya constructor'ı, bir nesne diğerini hazırlayıken çağrılan özel bir constructor'dır. Bu durum aşağıda belirtilen üç şekilde de ortaya çıkabilir : Bir nesne açıkça bir diğerinin hazırlığını yaparken kullanıldığından, bir nesne, bir fonksiyona gönderildiğinde veya geçici bir nesne, fonksiyonun döndürme değeri olarak oluşturulduğundan.

6.2. Alıştırmalar

```
1. // coord sınıfına bağlı olarak * ve / asırı yükle
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator*(coord ob2);
    coord operator/(coord ob2);
}

// coord sınıfına bağlı olarak * asırı yükleme
coord coord::operator*(coord ob2)
{
    coord temp;
    temp.x = x * ob2.x;
    temp.y = y * ob2.y;
    return temp;
}

// coord sınıfına bağlı olarak / asırı yükleme
coord coord::operator/(coord ob2)
{
```

```

coord temp;

temp.x = x / ob2.x;
temp.y = y / ob2.y;

return temp;
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 * o2;
    o3.get_xy(x, y);
    cout << "(o1*o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 / o2;
    o3.get_xy(x, y);
    cout << "(o1/o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

2. % operatörünü aşırı yüklemek pek uygun degildir çünkü işlevinin geleneksel kullanım ile ilgisi yoktur.

6.3. Alistirma

```

1. // coord sınıfına bağlı olarak < and > aşırı yükleme
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    int operator<(coord ob2);
    int operator>(coord ob2);
};

// coord için < operatörü aşırı yükleme
int coord::operator<(coord ob2)
{
    return x<ob2.x && y<ob2.y;
}

// Aşırı yükleme the > operator for coord.
int coord::operator>(coord ob2)
{
    return x>ob2.x && y>ob2.y;
}

int main()
{
    coord o1(10, 10), o2(5, 3);

```

```
if(o1>o2) cout << "o1 > o2\n";
else cout << "o1 <= o2 \n";

if(o1<o2) cout << "o1 < o2\n";
else cout << "o1 >= o2\n";

return 0;
}
```

6.4. Alıştırmalar

1. // coord sınıfına göre -- aşırı yükleme

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

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator--(); // ön ek
    coord operator--(int notused); // postfix
};

// coord sınıfı için ön ek - aşırı yükleme
coord coord::operator--()
{
    x--;
    y--;
    return *this;
}

// coord sınıfı için postfix - aşırı yükleme
coord coord::operator--(int notused)
{
    x--;
    y--;
    return *this;
}

int main()
{
    coord o1(10, 10);
    int x, y;

    o1--; // nesneyi bir azalt
    o1.get_xy(x, y);
    cout << "(o1--) X: " << x << ", Y: " << y << "\n";

    --o1; // nesneyi bir azat
    o1.get_xy(x, y);
    cout << "(-o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```
2. //coord sınıfına bağlı olarak + aşırı yükleme

```
#include <iostream>
```

```

using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator+(coord ob2); // ikili toplam
    coord operator+(); // birli toplam
};

// coord sınıfına bağlı olarak + aşırı yükleme
coord coord::operator+(coord ob2)
{
    coord temp;
    temp.x = x + ob2.x;
    temp.y = y + ob2.y;
    return temp;
}

// Coord sınıfı için birli + aşırı yükleme
coord coord::operator+()
{
    if(x<0) x = -x;
    if(y<0) y = -y;
    return *this;
}

int main()
{
    coord o1(10, 10), o2(-2, -2);
    int x, y;

    o1 = o1 + o2; // toplama
    o1.get_xy(x, y);
    cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

    o2 = +o2; // mutlak değer
    o2.get_xy(x, y);
    cout << "(+o2) X: " << x << ", Y: " << y << "\n";
}

```

6.5. Alistirmalar

- /* Arkadaş fonksiyonları kullanarak coord sınıfına bağlı şekilde
- ve / aşırı yüklemek */

```

#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }

```

```
coord(int i, int j) { x=i; y=j; }
void get_xy(int &i, int &j) { i=x; j=y; }
friend coord operator-(coord ob1, coord ob2);
friend coord operator/(coord ob1, coord ob2);
};

// coord sınıfına bağlı, arkadaş kullanarak - aşırı yükleme
coord operator-(coord ob1, coord ob2)
{
    coord temp;

    temp.x = ob1.x - ob2.x;
    temp.y = ob1.y - ob2.y;

    return temp;
}

// coord sınıfına bağlı, arkadaş kullanarak / aşırı yükleme
coord operator/(coord ob1, coord ob2)
{
    coord temp;

    temp.x = ob1.x / ob2.x;
    temp.y = ob1.y / ob2.y;

    return temp;
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 - o2;
    o3.get_xy(x, y);
    cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 / o2;
    o3.get_xy(x, y);
    cout << "(o1/o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}

2. // ob*int ve int*ob için * aşırı yükleme
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    friend coord operator*(coord ob1, int i);
    friend coord operator*(int i, coord ob2);
};

// * aşırı yüklemenin bir yolu
coord operator*(coord ob1, int i)
{
```

```

coord temp;

temp.x = ob1.x * i;
temp.y = ob1.y * i;

return temp;
}

// * diğer yolla aşırı yükleme
coord operator*(int i, coord ob2)
{
    coord temp;
    temp.x = ob2.x * i;
    temp.y = ob2.y * i;

    return temp;
}

int main()
{
    coord o1(10, 10), o2;
    int x, y;

    o2 = o1 * 2; // ob * int
    o2.get_xy(x, y);
    cout << "(o1*2) X: " << x << ", Y: " << y << "\n";

    o2 = 3 * o1; // int * ob
    o2.get_xy(x, y);
    cout << "(3*o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

3. Arkadaş fonksiyonları kullanarak, iç tipleri sol operand olarak tutmanız mümkün olur. Üye fonksiyonlar kullanıldığında sol operand operatörü tanımlanan, bir sınıfın nesnesi olmalı.

```

4. // Arkadaş kullanarak coord sınıfına bağlı - aşırı yükleme
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    friend coord operator--(coord &ob); // ön ek
    friend coord operator--(coord &ob, int notused); // postfix
};

// Arkadaş kullanarak coord sınıfına bağlı -- (ön ek) aşırı yükleme
coord operator--(coord &ob)
{
    ob.x--;
    ob.y--;
    return ob;
}

```

```

// Arkadaş kullanarak coord sınıfına bağlı -- (postfix) asıri yükleme
coord operator--(coord &ob, int notused)
{
    ob.x--;
    ob.y--;
    return ob;
}

int main()
{
    coord ol(10, 10);
    int x, y;

    --ol; // nesneyi bir azalt
    ol.get_xy(x, y);
    cout << "(--ol) X: " << x << ", Y: " << y << "\n";

    ol--; // nesneyi bir azalt
    ol.get_xy(x, y);
    cout << "(ol--) X: " << x << ", Y: " << y << "\n";

    return 0;
}

```

6.6. Alıştırma

- #include <iostream>
#include <cstdlib>
using namespace std;

class dynarray {
 int *p;
 int size;
public:
 dynarray(int s);
 int &put(int i);
 int get(int i);
 dynarray &operator=(dynarray &ob);
};

//yapılendirici
dynarray::dynarray(int s)
{
 p = new int [s];
 if(!p) {
 cout << "Bellekte yer ayırma hatası\n";
 exit(1);
 }
 size = s;
}
// Bir elemanı sakla
int &dynarray::put(int i)
{
 if(i<0 || i>=size) {
 cout << "Sınır hatası!\n";
 exit(1);
 }
 return p[i];
}

```

// elemanı al
int dynarray::get(int i)
{
    if(i<0 || i>=size) {
        cout << "Sınır hatası!\n";
        exit(1);
    }
    return p[i];
}

// dynarray için = aşırı yükleme
dynarray &dynarray::operator=(dynarray &ob)
{
    int i;
    if(size!=ob.size) {
        cout << "Cannot copy arrays of differing sizes!\n";
        exit(1);
    }
    for(i = 0; i<size; i++) p[i] = ob.p[i];
    return *this;
}

int main()
{
    int i;
    dynarray ob1(10), ob2(10), ob3(100);

    ob1.put(3) = 10;
    i = ob1.get(3);
    cout << i << "\n";

    ob2 = ob1;
    i = ob2.get(3);
    cout << i << "\n";

    // Hata üretir
    ob1 = ob3; // !!!
    return 0;
}

```

6.7. Alıştırmalar

- #include <iostream>
 #include <cstring>
 #include <cstdlib>
 using namespace std;

 class strtype {
 char *p;
 int len;
 public:
 strtype(char *s);
 ~strtype() {
 cout << "Freeing " << (unsigned) p << '\n';
 delete [] p;
 }
 char *get() { return p; }
 strtype &operator=(strtype &ob);
 char &operator[](int i);
 };

```
strtype::strtype(char *s)
{
    int l;
    l = strlen(s)+1;
    p = new char [l];
    if(!p) {
        cout << "Bellekte yer ayirma hatası\n";
        exit(1);
    }
    len = l;
    strcpy(p, s);
}

// bir nesne ata
strtype &strtype::operator=(strtype &ob)
{
    // daha fazla bellek geriye koy mu bak
    if(len < ob.len) { // bellekte daha fazla yere ihtiyaç var
        delete [] p;
        p = new char [ob.len];
        if(!p) {
            cout << "Bellekte yer ayirma hatası\n";
            exit(1);
        }
    }
    len = ob.len;
    strcpy(p, ob.p);
    return *this;
}

// Katarada indeks karakterleri
char &strtype::operator[](int i)
{
    if(i<0 || i>len-1) {
        cout << "\nIndex value of ";
        cout << i << " is out-of-bounds.\n";
        exit(1);
    }
    return p[i];
}

int main()
{
    strtype a("Hello"), b("There");
    cout << a.get() << '\n';
    cout << b.get() << '\n';

    a = b; // p'nin artık üstüne yazılmıştır
    cout << a.get() << '\n';
    cout << b.get() << '\n';

    // dizi indekslemesi kullanarak karakterlere erişin
    cout << a[0] << a[1] << a[2] << '\n';

    // dizi indekslemesi kullanarak karakterler atayın
    a[0] = 'X';
    a[1] = 'Y';
    a[2] = 'Z';
```

```
cout << a.get() << "\n";
return 0;
}

2. #include <iostream>
#include <cstdlib>
using namespace std;

class dynarray {
    int *p;
    int size;
public:
    dynarray(int s);
    dynarray &operator=(dynarray &ob);
    int &operator[](int i);
};

// Yapılandırıcı
dynarray::dynarray(int s)
{
    p = new int [s];
    if(!p) {
        cout << "Bellekte yer ayırma hatası\n";
        exit(1);
    }

    size = s;
}

// dynarray.için = aşırı yükleme
dynarray &dynarray::operator=(dynarray &ob)
{
    int i;
    if(size!=ob.size) {
        cout << "Cannot copy arrays of differing sizes!\n";
        exit(1);
    }

    for(i = 0; i<size; i++) p[i] = ob.p[i];
    return *this;
}

// Aşırı yükleme []
int &dynarray::operator[](int i)
{
    if(i<0 || i>size) {
        cout << "\nIndex value of ";
        cout << i << " is out-of-bounds.\n";
        exit(1);
    }
    return p[i];
}

int main()
{
    int i;

    dynarray ob1(10), ob2(10), ob3(100);

    ob1[3] = 10;
    i = ob1[3];
```

```

cout << i << "\n";
ob2 = ob1;
i = ob2[3];
cout << i << "\n";
//Hata üretir
ob1 = ob3; // uzunlukları farklı diziler
return 0;
}

```

Pekiştirme Testi: Bölüm 6

```

1. // << and >>. Aşırı yükleme
#include <iostream>
using namespace std;

class coord {
    int x, y; // koordinat değerleri
public:
    coord() { x=0; y=0; }
    coord(int i, int j) { x=i; y=j; }
    void get_xy(int &i, int &j) { i=x; j=y; }
    coord operator<<(int i);
    coord operator>>(int i);
};

// Aşırı yükleme <<.
coord coord::operator<<(int i)
{
    coord temp;

    temp.x = x << i;
    temp.y = y << i;

    return temp;
}

// Aşırı yükleme >>.
coord coord::operator>>(int i)
{
    coord temp;

    temp.x = x >> i;
    temp.y = y >> i;
    return temp;
}

int main()
{
    coord o1(4, 4), o2;
    int x, y;

    o2 = o1 << 2; // ob << int
    o2.get_xy(x, y);
    cout << "(o1<<2) X: " << x << ", Y: " << y << "\n";

    o2 = o1 >> 2; // ob >> int
}

```

```
    o2.get_xy(x, y);
    cout << "{'l>>2} X: " << x << ", Y: " << y << "\n";
    return 0;
}
2. #include <iostream>
using namespace std;

class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
    {
        x = i; y = j; z = k;
    }
    three_d() { x=0; y=0; z=0; }
    void get(int &i, int &j, int &k)
    {
        i = x; j = y; k = z;
    }
    three_d operator+(three_d ob2);
    three_d operator-(three_d ob2);
    three_d operator++();
    three_d operator--();
}

three_d three_d::operator+(three_d ob2)
{
    three_d temp;

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;
    temp.z = z + ob2.z;

    return temp;
}

three_d three_d::operator-(three_d ob2)
{
    three_d temp;

    temp.x = x - ob2.x;
    temp.y = y - ob2.y;
    temp.z = z - ob2.z;

    return temp;
}

three_d three_d::operator++()
{
    x++;
    y++;
    z++;

    return *this;
}

three_d three_d::operator--()
{
    x--;
    y--;
    z--;

    return *this;
}
```

```

        return *this;
    }

int main()
{
    three_d o1(10, 10, 10), o2(2, 3, 4), o3;
    int x, y, z;

    o3 = o1 + o2;
    o3.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    o3 = o1 - o2;
    o3.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    ++o1;
    o1.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    --o1;
    o1.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    return 0;
}

3. #include <iostream>
using namespace std;

class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
    {
        x = i; y = j; z = k;
    }
    three_d() { x=0; y=0; z=0; }
    void get(int &i, int &j, int &k)
    {
        i = x; j = y; k = z;
    }
    three_d operator+(three_d &ob2);
    three_d operator-(three_d &ob2);
    friend three_d operator++(three_d &ob);
    friend three_d operator--(three_d &ob);
};

three_d three_d::operator+(three_d &ob2)
{
    three_d temp;

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;
    temp.z = z + ob2.z;

    return temp;
}

```

```

three_d three_d::operator-(three_d &ob2)
{
    three_d temp;
    temp.x = x - ob2.x;
    temp.y = y - ob2.y;
    temp.z = z - ob2.z;

    return temp;
}

three_d operator++(three_d &ob)
{
    ob.x++;
    ob.y++;
    ob.z++;
    return ob;
}

three_d operator--(three_d &ob)
{
    ob.x--;
    ob.y--;
    ob.z--;
    return ob;
}

int main()
{
    three_d o1(10, 10, 10), o2(2, 3, 4), o3;
    int x, y, z;

    o3 = o1 + o2;
    o3.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    o3 = o1 - o2;
    o3.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    ++o1;
    o1.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    --o1;
    o1.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";
}

```

4. Bir ikili üye operatör fonksiyonu **this** işaretçisi ile sol operandı açık şekilde gönderilir. İkili arkadaş operatör fonksiyonu her iki operandı açık şekilde gönderir. Birli üye operatör fonksiyonlarının açık parametreleri yoktur. Bir arkadaş unary operatör fonksiyonunun ise bir parametresi vardır.

5. Normal bitleştirilmiş kopyası yeterli gelmediğinde = operatörünü aşırı yüklemek gerekebilir. Örneğin, bir nesnenin sadece bazı bilgilerini, diğer bir nesneye atamak istediğiniz durumlar olabilir.

6. Hayır.

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

class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
    {
        x = i; y = j; z = k;
    }
    three_d() { x=0; y=0; z=0; }
    void get(int &i, int &j, int &k)
    {
        i = x; j = y; k = z;
    }
    friend three_d operator+(three_d ob, int i);
    friend three_d operator+(int i, three_d ob);
};

three_d operator+(three_d ob, int i)
{
    three_d temp;

    temp.x = ob.x + i;
    temp.y = ob.y + i;
    temp.z = ob.z + i;

    return temp;
}

three_d operator+(int i, three_d ob)
{
    three_d temp;

    temp.x = ob.x + i;
    temp.y = ob.y + i;
    temp.z = ob.z + i;

    return temp;
}

int main()
{
    three_d o1(10, 10, 10);
    int x, y, z;

    o1 = o1 + 10;
    o1.get(x, y, z);
    cout << "X: " << x << ", Y: " << y
    cout << ", Z: " << z << "\n";

    o1 = -20 + o1;
    o1.get(x, y, z);
```

```

cout << "X: " << x << ", Y: " << y;
cout << ", Z: " << z << "\n";

return 0;
}

8. #include <iostream>
using namespace std;
class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
    {
        x = i; y = j; z = k;
    }
    three_d() { x=0; y=0; z=0; }
    void get(int &i, int &j, int &k)
    {
        i = x; j = y; k = z;
    }
    int operator==(three_d ob2);
    int operator!=(three_d ob2);
    int operator||(three_d ob2);
};

int three_d::operator==(three_d ob2)
{
    return x==ob2.x && y==ob2.y && z==ob2.z;
}

int three_d::operator!=(three_d ob2)
{
    return x!=ob2.x && y!=ob2.y && z!=ob2.z;
}

int three_d::operator||(three_d ob2)
{
    return x||ob2.x && y||ob2.y && z||ob2.z;
}

int main()
{
    three_d o1(10, 10, 10), o2(2, 3, 4), o3(0, 0, 0);

    if(o1==o1) cout << "o1 == o1\n";
    if(o1!=o2) cout << "o1 != o2\n";
    if(o3 || o1) cout << "o1 or o3 is true\n";

    return 0;
}

```

9. Sınıf içerisinde bir dizinin normal dizi indeksleme notasyonu ile indekslenmesi için depolanması gerekiğinden [] genellikle aşırı yüklenir.

Bütünleştirme Testi: Bölüm 6

```
1. /* Açık olsun diye hata denetimi kullanılmamıştır. Ancak eğer bu mu gerçek
uygulamalarda kullanacağınız birkaç denetim eklenmelisiniz. */
#include <iostream>
#include <cstring>
using namespace std;

class strtype {
    char s[80];
public:
    strtype() { *s = '\0'; }
    strtype(char *p) { strcpy(s, p); }
    char *get() { return s; }
    strtype operator+(strtype s2);
    strtype operator=(strtype s2);
    int operator<(strtype s2);
    int operator>(strtype s2);
    int operator==(strtype s2);
};

strtype strtype::operator+(strtype s2)
{
    strtype temp;
    strcpy(temp.s, s);
    strcat(temp.s, s2.s);

    return temp;
}

strtype strtype::operator=(strtype s2)
{
    strcpy(s, s2.s);

    return *this;
}

int strtype::operator<(strtype s2)
{
    return strcmp(s, s2.s) < 0;
}

int strtype::operator>(strtype s2)
{
    return strcmp(s, s2.s) > 0;
}

int strtype::operator==(strtype s2)
{
    return strcmp(s, s2.s) == 0;
}

int main()
{
    strtype o1("Hello"), o2(" There"), o3;
    o3 = o1 + o2;
    cout << o3.get() << "\n";
}
```

```

o3 = o1;
if(o1==o3) cout << "o1 equals o3\n";
if(o1>o2) cout << "o1 > o2\n";
if(o1<o2) cout << "o1 < o2\n";

return 0;
}

```

Gözden Geçirme Testi: Bölüm 7

1. Hayır. Bir operatörü aşırı yüklemek, onun çalşabileceği veri tiplerini genişletmektedir. Ancak bundan önceki işlemler bu olaydan etkilenmezler.
2. Evet. C++'ın, iç tiplerinden birine bağlı olarak bir operatörü aşırı yükleyemezsiniz.
3. Hayır önceki değiştirilemez. Hayır operandların sayısı değiştirilemez.
4. #include <iostream>
using namespace std;

```

class array {
    int num[10];
public:
    array();
    void set(int n[10]);
    void show();
    array operator+(array ob2);
    array operator-(array ob2);
    int operator==(array ob2);
};

array::array()
{
    int i;
    for(i=0; i<10; i++) num[i] = 0;
}

void array::set(int *n)
{
    int i;
    for(i=0; i<10; i++) num[i] = n[i];
}

void array::show()
{
    int i;

    for(i=0; i<10; i++)
        cout << num[i] << ' ';
    cout << "\n";
}

array array::operator+(array ob2)
{
    int i;
    array temp;

    for(i=0; i<10; i++)
        temp.num[i] = num[i] + ob2.num[i];
}

```

```

        return temp;
    }

    array array::operator-(array ob2)
    {
        int i;
        array temp;

        for(i=0; i<10; i++)
            temp.nums[i] = nums[i] - ob2.nums[i];

        return temp;
    }

    int array::operator==(array ob2)
    {
        int i;
        for(i=0; i<10; i++)
            if(nums[i]!=ob2.nums[i]) return 0;

        return 1;
    }

    int main()
    {
        array o1, o2, o3;

        int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        o1.set(i);
        o2.set(i);

        o3 = o1 + o2;
        o3.show();

        o3 = o1 - o3;
        o3.show();

        if(o1==o2) cout << "o1 equals o2\n";
        else cout << "o1 does not equal o2\n";

        if(o1==o3) cout << "o1 equals o3\n";
        else cout << "o1 does not equal o3\n";

        return 0;
    }
}

5. #include <iostream>
using namespace std;

class array {
    int nums[10];
public:
    array();
    void set(int n[10]);
    void show();
    friend array operator+(array ob1, array ob2);
    friend array operator-(array ob1, array ob2);
    friend int operator==(array ob1, array ob2);
};

```

```
array::array()
{
    int i;

    for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
    int i;

    for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
    int i;

    for(i=0; i<10; i++)
        cout << nums[i] << " ";
    cout << "\n";
}

array operator+(array ob1, array ob2)
{
    int i;
    array temp;

    for(i=0; i<10; i++)
        temp.nums[i] = ob1.nums[i] + ob2.nums[i];

    return temp;
}

array operator-(array ob1, array ob2)
{
    int i;
    array temp;

    for(i=0; i<10; i++)
        temp.nums[i] = ob1.nums[i] - ob2.nums[i];

    return temp;
}

int operator==(array ob1, array ob2)
{
    int i;

    for(i=0; i<10; i++)
        if(ob1.nums[i] != ob2.nums[i]) return 0;
    return 1;
}

int main()
{
    array o1, o2, o3;
    int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    o1.set(i);
    o2.set(i);
```

```

    o3 = o1 + o2;
    o3.show();

    o3 = o1 - o3;
    o3.show();

    if(o1==o2) cout << "o1 equals o2\n";
    else cout << "o1 does not equal o2\n";

    if(o1==o3) cout << "o1 equals o3\n";
    else cout << "o1 does not equal o3\n";

    return 0;
}

6. #include <iostream>
using namespace std;

class array {
    int nums[10];
public:
    array();
    void set(int n[10]);
    void show();
    array operator++();
    friend array operator--(array obj);
};

array::array()
{
    int i;

    for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
    int i;

    for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
    int i;

    for(i=0; i<10; i++)
        cout << nums[i] << ' ';
    cout << "\n";
}

// Üye fonksiyon kullanarak birlik op'u aşırı yükleme
array array::operator++()
{
    int i;

    for(i=0; i<10; i++)
        nums[i]++;
}

return *this;
}

```

```
// Arkadaş kullan
array operator--(array ob)
{
    int i;

    for(i=0; i<10; i++)
        ob.numbers[i]--;

    return ob;
}

int main()
{
    array o1, o2, o3;
    int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    o1.set(i);
    o2.set(i);

    o3 = ++o1;
    o3.show();

    o3 = --o1;
    o3.show();

    return 0;
}
```

7. Hayır! Bir atama operatörünü aşırı yüklemek için arkadaş fonksiyon kullanmalısınız.

7.1. Alıştırmalar

1. A ve C geçerli ifadelerdir.
2. Tabanın bir public üyesi, public olarak miras alınmışsa türetilmiş sınıfın bir public üyesi haline gelir. Tabanın public üyesi private olarak miras alınmışsa türetilmiş sınıfın private üyesi haline gelir.

7.2. Alıştırmalar

1. Taban sınıfın protected üyesi public olarak miras alınmış ise türetilmiş sınıfın protected üyesi haline gelir. Eğer private olarak miras alınmışsa, türetilmiş sınıfın private üyesi haline gelir. Eğer protected olarak miras alınmışsa sınıfın protected üyesi haline gelir.
2. protected kategorisine taban sınıfın, türetilmiş sınıflara erişme izni olan bazı üyelerini private olarak saklayabilmesi için gerek duyulmuştur.
3. Hayır.

7.3. Alistirmalar

```
1. #include <iostream>
#include <cstring>
using namespace std;

class mybase {
    char str[80];
public:
    mybase(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

class myderived : public mybase {
    int len;
public:
    myderived(char *s) : mybase(s) {
        len = strlen(s);
    }
    int getlen() { return len; }
    void show() { cout << get() << '\n'; }
};

int main()
{
    myderived ob("hello");

    ob.show();
    cout << ob.getlen() << '\n';

    return 0;
}

2. #include <iostream>
using namespace std;

// Çeşitli taşıtlar için taban sınıf
class vehicle {
    int num_wheels;
    int range;
public:
    vehicle(int w, int r)
    {
        num_wheels = w; range = r;
    }
    void showv()
    {
        cout << "Wheels: " << num_wheels << '\n';
        cout << "Range: " << range << '\n';
    }
};

class car : public vehicle {
    int passengers;
public:
    car(int p, int w, int r) : vehicle(w, r)
    {
        passengers = p;
    }
}
```

```

void show()
{
    show();
    cout << "Passengers: " << passengers << '\n';
}
};

class truck : public vehicle {
    int loadlimit;
public:
    truck(int l, int w, int r) : vehicle(w, r)
    {
        loadlimit = l;
    }
    void show()
    {
        show();
        cout << "loadlimit " << loadlimit << '\n';
    }
};

int main()
{
    car c(5, 4, 500);

    truck t(30000, 12, 1200);

    cout << "Car: \n";
    c.show();
    cout << "\nTruck:\n";
    t.show();

    return 0;
}

```

7.4. Alıştırmalar

1. Constructing A
Constructing B
Constructing C
Destructuring C
Destructuring B
Destructuring A
2. #include <iostream>
using namespace std;


```

class A {
    int i;
public:
    A(int a) { i = a; }
};

class B {
    int j;
public:
    B(int a) { j = a; }
};

```

```

class C : public A, public B {
    int k;
public:
    C(int c, int b, int a) : A(a), B(b) {
        k = c;
    }
};

```

7.5. Alıştırmalar

3. Sanal taban sınıfı, türetilmiş bir sınıfın aynı taban sınıfından türetilmiş iki (veya daha fazla) sınıfı kağıtum yoluyla aldığı zaman gerek duyulur. Sanal taban sınıf olmadan, ortak taban sınıfın iki (veya daha fazla) kopyası son türetilmiş sınıfta bulunur. Ancak, eğer ilk taban sınıf sanalsa, türetilmiş sınıfta sadece bir kopya bulunur.

Pekiştirme Testi: Bölüm 7

```

1. #include <iostream>
using namespace std;

class building {
protected:
    int floors;
    int rooms;
    double footage;
};

class house : public building {
    int bedrooms;
    int bathrooms;
public:
    house(int f, int r, double ft, int br, int bth) {
        floors = f; rooms = r; footage = ft;
        bedrooms = br; bathrooms = bth;
    }
    void show() {
        cout << "floors: " << floors << '\n';
        cout << "rooms: " << rooms << '\n';
        cout << "square footage: " << footage << '\n';
        cout << "bedrooms: " << bedrooms << '\n';
        cout << "bathrooms: " << bathrooms << '\n';
    }
};

class office : public building {
    int phones;
    int extinguishers;
public:
    office(int f, int r, double ft, int p, int ext) {
        floors = f; rooms = r; footage = ft;
        phones = p; extinguishers = ext;
    }
    void show() {
        cout << "floors: " << floors << '\n';
        cout << "rooms: " << rooms << '\n';
        cout << "square footage: " << footage << '\n';
    }
};

```

```

    cout << "Telephones: " << phones << '\n';
    cout << "fire extinguishers: ";
    cout << extinguishers << '\n';
}

int main()
{
    house h_ob(2, 12, 5000, 6, 4);
    office o_ob(4, 25, 12000, 30, 8);

    cout << "House: \n";
    h_ob.show();

    cout << "\nOffice: \n";
    o_ob.show();

    return 0;
}

```

2. Bir taban sınıf public olarak miras alındığında, tabanın public üyeleri türetilmiş sınıfın public üyeleri haline gelir ve tabanın private üyeleri tabana göre yine private kahr. Eğer taban private olarak miras alındığında, tabanın bütün üyeleri türetilmiş sınıfın private üyeleri haline gelir.
3. **protected** olarak bildirilmiş üyeleri taban sınıfı private'tır. Fakat herhangi bir türetilmiş sınıfından miras alınabilir (ve erişilebilir). **protected**, inheritance erişim belirteci ile kullanıldığında taban sınıfın bütün public ve protected üyelerinin, türetilmiş sınıfın protected üyeleri olmasına sebep olur.
4. constructor'lar (yapıldıkları) türetilme sırasında göre çağrırlar. destructor'lar (yok-ediciler) ise ters sırada çağrılr.

```

5. #include <iostream>
using namespace std;

class planet {
protected:
    double distance; // Güneşten olan mil uzaklıği
    int revolve; // in days
public:
    planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
    double circumference; // Ortam veya yörunge
public:
    earth(double d, int r) : planet(d, r) {
        circumference = 2*distance*3.1416;
    }
    void show() {
        cout << "Distance from sun: " << distance << '\n';
        cout << "Days in orbit: " << revolve << '\n';
        cout << "Circumference of orbit: ";
        cout << circumference << '\n';
    }
};

```

```

int main()
{
    earth ob(93000000, 365);
    ob.show();
    return 0;
}

```

6. Programı düzeltmek için, **motorized** ve **road_use**'u sanal taban sınıfı olarak **vehicle**'in miras almasını sağlayın. Ayrıca bu bölümdeki bütünllestirme testinin soru 1'ine göz atın.

Bütünllestirme Testi: Bölüm 7

1. Bazı derleyiciler **switch** fonksiyonunu satır arası fonksiyon olarak kullanmanıza izin vermez. Eğer derleyiciniz bu yaklaşımı sergiliyorsa, fonksiyonlar kendisinden "regular" fonksiyonlara dönüştürülür.
2. Atama operatörü miras alınmayan tek operatördür. Bu neden sebebiini zaten kolaydır. Türetilmiş sınıf taban **class**'ta bulunmayan üyeleri içerdikinden taban sınıfı bağlı aşırı yüklenmiş =, türetilmiş sınıfta eklenmiş üyelerin varlığından habersiz olacaktır. Örneğin, bu nedenle doğru şekilde o yem üyeleri kopyalayamaz.

Gözden Geçirme Testi: Bölüm 8

```

1. #include <iostream>
using namespace std;

class airship {
protected:
    int passengers;
    double cargo;
};

class airplane : public airship {
    char engine; // Pirpir için p, jet için j
    double range;
public:
    airplane(int p, double c, char e, double r)
    {
        passengers = p;
        cargo = c;
        engine = e;
        range = r;
    }
    void show();
};

class balloon : public airship {
    char gas; // hidrojen için h, helyum için e
    double altitude;
public:
    balloon(int p, double c, char g, double a)

```

```

    {
        passengers = p;
        cargo = c;
        gas = g;
        altitude = a;
    }
    void show();
};

void airplane::show()
{
    cout << "Passengers: " << passengers << '\n';
    cout << "Cargo capacity: " << cargo << '\n';
    cout << "Engine: ";
    if(engine=='p') cout << "Propeller\n";
    else cout << "Jet\n";
    cout << "Range: " << range << '\n';
}

void balloon::show()
{
    cout << "Passengers: " << passengers << '\n';
    cout << "Cargo capacity: " << cargo << '\n';
    cout << "Gas: ";
    if(gas=='h') cout << "Hydrogen\n";
    else cout << "Helium\n";
    cout << "Altitude: " << altitude << '\n';
}

int main()
{
    balloon b(2, 500.0, "h", 12000.0);
    airplane b727(100, 40000.0, "j", 45000.0);

    b.show();
    cout << "\n";
    b727.show();

    return 0;
}

```

- protected** erişim belirteci bir sınıf üyesinin kendi sınıfına **private** olmasını, aynı zamanda türetilmiş her sınıfında o sınıf üyesine erişebilmesini sağlar.
- Program constructor'ların (yapılandırıcıların) ve destructor'ların (yok-edicilerin) çağrıldığını gösteren aşağıdaki çıkışı vermektedir.

```

Constructing A
Constructing B
Constructing C
Destructing C
Destructing B
Destructing A

```

- Constructor'lar ABC sırasıyla, destructor'lar ise CBA sırasıyla çağrırlırlar.
- #include <iostream>
using namespace std;

```
class base {
```

```

    int i, j;
public:
    base(int x, int y) { i = x; j = y; }
    void showij() { cout << i << ' ' << j << '\n'; }
};

class derived : public base {
    int k;
public:
    derived(int a, int b, int c) : base(b, c) {
        k = a;
    }
    void show() { cout << k << ' ' ; showij(); }
};

int main()
{
    derived ob(1, 2, 3);

    ob.show();

    return 0;
}

```

6. Unutulan kelimeler "general" ve "specific" dir.

8.2. Alıştırmalar

- #include <iostream>
using namespace std;

int main()
{
 cout.setf(ios::showpos);

 cout << -10 << ' ' << 10 << '\n';

 return 0;
}
- #include <iostream>
using namespace std;

int main()
{
 cout.setf(ios::showpoint | ios::uppercase | ios::scientific);

 cout << 100.0;

 return 0;
}
- #include <iostream>
using namespace std;

int main()
{
 ios::fmtflags f;

```

    f = cout.flags(); // bayrakları sakla
    cout.unsetf(ios::dec);
    cout.setf(ios::showbase | ios::hex);
    cout << 100 << '\n';

    cout.flags(f); // bayrakları sıfırla

    return 0;
}

```

8.3. Alıştırmalar

1. // 2'den 100'e kadar log ve log102num tablosunu oluşturun.

```

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

int main()
{
    double x;

    cout.precision(5);
    cout << "      x      log x      ln e\n\n";
    for(x = 2.0; x <= 100.0; x++) {
        cout.width(10);
        cout << x << " ";
        cout.width(10);
        cout << log10(x) << " ";
        cout.width(10);
        cout << log(x) << '\n';
    }

    return 0;
}

```
2. #include <iostream>

```

#include <cstring>
using namespace std;

void center(char *s);

int main()
{
    center("Hi there!");
    center("C++'i seviyorum.");

    return 0;
}

void center(char *s)
{
    int len;

    len = 40+(strlen(s)/2);
    cout.width(len);
    cout << s << '\n';
}

```

8.4. Alıştırmalar

- 1a. // 2'den 100'e kadar log ve log10nun tablosunu oluşturun.
- ```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
 double x;

 cout << setprecision(5);
 cout << " x log x ln x\n";
 cout << "-----\n";

 for(x = 2.0; x <= 100.0; x++) {
 cout << setw(10) << x << " ";
 cout << setw(10) << log(x) << " ";
 cout << setw(10) << log(x) << "\n";
 }

 return 0;
}
```
- 1b. #include <iostream>  
#include <iomanip>  
#include <cstring>  
using namespace std;
- ```
void center(char *s);

int main()
{
    center("Hi there!");
    center("C++'i seviyorum.");
    return 0;
}

void center(char *s)
{
    int len;

    len = 40+(strlen(s)/2);

    cout << setw(len) << s << "\n";
}
```
2. cout << setiosflags(ios::showbase | ios::hex) << 100;
3. Çıkış veri akışında **boolalpha** bayrağını ayarlamak boolean değerlerin *true* ve *false* kelimeleriyle gösterilmesini sağlar. Bir giriş sisteminde **boolalpha**'yı ayarlamak size, Boolean değerleri *true* ve *false* kelimeleriyle girme imkanı verir.

8.5. Alistirmalar

```
1. #include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr)
    -strtype() (delete [] p)
    friend ostream &operator<<(ostream &stream, strtype &ob);
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr)+1;
    p = new char [len];
    if(!p) {
        cout << "Sistemde yer ayirma hatasi\n";
        exit(1);
    }
    strcpy(p, ptr);
}

ostream &operator<<(ostream &stream, strtype &ob)
{
    stream << ob.p;
    return stream;
}

int main()
{
    strtype s1("Bu bir deneme."), s2("C++'i seviyorum.");

    cout << s1;
    cout << endl << s2 << endl;

    return 0;
}

2. #include <iostream>
using namespace std;

class planet {
protected:
    double distance; // g̃neşten olan mil uzaklığı
    int revolve; // günde
public:
    planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
    double circumference; // ortam veya yörülge
public:
    earth(double d, int r) : planet(d, r) {
        circumference = 2*distance*3.1416;
    }
}
```

```
    friend ostream &operator<<(ostream &stream, earth ob);
}

ostream &operator<<(ostream &stream, earth ob)
{
    stream << "Distance from sun: " << ob.distance << "\n";
    stream << "Days in orbit: " << ob.revolve << "\n";
    stream << "Circumference of orbit: " << ob.circumference;
    stream << "\n";

    return stream;
}

int main()
{
    earth ob(93000000, 365);
    cout << ob;

    return 0;
}
```

3. Bir ekleyici üye fonksiyon olamaz çünkü ekleyici'ye çağrı yapan nesne kullanıcı tarafından tanımlanmış bir sınıfın nesnesi *değildir*.

8.6. Alıştırmalar

```
1. #include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype() {delete [] p;}
    friend ostream &operator<<(ostream &stream, strtype &ob);
    friend istream &operator>>(istream &stream, strtype &ob);
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr)+1;
    p = new char [len];
    if(!p) {
        cout << "Bellekte yer ayirma hatası\n";
        exit(1);
    }
    strcpy(p, ptr);
}

ostream &operator<<(ostream &stream, strtype &ob)
{
    stream << ob.p;

    return stream;
}
```

```

istream &operator>>(istream &stream, strtype &ob)
{
    char temp[255];

    stream >> temp;

    if(strlen(temp) >= ob.len) {
        delete [] ob.p;
        ob.len = strlen(temp)+1;
        ob.p = new char [ob.len];
        if(!ob.p) {
            cout << "Bellekte yer ayirma hatası\n";
            exit(1);
        }
    }
    strcpy(ob.p, temp);

    return stream;
}

int main()
{
    strtype s1("Bu bir deneme."), s2("C++'yi seviyorum.");

    cout << s1;
    cout << '\n' << s2;

    cout << "\nEnter a string: ";
    cin >> s1;
    cout << s1;

    return 0;
}

2. #include <iostream>
using namespace std;

class factor {
    int num; // sayı
    int lfact; // en düşük faktör
public:
    factor(int i);
    friend ostream &operator<<(ostream &stream, factor ob);
    friend istream &operator>>(istream &stream, factor &ob);
};

factor::factor(int i)
{
    int n;

    num = i;
    for(n=2; n < (i/2); n++)
        if(!(i%n)) break;

    if(n<(i/2)) lfact = n;
    else lfact = 1;
}

istream &operator>>(istream &stream, factor &ob)
{
    stream >> ob.num;
}

```

```
int n;

for(n=2; n < (ob.num/2); n++)
    if(!(ob.num%n)) break;
if(n<(ob.num/2)) ob.lfact = n;
else ob.lfact = 1;

return stream;
}

ostream &operator<<(ostream &stream, factor ob)
{
    stream << ob.lfact << " is lowest factor of ";
    stream << ob.num << '\n';

    return stream;
}

int main()
{
    factor o(32);
    cout << o;

    cin >> o;
    cout << o;

    return 0;
}
```

Pekiştirme Testi: Bölüm 8

1.

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

int main()
{
    cout << 100 << ' ';

    cout.unsetf(ios::dec); // dec bayrağını sil
    cout.setf(ios::hex);
    cout << 100 << ' ';

    cout.unsetf(ios::hex); // hex bayrağını sil
    cout.setf(ios::oct);
    cout << 100 << '\n';

    return 0;
}
```
2.

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

int main()
{
    cout.setf(ios::left);
    cout.precision(2);
    cout.fill('*');
    cout.width(20);
```

```
cout << 1000.5364 << '\n';

return 0;
}

3a. #include <iostream>
using namespace std;

int main()
{
    cout << 100 << ' ';
    cout << hex << 100 << ' ';
    cout << oct << 100 << '\n';

    return 0;
}

3b. #include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << setiosflags(ios::left);
    cout << setprecision(2);
    cout << setfill('*');
    cout << setw(20);

    cout << 1000.5364 << '\n';

    return 0;
}

4. ios::fmtflags f;

f = cout.flags(); // kaydet
// ...
cout.flags(f); // yükle

5. #include <iostream>
using namespace std;

class pwr {
    int base;
    int exponent;
    double result; // exp usse taban
public:
    pwr(int b, int e);
    friend ostream &operator<<(ostream &stream, pwr ob);
    friend istream &operator>>(istream &stream, pwr &ob);
};

pwr::pwr(int b, int e)
{
    base = b;
    exponent = e;
    result = 1;
```

```

        for( ; e-- ) result = result * base;
    }

ostream &operator<<(ostream &stream, pwr ob)
{
    stream << ob.base << "^^ " << ob.exponent;
    stream << " is " << ob.result << "\n";
    return stream;
}

istream &operator>>(istream &stream, pwr &ob)
{
    int b, e;

    cout << "Enter base and exponent: ";
    stream >> b >> e;

    pwr temp(b, e); // geçici oluştur
    ob = temp;

    return stream;
}

int main()
{
    pwr ob(10, 2);

    cout << ob;

    cin >> ob;

    cout << ob;

    return 0;
}

6. // Bu program kutu çizer
#include <iostream>
using namespace std;

class box {
    int len;
public:
    box(int l) { len = l; }
    friend ostream &operator<<(ostream &stream, box ob);
};

// Bir kutu çiz.
ostream &operator<<(ostream &stream, box ob)
{
    int i, j;

    for(i=0; i<ob.len; i++) stream << "*";
    stream << '\n';
    for(i=0; i<ob.len-2; i++) {
        stream << '*';
        for(j=0; j<ob.len-2; j++) stream << ' ';
        stream << "**\n";
    }
}

```

```
    for(i=0; i<ob.len; i++) stream << '**';
    stream << '\n';

    return stream;
}

int main()
{
    box b1(4), b2(7);

    cout << b1 << endl << b2;

    return 0;
}
```

Bütünleştirme Testi: Bölüm 8

```
1. #include <iostream>
using namespace std;
#define SIZE 10

// Karakterler için bir yiğin sınıfı bildir
class stack {
    char stck[SIZE]; // yiğini tut
    int tos; // yiğinin üst indeksi
public:
    stack();
    void push(char ch); // yiğine karakter it
    char pop(); // yiğinden karakter çek
    friend ostream &operator<<(ostream &stream, stack ob);
};

// stack'ı (yiğini) başlat
stack::stack()
{
    tos = -1;
}

// Karakter it
void stack::push(char ch)
{
    if(tos==SIZE) {
        cout << "Stack is full\n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Karakter çek
char stack::pop()
{
    if(tos==0) {
        cout << "Stack is empty\n";
        return 0; // Yiğin boşsa null döndürür
    }
    tos--;
    return stck[tos];
}
```

```
ostream &operator<<(ostream &stream, stack ob)
{
    char ch;

    while(ch=ob.pop()) stream << ch;
    stream << endl;

    return stream;
}

int main()
{
    stack s;

    s.push('a');
    s.push('b');
    s.push('c');

    cout << s;
    cout << s;

    return 0;
}

2. #include <iostream>
#include <ctime>
using namespace std;

class watch {
    time_t t;
public:
    watch() { t = time(NULL); }
    friend ostream &operator<<(ostream &stream, watch ob);
};

ostream &operator<<(ostream &stream, watch ob)
{
    struct tm *localt;

    localt = localtime(&ob.t);

    stream << asctime(localt) << endl;

    return stream;
}

int main()
{
    watch w;

    cout << w;

    return 0;
}

3. #include <iostream>
using namespace std;

class ft_to_inches {
    double feet;
    double inches;
```

```
public:  
    void set(double f)  
    {  
        feet = f;  
        inches = f * 12;  
    }  
    friend istream &operator>>(istream &stream, ft_to_inches &ob);  
    friend ostream &operator<<(ostream &stream, ft_to_inches ob);  
};  
  
istream &operator>>(istream &stream, ft_to_inches &ob)  
{  
    double f;  
  
    cout << "Enter feet: ";  
    stream >> f;  
    ob.set(f);  
  
    return stream;  
}  
  
ostream &operator<<(ostream &stream, ft_to_inches ob)  
{  
    stream << ob.feet << " feet is " << ob.inches;  
    stream << " inches\n";  
  
    return stream;  
}  
  
int main()  
{  
    ft_to_inches x;  
  
    cin >> x;  
    cout << x;  
  
    return 0;  
}
```

Gözden Geçirme Testi: Bölüm 9

1.

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    cout.width(40);  
    cout.fill(':');  
  
    cout << "C++ is fun" << '\n';  
  
    return 0;  
}
```
2.

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    cout.precision(4);
```

- ```
 cout << 10.0/3.0 << '\n';

 return 0;
}

3. #include <iostream>
#include <iomanip>
using namespace std;

int main()
{
 cout << setprecision(4) << 10.0/3.0 << '\n';

 return 0;
}

4. Bir ekleyici, sınıf versini çıkış akışına veren aşırı yüklenmiş operator <<() fonksiyonudur. Extractor ise sınıf versini giriş akışına veren aşırı yüklenmiş operator >>() fonksiyonudur.
```
- ```
5. #include <iostream>
using namespace std;

class date {
    char d[9]; // tarihi katar olarak aa/gg/yy şeklinde sakla
public:
    friend ostream &operator<<(ostream &stream, date ob);
    friend istream &operator>>(istream &stream, date &ob);
};

ostream &operator<<(ostream &stream, date ob)
{
    stream << ob.d << '\n';

    return stream;
}

istream &operator>>(istream &stream, date &ob)
{
    cout << "Enter date (mm/dd/yy): ";
    stream >> ob.d;

    return stream;
}

int main()
{
    date ob;

    cin >> ob;
    cout << ob;
    return 0;
}
```
6. Parametreleştirilmiş manipülatör kullanmak için programınıza **<iomanip>**'i dahil etmelisiniz.
7. Önceden tanımlanmış akımlar **cin**, **cout**, **cerr**, ve **clog**'tur.

9.1. Alıştırmalar

1. // tarihi ve saatı göster.
#include <iostream>
#include <ctime>
using namespace std;

// Saat ve Tarih çıkış manipulator'u.
ostream &td(ostream &stream)
{
 struct tm *localt;
 time_t t;

 t = time(NULL);
 localt = localtime(&t);
 stream << asctime(localt) << endl;

 return stream;
}

int main()
{
 cout << td << '\n';

 return 0;
}
2. #include <iostream>
using namespace std;

// Hex çıkışını X ile aktifleştir.
ostream &sethex(ostream &stream)
{
 stream.unsetf(ios::dec | ios::oct);
 stream.setf(ios::hex | ios::uppercase | os::showbase);

 return stream;
}

// Bayrakları sıfırla
ostream &reset(ostream &stream)
{
 stream.unsetf(ios::hex | ios::uppercase | ios::showbase);
 stream.setf(ios::dec);
 return stream;
}

int main()
{
 cout << sethex << 100 << '\n';
 cout << reset << 100 << '\n';
 return 0;
}
3. #include <iostream>
using namespace std;

// 10 karakter geç
istream &skipchar(istream &stream)
{
 int i;

```

    char c;

    for(i=0; i<10; i++) stream >> c;

    return stream;
}

int main()
{
    char str[80];

    cout << "Enter some characters: ";
    cin >> skipchar >> str;

    cout << str << '\n';

    return 0;
}

```

9.2. Alıştırmalar

- // Bir text dosyasını kopyala ve kaç karakter kopyalandığını bildir.

```

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

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Usage: CPY <input> <output>\n";
        return 1;
    }

    ifstream fin(argv[1]); // Giriş dosyasını aç
    ofstream fout(argv[2]); // Çıkış dosyasını aç

    if(!fin) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    if(!fout) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    char ch;
    unsigned count=0;

    fin.unsetf(ios::skipws); // boşlukları atla
    while(!fin.eof()) {
        fin >> ch;
        if(!fin.eof()) {
            fout << ch;
            count++;
        }
    }

    cout << "Number of bytes copied: " << count << '\n';
}

```

```
fin.close();
fout.close();

return 0;
}
```

Programın, dizini listelediğinizde gösterilen değerden farklı bildirmesi, bazı karakter dönüşümlerinin meydana geldiğinden kaynaklanır. Özellikle satırbaşı ve satır besleme okunduğunda, bu yeni satıra çevrilir. Çıkışta bu yeni satırlar bir karakter olarak sayılır fakat yine satırbaşı ve satır besleme işaretine dönüştürülür.

2.

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

int main()
{
    ofstream pout("phone");

    if(!pout) {
        cout << "Cannot open PHONE file.\n";
        return 1;
    }

    pout << "Isaac Newton 415 555-3423\n";
    pout << "Robert Goddard 213 555-2312\n";
    pout << "Enrico Fermi 202 555-1111\n";

    pout.close();

    return 0;
}
```
3. // Sözcük sayma

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

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Usage: COUNT <input>\n";
        return 1;
    }
    ifstream in(argv[1]);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }
    int count=0;
    char ch;

    in >> ch; // boşluk olmayan ilk karakteri bul

    // boşluk olmayan ilk karakteri bulduğunda boşluk atlama
    in.unsetf(ios::skipws); // boşluk atlama
```

```

        while(!in.eof()) {
            in >> ch;
            if(isspace(ch)) {
                count++;
                while(isspace(ch) && !in.eof()) in >> ch;
            }
        }

        cout << "Word count: " << count << '\n';

        in.close();

        return 0;
    }
}

```

4. eğer uyarılan akım açık dosyaya bağlanmışsa `is_open()` fonksiyonu true değeri döndürür.

9.3. Alıştırmalar

1a. // Bir dosyayı kopyala ve kaç karakter kopyalandığını bildir.

```

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

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Usage: CPY <input> <output>\n";
        return 1;
    }

    ifstream fin(argv[1], ios::in | ios::binary); // giriş dosyasını aç
    ofstream fout(argv[2], ios::out | ios::binary); // çıkış dosyasını oluştur

    if(!fin) {
        cout << "Cannot open input file\n";
        return 1;
    }

    if(!fout) {
        cout << "Cannot open output file\n";
        return 1;
    }

    char ch;
    unsigned count=0;

    while(!fin.eof()) {
        fin.get(ch);
        if(!fin.eof()) {
            fout.put(ch);
            count++;
        }
    }

    cout << "Number of bytes copied: " << count << '\n';
}

```

```
fin.close();
fout.close();

return 0;
}

1b. // sözcük sayma
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Usage: COUNT <input>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    int count=0;
    char ch;

    // boşluk olmayan ilk karakteri bul
    do {
        in.get(ch);
    } while(ispace(ch));

    while(!in.eof()) {
        in.get(ch);
        if(ispace(ch)) {
            count++;
            while(ispace(ch) && !in.eof()) in.get(ch); // Bir sonraki sözcüğü bul
        }
    }

    cout << "Word count: " << count << '\n';

    in.close();

    return 0;
}

2. // ekleyici kullanarak bir hesap bilgisini dosyaya al
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class account {
    int custnum;
    char name[80];
    double balance;
public:
    account(int c, char *n, double b)
```

```
    {
        custnum = c;
        strcpy(name, n);
        balance = b;
    }

    friend ostream &operator<<(ostream &stream, account ob);
};

ostream &operator<<(ostream &stream, account ob)
{
    stream << ob.custnum << ' ';
    stream << ob.name << ' ' << ob.balance;
    stream << '\n';

    return stream;
}

int main()
{
    account Rex(1011, "Ralph Rex", 12323.34);
    ofstream out("accounts", ios::out | ios::binary);

    if(!out) {
        cout << "Cannot open output file.\n";
        return 1;
    }

    out << Rex;

    out.close();
    return 0;
}
```

9.4. Alıştırmalar

1. // Boşluk içeren katarları okumak için get() kullanın.

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

int main()
{
    char str[80];

    cout << "Enter your name: ";
    cin.get(str, 79);

    cout << str << '\n';
    return 0;
}
```

Program get() veya getline() kullanırsa da aynı işlevi görüyor.

2. // Bir metin dosyasını görüntülemek için getline()'ı kullanın

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

int main(int argc, char *argv[])
{
    ifstream in(argv[1]);
    string str;
```

```

    if(argc!=2) {
        cout << "usage: PR <filename>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    char str[255];

    while(!in.eof()) {
        in.getline(str, 254);
        cout << str << '\n';
    }

    in.close();
    return 0;
}

```

9.5. Alıştırmalar

1. // Dosyayı ekranda tersten göster

```

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

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "usage: REVERSE <filename>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        return 1;
    }

    char ch;
    long i;

    // dosyanın sonuna git(eof karakterine kadar)
    in.seekg(0, ios::end);
    i = (long) in.tellg(); // dosyada kaç bayt olduğuna bak
    i-->2; // eof'tan önce yedekle

    for( ;i>=0; i--) {
        in.seekg(i, ios::beg);
        in.get(ch);
        cout << ch;
    }
}

```

```

        in.close();
    }

    return 0;
}

2. // Dosyadaki karakterleri değiştir.
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "usage: SWAP <filename>\n";
        return 1;
    }

    // Dosyayı giriş çıkış için aç
    ifstream io(argv[1], ios::in | ios::out | ios::binary);
    if(!io) {
        cout << "Cannot open file.\n";
        return 1;
    }

    char ch1, ch2;
    long i;

    for(i=0 ; !io.eof(); i+=2) {
        io.seekg(i, ios::beg);
        io.get(ch1);
        if(io.eof()) continue;
        io.get(ch2);
        if(io.eof()) continue;
        io.seekg(i, ios::beg);
        io.put(ch2);
        io.put(ch1);
    }

    io.close();

    return 0;
}

```

9.6. Alistirma

```

1a. /* Dosyayı ekranda tersten göster ve hata denetimi yap */
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "usage: REVERSE <filename>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);

```

```
if(!in) {
    cout << "Cannot open input file\n";
    return 1;
}

char ch;
long i;

// dosyamın sonuna git(eof karakterine kadar)
in.seekg(0, ios::end);
if(!in.good()) return 1;
i = (long) in.tellg(); // dosyada kaç bayt olduğuna bak
if(!in.good()) return 1;
i -= 2; // eof'tan önce yedekle

for( ;i>=0; i--) {
    in.seekg(i, ios::beg);
    if(!in.good()) return 1;
    in.get(ch);
    if(!in.good()) return 1;
    cout << ch;
}

in.close();
if(!in.good()) return 1;

return 0;
}
```

1b. // Hata denetimi yaparak dosyadaki karakterleri değiştir.

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

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "usage: SWAP <filename>\n";
        return 1;
    }

    // Dosyayı giriş çıkış için aç
    fstream io(argv[1], ios::in | ios::out | ios::binary);

    if(!io) {
        cout << "Cannot open file.\n";
        return 1;
    }

    char ch1, ch2;
    long i;

    for(i=0 ;!io.eof(); i+=2) {
        io.seekg(i, ios::beg);
        if(!io.good()) return 1;
        io.get(ch1);
        if(io.eof()) continue;
        io.get(ch2);
        if(!io.good()) return 1;
        if(io.eof()) continue;
        io.seekg(i, ios::beg);
```

```

        if(!io.good()) return 1;
        io.put(ch2);
        if(!io.good()) return 1;
        io.put(ch1);
        if(!io.good()) return 1;
    }

    io.close();
    if(!io.good()) return 1;

    return 0;
}

```

Pekiştirme Testi: Bölüm 9

- ```

#include <iostream>
using namespace std;

ostream &tabs(ostream &stream)
{
 stream << '\t' << '\t' << '\t';
 stream.width(20);

 return stream;
}

int main()
{
 cout << tabs << "Testing\n";

 return 0;
}

```
- ```

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

istream &findalpha(istream &stream)
{
    char ch;

    do {
        stream.get(ch);
    } while(!isalpha(ch));
    return stream;
}

int main()
{
    char str[80];
    cin >> findalpha >> str;
    cout << str << '\n';

    return 0;
}

```
- ```

// Bir dosyanın büyük küçük harflerini değiştirerek kopyala
#include <iostream>
#include <fstream>

```

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

int main(int argc, char *argv[])
{
 char ch;

 if(argc!=3) {
 cout << "Usage: COPYREV <source> <target>\n";
 return 1;
 }

 ifstream in(argv[1]);
 if(!in) {
 cout << "Cannot open input file.\n";
 return 1;
 }

 ofstream out(argv[2]);
 if(!out) {
 cout << "Cannot open output file.\n";
 return 1;
 }

 while(in.get(ch))
 {
 if(isupper(ch)) ch = tolower(ch);
 else ch = toupper(ch);
 out.put(ch);
 }

 in.close();
 out.close();
 return 0;
}

// Harfleri say
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int alpha[26];

int main(int argc, char *argv[])
{
 char ch;
 if(argc!=2) {
 cout << "Usage: COUNT <source>\n";
 return 1;
 }

 ifstream in(argv[1]);
 if(!in) {
 cout << "Cannot open input file.\n";
 }
}
```

```

 return 1;
 }

 // alpha[] sıfırla
 int i;
 for(i=0; i<26; i++) alpha[i] = 0;

 while(!in.eof()) {
 ch = in.get();
 if(isalpha(ch)) { // harf bulunsrsa say
 ch = toupper(ch); // normallestir
 alpha[ch-'A']++; // 'A'-'A' == 0, 'B'-'A' == 1, gibi
 }
 }

 // toplamı göster
 for(i=0; i<26; i++) {
 cout << (char) ('A'+ i) << ":" << alpha[i] << "\n";
 }

 in.close();

 return 0;
}

5a. /* Copy a file and reverse case of letters with error checking. */
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main(int argc, char *argv[])
{
 char ch;

 if(argc!=3) {
 cout << "Usage: COPYREV <source> <target>\n";
 return 1;
 }

 ifstream in(argv[1]);

 if(!in) {
 cout << "Cannot open input file.\n";
 return 1;
 }
 ofstream out(argv[2]);
 if(!out) {
 cout << "Cannot open output file";
 return 1;
 }
 while(!in.eof()) {
 ch = in.get();
 if(!in.good() && !in.eof()) return 1;
 if(!in.eof()) {
 if(islower(ch)) ch = toupper(ch);
 else ch = tolower(ch);
 out.put(ch);
 if(!out.good()) return 1;
 }
 }
}

```

```

 in.close();
 out.close();

 if(!in.good() && !out.good()) return 1;

 return 0;
 }

5b. // toplamı hata denetimi yaparak gerçekleştirir
#include <iostream>
#include <ifstream>
#include <cctype>
using namespace std;

int alpha[26];

int main(int argc, char *argv[])
{
 char ch;

 if(argc!=2) {
 cout << "Usage: COUNT <source>\n";
 return 1;
 }

 ifstream in(argv[1]);

 if(!in) {
 cout << "Cannot open input file.\n";
 return 1;
 }

 // alpha[]'yi hazırla
 int i;
 for(i=0; i<26; i++) alpha[i] = 0;

 while(!in.eof()) {
 ch = in.get();
 if(!in.good() && !in.eof()) return 1;
 if(isalpha(ch)) { // harf bulunursa say
 ch = toupper(ch); // normalleştir
 alpha[ch-'A']++; // 'A'-'A' == 0, 'B'-'A' == 1, gibi
 }
 }

 //toplamı göster
 for(i=0; i<26; i++) {
 cout << (char) ('A'+ i) << ":" << alpha[i] << '\n';
 }

 in.close();
 if(!in.good()) return 1;

 return 0;
}

```

6. get işaretçisini ayarlamak için, seekg( )'yi kullanın. put işaretçisini ayarlamak için seekp( )'yi kullanın.

## Bütünleştirme Testi: Bölüm 9

```
1. #include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

#define SIZE 40

class inventory {
 char item[SIZE]; // birimin isı
 int onhand; // eldeki sayı
 double cost; // birimin maliyeti
public:
 inventory(char *i, int o, double c)
 {
 strcpy(item, i);
 onhand = o;
 cost = c;
 }
 void store(fstream &stream);
 void retrieve(fstream &stream);
 friend ostream &operator<<(ostream &stream, inventory ob);
 friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
 stream << ob.item << ":" << ob.onhand;
 stream << " on hand at $" << ob.cost << '\n';

 return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
 cout << "Enter item name: ";
 stream >> ob.item;
 cout << "Bir sayı girin on hand: ";
 stream >> ob.onhand;
 cout << "Enter cost: ";
 stream >> ob.cost;

 return stream;
}

void inventory::store(fstream &stream)
{
 stream.write(item, SIZE);
 stream.write((char *) &onhand, sizeof(int));
 stream.write((char *) &cost, sizeof(double));
}

void inventory::retrieve(fstream &stream)
{
 stream.read(item, SIZE);
 stream.read((char *) &onhand, sizeof(int));
 stream.read((char *) &cost, sizeof(double));
}
```

```
int main()
{
 fstream inv("inv", ios::out | ios::binary);

 int i;

 inventory pliers("pliers", 12, 4.95);
 inventory hammers("hammers", 5, 9.45);
 inventory wrenches("wrenches", 22, 13.90);
 inventory temp("", 0, 0.0);

 if(!inv) {
 cout << "Cannot open file for output.\n";
 return 1;
 }
 //dosyasına yaz
 pliers.store(inv);
 hammers.store(inv);
 wrenches.store(inv);
 inv.close();

 // girişi içten sağ
 inv.open("inv", ios::in | ios::binary);

 if(inv) {
 cout << "Cannot open file for input.\n";
 return 1;
 }

 do {
 cout << "Record # (-1 to quit): ";
 cin >> i;
 if(i == -1) break;
 inv.seekg(i*(SIZE+sizeof(int)+sizeof(double)), ios::beg);
 temp.retrieve(inv);
 cout << temp;
 } while(inv.good());

 inv.close();
 return 0;
}
```

## Gözden Geçirme Testi: Bölüm 10

- #include <iostream>
using namespace std;

ostream &setsci(ostream &stream)
{
 stream.setf(ios::scientific | ios::uppercase);

 return stream;
}

int main()
{
 double f = 123.23;
 cout << setsci << f;

## EK B: Cevaplar

```

 cout << '\n';

 return 0;
 }

2. // Sekmeleri kopyala ve boşluğa dönüştür
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
 if(argc!=3) {
 cout << "Usage: COPY <in> <out>\n";
 return 1;
 }

 ifstream in(argv[1]);
 if(!in) {
 cout << "Cannot open input file.\n";
 return 1;
 }

 ofstream out(argv[2]);
 if(!out) {
 cout << "Cannot open output file.\n";
 return 1;
 }

 char ch;
 int i = 8;

 while(!in.eof()) {
 in.get(ch);
 if(ch=='\t') for(; i>0; i--) out.put(' ');
 else out.put(ch);
 if(i == -1 || ch=='\n') i = 8;
 i--;
 }

 in.close();
 out.close();

 return 0;
}

3. // Dosyayı ara
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

int main(int argc, char *argv[])
{
 if(argc!=3) {
 cout << "Usage: SEARCH <file> <word>\n";
 return 1;
 }

 ifstream in(argv[1]);
 if(!in) {

```

```

 cout << "Cannot open input file.\n";
 return 1;
}

char str[255];
int count=0;

while(!fin.eof()) {
 in >> str;
 if(!strcmp(str, argv[2])) count++;
}

cout << argv[2] << " found " << count;
cout << " number of times.\n";

in.close();

return 0;
}

```

4. İfade:

```
out.seekp(234, ios::beg);
```

5. Bu fonksiyonlar sunlardır: **rdstate()**, **good()**, **eof()**, **fail()**, ve **bad()**.
6. C++'ın I/O sistemi oluşturduğunuz sınıflar üzerinde çalışacak şekilde özelleştirebilir.

## 10.2. Alıştırmalar

```

1. #include <iostream>
using namespace std;

class num {
public:
 int i;
 num(int x) { i = x; }
 virtual void shownum() { cout << i << '\n'; }
};

class outhex : public num {
public:
 outhex(int n) : num(n) {}
 void shownum() { cout << hex << i << '\n'; }
};

class outoct : public num {
public:
 outoct(int n) : num(n) {}
 void shownum() { cout << oct << i << '\n'; }
};

int main()
{
 outoct o(10);
 outhex h(20);
 o.shownum();
}

```

```

 h.shownum();
 return 0;
 }

2. #include <iostream>
using namespace std;

class dist {
public:
 double d;
 dist(double f) { d = f; }
 virtual void trav_time()
 {
 cout << "Travel time at 60 mph: ";
 cout << d / 60 << '\n';
 }
};

class metric : public dist {
public:
 metric(double f) : dist(f) {}
 void trav_time()
 {
 cout << "Travel time at 100 kph: ";
 cout << d / 100 << '\n';
 }
};

int main()
{
 dist *p, mph(88.0);
 metric kph(88);

 p = &mph;
 p->trav_time();

 p = &kph;
 p->trav_time();

 return 0;
}

```

### 10.3. Alıştırmalar

2. Tanımda olduğu gibi bir soyut sınıf en azından saf sanal bir fonksiyon içermelidir. Bu demektir ki o sınıfa bağlı olarak fonksiyonun içeriği yoktur. Bu sebeple, nesne oluşturmamın bir yolu yoktur çünkü sınıf tanımlaması henüz tamamlanmamıştır.
3. **derived1**'e bağlı olarak **func()** çağrılarında, **base** içerisindeki **func()** kullanılır. Bunun çalışabilmesinin sebebi sanal fonksiyonların hiyerarşik olmasıdır.

### 10.4. Alıştırma

1. // Sanal fonksiyonları gösterir.
 

```
#include <iostream>
#include <cstdlib>
```

```
using namespace std;

class list {
public:
 list *head; // listenin başına işaretçi
 list *tail; // listenin sonuna işaretçi
 list *next; // bir sonraki elemana işaretçi
 int num; // saklanacak değer

 list() { head = tail = next = NULL; }
 virtual void store(int i) = 0;
 virtual int retrieve() = 0;
};

// kuyruk tipi liste oluştur
class queue : public list {
public:
 void store(int i);
 int retrieve();
};

void queue::store(int i)
{
 list *item;

 item = new queue;
 if(item)
 cout << "Bellekte yaradılmış hatalı.\n";
 else
 cout << "Bellekte yaradılmış.\n";

 item->num = i;
 item->next = NULL;

 // listenin sonuna boy
 if(item) tail->next = item;
 tail = item;
 item->next = NULL;
 if(item) head = tail;
}

int queue::retrieve()
{
 int n;
 list *p;

 if(head) {
 cout << "List empty.\n";
 return 0;
 }

 // listenin başından çıkar
 n = head->num;
 p = head;
 head = head->next;
 delete p;
 return n;
}

// Yığın tipi liste oluştur.
class stack : public list {
public:
 void store(int i);
```

## EK B: Cevaplar

```

 int retrieve();
 };

 void stack::store(int i)
 {
 list *item;

 item = new stack;
 if(!item) {
 cout << "Bellekte yer ayirma hatasi.\n";
 exit(1);
 }
 item->num = i;

 // yığın tipi işlemler için listenin sonuna koyma
 if(head) item->next = head;
 head = item;
 if(!tail) tail = head;
 }

 int stack::retrieve()
 {
 int i;
 list *p;

 if(!head) {
 cout << "List empty.\n";
 return 0;
 }

 // listenin başından kaldır
 i = head->num;
 p = head;
 head = head->next;
 delete p;

 return i;
 }

 // Sıralanmış liste oluştur
 class sorted : public list {
 public:
 void store(int i);
 int retrieve();
 };

 void sorted::store(int i)
 {
 list *item;
 list *p, *p2;
 item = new sorted;
 if(!item) {
 cout << "Bellekte yer ayirma hatasi.\n";
 exit(1);
 }
 item->num = i;

 // bir sonraki elemanın nereye konacağıını bul
 p = head;
 p2 = NULL;
 while(p) { // ortaya gider

```

```
if(p->num > 1) {
 item->next = p;
 if(p2) p2->next = item; // Birinci Eleman değil
 if(p==head) head = item; // yeni birinci eleman
 break;
}
p2 = p;
p = p->next;
}

if(!p) { // end'e devam eder
 if(tail) tail->next = item;
 tail = item;
 item->next = NULL;
}
if(!head) // ilk eleman
 head = item;
}

int sorted::retrieve()
{
 int li;
 list *p;

 if(!head) {
 cout << "List empty.\n";
 return 0;
 }

 // listenin başından al
 i = head->num;
 p = head;
 head = head->next;
 delete p;
}

return li;
}

int main()
{
 list *p;

 // kuyruğu göster
 queue q_obi;
 p = &q_obi; // kuyruğu işaret et

 p->store(1);
 p->store(2);
 p->store(3);

 cout << "Queue: ";
 cout << p->retrieve();
 cout << p->retrieve();
 cout << p->retrieve();

 cout << '\n';

 // yiğini göster
 stack s_obi;
 p = &s_obi; // yiğini işaret ed
```

```
p->store(1);
p->store(2);
p->store(3);

cout << "Stack: ";
cout << p->retrieve();
cout << p->retrieve();
cout << p->retrieve();

cout << '\n';

//sıralanmış listeyi göster
sorted sorted_ob;
p = &sorted_ob;

p->store(4);
p->store(1);
p->store(3);
p->store(9);
p->store(5);

cout << "Sorted: ";
cout << p->retrieve();
cout << p->retrieve();
cout << p->retrieve();
cout << p->retrieve();
cout << p->retrieve();

cout << '\n';

return 0;
}
```

## Pekiştirme Testi: Bölüm 10

1. Sanal fonksiyon其实是一个在基类中声明但在派生类中重新实现的成员函数。它在派生类中被重新实现，从而覆盖了基类中的实现。这被称为“重写”或“重载”。
2. 构造函数和析构函数不是虚函数，因此不能被派生类重写。
3. 虚函数在调用时，会根据调用对象的类型（动态类型）来选择正确的实现。如果一个对象是通过基类指针或引用调用虚函数的，编译器会在运行时根据对象的实际类型（动态类型）来决定调用哪个函数。
4. 安全的虚函数是指那些在基类中声明为虚的函数，它们在派生类中被重新实现。
5. 多态性是指一个类可以有多个不同的实现形式，这些实现形式可以通过不同的对象来访问。
6. 在多态性中，如果派生类重写了基类的虚函数，那么在派生类对象上调用该函数时，将调用派生类的实现，而不是基类的实现。
7. 是的，这是可能的。如果派生类重写了基类的虚函数，那么在派生类对象上调用该函数时，将调用派生类的实现，而不是基类的实现。

## Bütünleştirme Testi: Bölüm 10

```
1. // Sanal fonksiyonları tanıtan program
#include <iostream>
#include <cstdlib>
using namespace std;

class list {
public:
 list *head; // listenin başına işaretçi
 list *tail; // listenin sonuna işaretçi
 list *next; // Bir sonraki elemana işaretçi
 int num; // saklanacak değer

 list() { head = tail = next = NULL; }
 virtual void store(int i) = 0;
 virtual int retrieve() = 0;
};

// Sayısal türü liste oluştur.
class queue : public list {
public:
 void store(int i);
 int retrieve();
 queue operator=(queue q) { store(q.retrieve()); return *this; }
 int operator-(list &another) { return another.retrieve(); }
};

void queue::store(int i)
{
 list *item;
 item = new queue();
 if(item) {
 cout << "Belirtilen yer ayırma hatası.\n";
 exit(1);
 }
 item->num = i;

 // Listenin sonuna koyma
 if(tail) tail->next = item;
 tail = item;
 item->next = NULL;
 if(!head) head = tail;
}

int queue::retrieve()
{
 int i;
 list *p;

 if(!head) {
 cout << "List empty.\n";
 return 0;
 }

 // listenin başından çıkar
 i = head->num;
 p = head;
```

```
head = head->next;
delete p;
return i;
}

// Yiğin tipinde liste oluştur.
class stack : public list {
public:
 void store(int i);
 int retrieve();
 stack operator+(int i) { store(i); return *this; }
 int operator--(int unused) { return retrieve(); }
};

void stack::store(int i)
{
 list *item;

 item = new stack;
 if(!item) {
 cout << "Bellekte yer ayırma hatası.\n";
 exit(1);
 }
 item->num = i;

 // yiğin gibi çalışarak listenin başına koy
 if(head) item->next = head;
 head = item;
 if(!tail) tail = head;
}

int stack::retrieve()
{
 int i;
 list *p;

 if(!head) {
 cout << "List empty.\n";
 return 0;
 }

 // listenin başından çıkar
 i = head->num;
 p = head;
 head = head->next;
 delete p;
 return i;
}

int main()
{
 //kuyruğu göster
 queue q_ab;

 q_ab + 1;
 q_ab + 2;
 q_ab + 3;

 cout << "Queue: ";
 cout << q_ab--;
 cout << q_ab--;
 cout << q_ab--;
}
```

```

cout << '\n';

//yiğini göster
stack s_ob;

s_ob + 1;
s_ob + 2;
s_ob + 3;

cout << "Stack: ";
cout << s_ob--;
cout << s_ob--;
cout << s_ob--;

cout << '\n';

return 0;
}

```

2. Sanal fonksiyonlar aşırı yüklenmiş fonksiyonlardan farklıdır. Çünkü aşırı yüklenmiş fonksiyonlar ya parametre sayısı bakımından ya da parametre tipi bakımından farklı olmak *zorundadır*. Düzenlenmiş sanal fonksiyon, orijinal fonksiyonun tamamen aynı olan bir prototipi olmalıdır (yani, aynı tipte döndürme, aynı tip ve sayıda parametre).

## Gözden Geçirme Testi: Bölüm 11

1. Sanal fonksiyon, taban sınıf tarafından **virtual** olarak bildirilen ve türetilmiş sınıf tarafından düzenlenen bir fonksiyondur.
2. Saf sanal fonksiyon taban sınıfında içi boş olarak bildirilen bir fonksiyondur. Bu demektir ki bu fonksiyon türetilmiş sınıf tarafından düzenlenmelidir. En azından bir tane saf sanal fonksiyon içeren sınıfa soyut sınıf denir.
3. Unutulan kelimeler "virtual" ve "base"tir.
4. Eğer türetilmiş sınıf saf olmayan sanal fonksiyonu kullanılemese, türetilmiş sınıf taban sınıfında tanımlanan fonksiyonu kullanır.
5. Çalışma anı polimorfizmin en büyük avantajı esnek olmasıdır. Dezavantaj ise işlem hızını düşürmesidir.

### 11.1. Alistirmalar

```

2. #include <iostream>
using namespace std;

template <class X> X min(X a, X b)
{
 if(a<b) return a;
 else return b;
}

int main()

```

```

 cout << min(12.2, 2.0);
 cout << endl;
 cout << min(3, 4);
 cout << endl;
 cout << min('c', 'a');
 return 0;
}

3. #include <iostream>
#include <cstring>
using namespace std;

template <class X> int find(X object, X *list, int size)
{
 int i;
 for(i=0; i<size; i++)
 if(object == list[i]) return i;
 return -1;
}

int main()
{
 int a[] = {1, 2, 3, 4};
 char *c = "Bu bir denemedir";
 double d[] = {1.1, 2.2, 3.3};

 cout << find(3, a, 4);
 cout << endl;

 cout << find('a', c, (int) strlen(c));
 cout << endl;

 cout << find(0.0, d, 3);

 return 0;
}

```

4. Sosyal fonksiyonlar değerlidir çünkü çeşitli veri tiplerine uygulanabilen genel algoritmalar hazırlamanıza yardımcı olur. (Yani, algoritmanın özel sürümlerini hazırlamak sizin göreviniz değil) Sosyal fonksiyonlar C++ programcılığında bilinen "bir arabirim, çok metot" kavramını belirtmenize yardımcı olur.

## 11.2. Alıştırmalar

```

2. // Sosyal bir sıra oluştur
#include <iostream>
using namespace std;

#define SIZE 100

template <class Qtype> class q_type {
 Qtype queue[SIZE]; // kuyruğu tut
 int head, tail; // baş ve sonun indisleri
public:
 q_type() { head = tail = 0; }
 void q(Qtype num); // sakla

```

```
Qtype deq(); // çek
};

// kuyruğa değer koy
template <class Qtype> void q_type<Qtype>::q(Qtype num)
{
 if(tail+1==head || (tail+1==SIZE && !head)) {
 cout << "Queue is full.\n";
 return;
 }
 tail++;
 if(tail==SIZE) tail = 0; // çevrim
 queue[tail] = num;
}

// sıradan değeri çıkar.
template <class Qtype> Qtype q_type<Qtype>::deq()
{
 if(head == tail) {
 cout << "Queue is empty.\n";
 return 0; //veya diğer bir hata belirteci
 }
 head++;
 if(head==SIZE) head = 0; // çevrim
 return queue[head];
}

int main()
{
 q_type<int> q1;
 q_type<char> q2;
 int i;

 for(i=1; i<=10; i++) {
 q1.q(i);
 q2.q(i-1+'A');
 }

 for(i=1; i<=10; i++) {
 cout << "Sequence 1: " << q1.deq() << "\n";
 cout << "Sequence 2: " << q2.deq() << "\n";
 }

 return 0;
}

3. #include <iostream>
using namespace std;

template <class X> class input {
 X data;
public:
 input(char *s, X min, X max);
 // ...
};

template <class X>
input<X>::input(char *s, X min, X max)
{
 do {
 cout << s << ":";
```

```

 cin >> data;
 } while(data < min || data > max);
}

int main()
{
 input<int> i("enter int", 0, 10);
 input<char> c("enter char", 'A', 'Z');

 return 0;
}

```

## 11.3. Alıştırmalar

2. İşlem try bloğundan geçmeden throw çağrıılır.
3. Karakter hatası atılır fakat catch ifadesi sadece karakter işaretçisini yakalar. (Yani, karakter hmasını yakalayacak uygun bir catch ifadesi yoktur)
4. Eğer uygun catch ifadesi bulunmayan bir hata atılırsa, terminate() çağrılır ve a-normal program sonlanması meydana gelir.

## 11.4. Alıştırmalar

2. throw için uygun bir catch ifadesi yoktur.
3. Problemi çözmeyen bir yolu catch(int) işleyicisini oluşturmaktır. Diğer bir yol ise bütün hataları catch(...) ile yakalamaktır.
4. catch(...) bütün hataları yakalar.

```

5. #include <iostream>
#include <cstdlib>
using namespace std;

double divide(double a, double b)
{
 try {
 if(!b) throw(b);
 }

 catch(double) {
 cout << "Cannot divide by zero.\n";
 exit(1);
 }

 return a/b;
}

int main()
{
 cout << divide(10.0, 2.5) << endl;
 cout << divide(10.0, 0.0);

 return 0;
}

```

## 11.5 ALIŞTIRMALAR

1. Normalde, bellekte yer ayırma hatalı meydana geldiğinde new bir hata atar. new'in nothrow sürümü eğer yer ayrılmazsa boş bir işaretçi döndürür.

```
2. p = new(nothrow) int;
 if(!p) {
 cout << "Bellekte yer ayırma hatalı.\n";
 // ...
 }
 try {
 p = new int;
 } catch(bad_alloc ba) {
 cout << "Bellekte yer ayırma hatalı.\n";
 // ...
 }
```

## Pekiştirme Testi: Bölüm 11

```
1. #include <iostream>
#include <cstring>
using namespace std;
// Sosyal bir mod bulma fonksiyonu
template <class X> X mode(X *data, int size)
{
 register int t, w;
 X md, oldmd;
 int count, oldcount;
 oldmd = 0;
 oldcount = 0;
 for(t=0; t<size; t++) {
 md = data[t];
 count = 1;
 for(w = t+1; w < size; w++)
 if(md==data[w]) count++;
 if(count > oldcount) {
 oldmd = md;
 oldcount = count;
 }
 }
 return oldmd;
}
int main()
{
 int i[] = { 1, 2, 3, 4, 2, 3, 2, 2, 1, 5 };
 char *p = "Bu bir denemedir";

 cout << "mode of i: " << mode(i, 10) << endl;
 cout << "mode of p: " << mode(p, (int) strlen(p)) << endl;

 return 0;
}

2. #include <iostream>
using namespace std;

template <class X> X sum(X *data, int size)
{
```

## EK B: Cevaplar

```

int i;
X result = 0;

for(i=0; i<size; i++) result += data[i];

return result;
}

int main()
{
 int i[] = {1, 2, 3, 4};
 double d[] = {1.1, 2.2, 3.3, 4.4};

 cout << sum(i, 4) << endl;
 cout << sum(d, 4) << endl;

 return 0;
}

3. #include <iostream>
using namespace std;

// Sosyal bir bubble sort
template <class X> void bubble(X *data, int size)
{
 register int *a, *b;
 X t;

 for(a=i; a < size; a++)
 for(b=size-1; b >= a; b--)
 if(data[b-1] > data[b]) {
 t = data[b-1];
 data[b-1] = data[b];
 data[b] = t;
 }
}

int main()
{
 int i[] = {3, 2, 5, 4, 1, 8, 9, 7, 6, 10};
 double d[] = {1.2, 5.5, 2.2, 3.3, 4.4, 0.1, 6.6, 7.7, 8.8, 9.9};

 bubble(i, 10); // sort ints
 bubble(d, 4); // sort doubles
 for(j=0; j<10; j++) cout << i[j] << " ";
 cout << endl;

 for(j=0; j<4; j++) cout << d[j] << " ";
 cout << endl;

 return 0;
}

4. /* Bu fonksiyon iki değer tutan sosyal bir yığını gösteriyor */
#include <iostream>
using namespace std;

#define SIZE 10

// sosyal bir yığın sınıf oluştur

```

```
template <class StackType> class stack {
 StackType stck[SIZE][2]; // yiğini tut
 int tos; // yiğin üstünün indeksi

public:
 void init() { tos = 0; }
 void push(StackType ob, StackType ob2);
 StackType pop(StackType &ob2);
}

// nesneleri it
template <class StackType>
void stack<StackType>::push(StackType ob, StackType ob2)
{
 if(tos==SIZE) {
 cout << "Stack is full.\n";
 return;
 }
 stck[tos][0] = ob;
 stck[tos][1] = ob2;
 tos++;
}

// Nesneleri çek.
template <class StackType>
StackType stack<StackType>::pop(StackType &ob2)
{
 if(tos==0) {
 cout << "Stack is empty.\n";
 return 0; // Yiğin boşsa null döndürür
 }
 tos--;
 ob2 = stck[tos][1];
 return stck[tos][0];
}

int main()
{
 // Karakter yiğinlarını göster
 stack<char> s1, s2; // İki yiğin oluştur
 int i;
 char ch;

 // yiğinları hazırla
 s1.init();
 s2.init();

 s1.push('a', 'b');
 s2.push('x', 'z');
 s1.push('b', 'd');
 s2.push('y', 'e');
 s1.push('c', 'a');
 s2.push('z', 'x');

 for(i=0; i<3; i++) {
 cout << "Pop s1: " << s1.pop(ch);
 cout << ' ' << ch << "\n";
 }
 for(i=0; i<3; i++) {
 cout << "Pop s2: " << s2.pop(ch);
 cout << ' ' << ch << "\n";
 }
}
```

```

// çift yiğinları göster
stack<double> ds1, ds2; // İki yiğin oluştur
double d;

// yiğinları hazırla
ds1.init();
ds2.init();

ds1.push(1.1, 2.0);
ds2.push(2.2, 3.0);
ds1.push(3.3, 4.0);
ds2.push(4.4, 5.0);
ds1.push(5.5, 6.0);
ds2.push(6.6, 7.0);

for(i=0; i<3; i++) {
 cout << "Pop ds1: " << ds1.pop(d);
 cout << " " << d << "\n";
}
for(i=0; i<3; i++) {
 cout << "Pop ds2: " << ds2.pop(d);
 cout << " " << d << "\n";
}
return 0;
}

```

5. try, catch, ve throw'un genel formüləri aşağıdakılardır:

```

try {
 // try bloğu
 throw exp;
}
catch(type arg) {
 // ...
}

6. /* Bu fonksiyon hata işleme içeren sosyal bir yiğini gösteriyor. */
#include <iostream>
using namespace std;

#define SIZE 10

// Sosyal yiğin sınıfı oluştur
template <class StackType> class stack {
 StackType stck(SIZE); // yiğini tutar
 int tos; // yiğin üstünün indeksi

public:
 void init() { tos = 0; } // yiğini hazırla
 void push(StackType ch); // yiğine nesne it
 StackType pop(); // yiğindan nesne çek
};

// bir nesne it
template <class StackType>
void stack<StackType>::push(StackType ob)
{
 try {
 if(tos==SIZE) throw SIZE;
 }
}

```

```
 catch(int) {
 cout << "Stack is full.\n";
 return;
 }
 stck[tos] = ob;
 tos++;
}

// nesne çek
template <class StackType>
StackType stack<StackType>::pop()
{
 try {
 if(tos==0) throw 0;
 }
 catch(int) {
 cout << "Stack is empty.\n";
 return 0; // Yığın boşsa null döndürür
 }
 tos--;
 return stck[tos];
}

int main()
{
 // karakter yığınlarını göster
 stack<char> s1, s2; // İki yığın oluştur
 int i;
 // yığınları hazırla
 s1.init();
 s2.init();

 s1.push('a');
 s2.push('x');
 s1.push('b');
 s2.push('y');
 s1.push('c');
 s2.push('z');

 for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
 for(i=0; i<4; i++) cout << "Pop s2: " << s2.pop() << "\n";
 // çift yığınları göster
 stack<double> ds1, ds2; // İki yağıt oluştur

 // yığınları hazırla
 ds1.init();
 ds2.init();

 ds1.push(1.1);
 ds2.push(2.2);
 ds1.push(3.3);
 ds2.push(4.4);
 ds1.push(5.5);
 ds2.push(6.6);

 for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";
 for(i=0; i<4; i++) cout << "Pop ds2: " << ds2.pop() << "\n";

 return 0;
}
```

7. Eğer new, bellekte yer ayırma hatası meydana geldiğinde hata atarsa, hatanın bir yolla işleneceğinden emin olabilirsiniz. Hatta anormal program sonlanması bile... Bunun tersine, new'in boş işaretçi döndürmesiyle haber verilen yer ayırma başarısızlığı, bu ihtimal değerlendirilmemezse yeniden yaşanabilir.

## Gözden Geçirme Testi: Bölüm 12

1. C++'ta sosyal fonksiyon çeşitli tiplerde veri üzerine uygulanabilecek bir grup genel işlem tanımlar. Bu işlem template anahtar kelimesiyle belirtilir. Genel hali size göstereyim:

```
template <class Ttype> ret-type func-name(param-list)
{
 // ...
}
```

2. C++'da sosyal sınıf o sınıfta ilişkili tüm işlemleri tanımlar, fakat gerçek veri tipi o sınıftan bir nesne oluşturulduğunda, parametre olarak belirtilir. Genel hali aşağıdaki gibidir:

```
template <class Ttype> class class-name
{
 // ...
}
```

3. #include <iostream>  
 using namespace std;  
  
 // a'yi b'ye dönür  
 template <class X> X qexp(X a, X b)  
 {  
 X i, result=1;  
  
 for(i=0; i<b; i++) result \*= a;  
 return result;
 }

 int main()
 {
 cout << qexp(2, 3) << endl;
 cout << qexp(10.0, 2.0);

 return 0;
 }

4. #include <iostream>  
 #include <fstream>  
 using namespace std;  
  
 template <class CoordType> class coord {
 CoordType x, y;
public:
 coord(CoordType i, CoordType j) { x = i; y = j; }
 void show() { cout << x << ", " << y << endl; }
 };

```

int main()
{
 coord<int> o1(1, 2), o2(3, 4);

 o1.show();
 o2.show();

 coord<double> o3(0.0, 0.23), o4(10.19, 3.098);

 o3.show();
 o4.show();

 return 0;
}

```

5. **try, catch, ve throw** birlikte şöyle çalışır: Hata için gözlemek istediğiniz tüm ifadeleri **try** bloğu içerisinde yerleştirin. Eğer bir hata meydana gelirse, **throw** kullanarak hatayı atın ve uygun **catch** ifadesi ile onu yakalayın.
6. Hayır
7. **terminate()**, atılan bir hatanın uygun bir **catch** ifadesi bulunmadığında çağrılır. **unexpected()** ise fonksiyonun throws yapısında belirtilmemiş bir hatanın fonksiyon dışına atılmasıyla çağrılır.
8. **catch(...)**

## 12.1. Alıştırmalar

1. RTTI C++ için gereklidir çünkü derleme zamanında, sınıf taban işaretçisi tarafından işaret edilen veya taban sınıf referansı tarafından referans gösterilen bir nesnenin tipini her zaman bilmek mümkün değildir.
2. **BaseClass** artık polimorfik sınıf olmadığından üretilen çıkış.

```

typeid of i is int
p is pointing to an object of type class BaseClass
p is pointing to an object of type class BaseClass
p is pointing to an object of type class BaseClass

```

3. Evet.
4. `if(typeid(*p) == typeid(D2)) ...`
5. Yanlış. Aynı şablon sınıf kullanıldığından her sürüm tarafından kullanılan verinin tipi farklıdır.

## 12.2. Alıştırmalar

1. **dynamic\_cast** operatörü polimorfik tip dönüşümünün geçerliliğini belirler.
2. `#include <iostream>`  
`#include <typeinfo>`

## EK B: Cevaplar

```

using namespace std;

class B {
 virtual void f() {}
};

class D1: public B {
 void f() {}
};

class D2: public B {
 void f() {}
};

int main()
{
 B *p;
 D2 obj;

 p = dynamic_cast<D2 *> (&obj);

 if(p) cout << "Cast OK";
 else cout << "Cast Fails";

 return 0;
}

3. int main()
{
 int i;
 Shape *p;

 for(i=0; i<10; i++) {
 p = generator(); // bir sonraki nesneyi al

 cout << typeid(*p).name() << endl;

 // draw object only if it is not a NullShape
 if(!dynamic_cast<NullShape *> (p))
 p->example();
 }

 return 0;
}

```

4. Hayır. **Bp** ve **Dp** değişik tiplerdeki nesnelere olan işaretçileridir.

### 12.3. Alıştırmalar

- Yeni dönüşüm operatörü dönüşümleri gerçekleştirmek için daha güvenli, açık yollar sağlar.

```

2. #include <iostream>
using namespace std;

void f(const double &i)
{
 double &v = const_cast<double &> (i);

```

```

 v = 100.0;
}

int main()
{
 double x = 98.6;

 cout << x << endl;
 f(x);
 cout << x << endl;

 return 0;
}

```

3. **const\_cast**, **const** belirleyicisini düzenlediğinden nesnele beklenmeyen değişikliklerin yapılmasına imkan verir.

## Pekiştirme Testi: Bölüm 12

1. **typeid** operatörü tip bilgisini taşıyan **type\_info** sınıfından bir nesneye referans döndürür.
2. **typeid**'yi kullanmak için **<typeinfo>**'yu programınıza dahil etmeniz gereklidir.
3. yeni dönüşüm operatörleri aşağıda gösterilmiştir.

| Operatör                | Amaç                                 |
|-------------------------|--------------------------------------|
| <b>dynamic_cast</b>     | Polimorfik dönüşüm sağlar            |
| <b>reinterpret_cast</b> | Bir tip işaretçiyi diğerine çevirir. |
| <b>static_cast</b>      | "normal" dönüşüm gerçekleştirir.     |
| <b>const_cast</b>       | <b>const</b> -ness'ıan dönüştürür.   |

```

4. #include <iostream>
#include <typeinfo>
using namespace std;

class A {
 virtual void f() {}
};

class B : public A {
};

class C : public B {
};

int main()
{
 A *p, *a_obj;
 B b_obj;
 C c_obj;
 int i;

 cout << "Enter 0 for A objects, ";

```

```

cout << "1 for B objects or ";
cout << "2 for C objects.\n";
cin >> i;

if(i==1) p = &b_obj;
else if(i==2) p = &c_obj;
else p = &a_obj;

if(typeid(*p) == typeid(A))
 cout << "A object";
if(typeid(*p) == typeid(B))
 cout << "B object";
if(typeid(*p) == typeid(C))
 cout << "C object";

return 0;
}

```

5. typeid'nin polimorfik sınıf dönüşümlerinin geçerliliğini kontrol etmek için kullanılması gereken durumlarda **dynamic\_cast** operatörü onun yerini alabilir.
6. typeid operatörü **type\_info** nesnesine referans döndürür.

## Bütünleştirme Testi: Bölüm 12

1. Bütün bellek ayırma hatalarını denetlemek için hata işleme kullanan **generator()** sınıfının yeni sürümü:

```

// Tüm ayırma başarısızlıklarını denetlemek için hata denetim kullanma
Shape *generator()
{
 try {
 switch(rand() % 4) {
 case 0:
 return new Line;
 case 1:
 return new Rectangle;
 case 2:
 return new Triangle;
 case 3:
 return new NullShape;
 }
 } catch (bad_alloc ba) {
 return NULL;
 }
 return NULL;
}

```

2. **generator( )**'in **newnothrow**'u kullanan yeni düzenlemesi.

```

// newnothrow kullan
Shape *generator()
{
 Shape *temp;
 switch(rand() % 4) {
 case 0:

```

```

 temp = new(nothrow) Line;
 break;
 case 1:
 temp = new(nothrow) Rectangle;
 break;
 case 2:
 temp = new(nothrow) Triangle;
 break;
 case 3:
 temp = new(nothrow) NullShape;
 }
 if(temp) return temp;
 else return NULL;
}

```

## Gözden Geçirme Testi: Bölüm 13

1. C tarzı dönüşüme ek olarak

| Operatör         | Amaç                                     |
|------------------|------------------------------------------|
| dynamic_cast     | Polimorfik dönüşüm sağlar.               |
| reinterpret_cast | Bir tip işaretçisi diğerine çevirebilir. |
| static_cast      | "normal" dönüşüm gerçekleştirir.         |
| const_cast       | const-nesneler dönüştürür.               |

2. type\_info ve typeid hakkında bilgi depolayan bir sınıfı. type\_info nesnesine reference typeid operatörünü tarafından döndürülür.
3. typeid
4.

```

if(typeid(*derived) == typeid(*p))
 cout << "p points to derived object\n";
else
 cout << "p points to a Base object\n";

```
5. "derived." unsurları kelimedir.
6. Hazır

### 13.1. Alistirmalar

1. /\* Bağlantıları "using namespace std" ifadesi kullanmadan ls'ye dönüştürün \*/

```

#include <iostream>
#include <fstream>

int main(int argc, char *argv[])
{
 if(argc!=3) {
 std::cout << "Usage: CONVERT <input> <output>\n";
 return 1;
 }

 std::ifstream fin(argv[1]); // Giriş dosyasını aç
 std::ofstream fout(argv[2]); // Çıkış dosyasını aç

```

```

if(!fout) {
 std::cout << "Cannot open output file.\n";
 return 1;
}
if(!fin) {
 std::cout << "Cannot open input file.\n";
 return 1;
}

char ch;

fin.unsetf(std::ios::skipws); // boşlukları atlama
while(!fin.eof()) {
 fin >> ch;
 if(ch==' ') ch = '|';
 if(!fin.eof()) fout << ch;
}

fin.close();
fout.close();

return 0;
}

```

2. İsimlendirilmemiş bildirim bölgeleri, içerisinde tanımlanmış değişkenlerin kavramını dosyasına kısıtlar.

3. **using'in bu formu**

`using name::member ;`

sadece belirtilen üyesi şu anki bildirim bölgesinde alır. Şu form ise

`using namespace name;`

tüm bildirim bölgesini alana çeker.

4. `cin` ve `cout` akımlarını içeren Tüm C++ standart kütüphanesi, std bildirim bölgesi içerisinde bildirilmiştir. Geçerlilik açısından bir çok program std bildirim bölgelerini alana gekmişlerdir. Bu olay, standart kütüphane isimlerine std:: belirteci kullanılmadan, doğrudan erişilebilmesi sağlar. Birçok program için bir alternatif, bütün referansları standart kütüphaneye std:: ile bağlamaktır. Diğer bir alternatif ise sadece std::cin veya std::cout içeri using içindeki oluputrmakta.
5. Kütüphane kodunu kendi bildirim bölge sine kaynakla etm çökmenin ihtimalini azaltmış olursunuz.

## 13.2. Alıştırmalar

- // tarter tipini tamsayiya çevir
 

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

class strtype {
```

```
char str[80];
int len;
public:
 strtype(char *s) { strcpy(str, s); len = strlen(s); }
 operator char *() { return str; }
 operator int() { return len; }
};

int main()
{
 strtype a("Conversion fonksiyons are convenient.");
 char *p;
 int l;

 l = a; // katarın uzunluğu olan a'ı tamayaçılığa çevir -
 p = a; // katarı işaretçi olan a char **'a çevir -
 cout << "The string:\n";
 cout << p << "\nis " << l << " chars long.\n";

 return 0;
}

2. #include <iostream>
using namespace std;

int p(int base, int exp);

class pwr {
 int base;
 int exp;
public:
 pwr(int b, int e) { base = b; exp = e; }
 operator int() { return p(base, exp); }
};

// exp üssüne taban döndür
int p(int base, int exp)
{
 int temp;
 for(temp=1; exp; exp--) temp = temp * base;
 return temp;
}

int main()
{
 pwr o1(2, 3), o2(3, 3);
 int result;

 result = o1;
 cout << result << '\n';

 result = o2;
 cout << result << '\n';

 // doğrudan cout ifadesi içerisinde kullanılır
 cout << o1+100 << '\n';

 return 0;
}
```

### 13.3. Alıştırmalar

```
1. // Çıkış üzerinde dolaşan paylaşılmış kaynak Orneği
#include <iostream>
#include <cstring>
using namespace std;
class output {
 static char outbuf[255]; // paylaşılan kaynak
 static int inuse; // tampon aktif ejer 0; kullanılmadysa
 static int oindex; // outbuf'un indeksi
 char str[80];
 int i; // str'deki bir sonraki karakterin indeksi
 int who; // nesneyi tanır, > 0 olmalı
public:
 output(int w, char *s) { strcpy(str, s); i = 0; who = w; }

 /* Bu fonksiyon Tampon için bekliyorsa -1,
 Çıkışı tamamladı ise 0,
 Tamponu kullanıyorsa who'yu döndürür.
 */
 int putbuf()
 {
 if(!str[i]) { // çıkış tamam
 inuse = 0; // tamponu bırak
 return 0; // sinyal kesilmesi
 }
 if(!inuse) inuse = who; // tamponu al
 if(inuse != who) {
 cout << "Process " << who << " Currently blocked\n";
 return -1; // başkası tarafından kullanımda
 }
 if(str[i]) { // çıkışa verilecek karakterler halen mevcut
 outbuf[oindex] = str[i];
 cout << "Process " << who << " sending char\n";
 i++; oindex++;
 outbuf[oindex] = '\0'; // her zaman boş'a sonlandırılmış olarak sakla
 return 1;
 }
 return 0;
 }
 void show() { cout << outbuf << '\n'; }
};
char output::outbuf[255]; // Bu paylaşılan kanaktır
int output::inuse = 0; // tampon kullanılabilir 0; kullanılmadysa
int output::oindex = 0; // outbuf'un indeksi
int main()
{
 output o1(1, "Bu bir denemedir"), o2(2, " of statics");
 while(o1.putbuf() | o2.putbuf()); // çıkışa karakterler sürüklenebilir
 o1.show();
 return 0;
}
2.
#include <iostream>
#include <new>
using namespace std;
class test {
 static int count;
public:
 test() { count++; }
```

```
-test(); count--; }
int getcount() { return count; }
}
int test::count = 0;
int main()
{
 test o1, o2, o3;
 cout << o1.getcount() << " objects in existence\n";
 test *p;
 /* eski ve yeni tip hata işleme ile
 bellekte yer ayırma hataları için tetikte */

 try {
 p = new test; // nesne için yer ayılır
 if(!p) { // eski tip
 cout << "Bellekte yer ayırma hatası\n";
 return 1;
 }

 } catch(bad_alloc ba) { // yeni tip
 cout << "Bellekte yer ayırma hatası\n";
 return 1;
 }
 cout << o1.getcount();
 cout << " objects in existence after allocation\n";
 // nesneyi sil
 delete p;

 cout << o1.getcount();
 cout << " objects in existence after deletion\n";
 return 0;
}
```

## 13.4. Alıştırmalar

1. Programı düzeltmek için basit şekilde **current**'ı değiştirebilir yapın. Böylece **const** üye fonksiyonu olan **counting()** tarafından değiştirilebilsin. Çözümü aşağıda veriyorum:

```
// Program düzeltildi
#include <iostream>
using namespace std;

class CountDown {
 int incr;
 int target;
 mutable int current; // current'ı değiştirebilir yap
public:
 CountDown(int delay, int i=1) {
 target = delay;
 incr = i;
 current = 0;
 }
```

```
bool counting() const {
 current += incr;
 if(current >= target) {
 cout << "\a";
 return false;
 }
 cout << current << " ";
 return true;
}

int main()
{
 CountDown ob(100, 2);
 while(ob.counting());
 return 0;
}
```

2. Hayır. Eğer **const** üye fonksiyonunun **const**-olamayan üye fonksiyonunu çağrıması mümkün olsaydı, **const** -olmayan fonksiyon uyarılan nesnenin değiştirilmesinde kullanılabilirdi.

## 13.5. Alıştırmalar

1. Evet.
2. Evet çünkü C++ int'ten double'a kendiliğinden dönüşüm tanımlar.
3. Kapalı constructor dönüşümleriyle ilgili potansiyel bir problem de, öyle bir dönüşüm meydana geldiğini unutabilmenizdir. Örneğin, atama ifadesi içerisinde meydana gelen kapalı bir dönüşü asını yüklenmiş bir atama operatörüne çok benzer. Ancak, işlevleri aynı olmak zorunda değildir. Siz, başkalarının kullanacağı bir sınıf oluşturuyorken, sınıflarınızın kullanıcıları tarafından yanlış anlaşılmayı engellemek maksadıyla kapalı constructor dönüşümlerini korumak gerekebilir.

## 13.7. Alıştırmalar

1. /\* bu sürüm buf'a yazılan char'ları görüntüler \*/
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
 char buf[255];

 ostrstream ostr(buf, sizeof buf);

 ostr << "Array-based I/O uses streams just like ";
 ostr << "'normal' I/O\n" << 100;
 ostr << ' ' << 123.23 << '\n';

```

// manipülatörleri de kullanabilirsiniz
ostr << hex << 100 << ' ';
// veya bayrakları biçimlendir
ostr.setf(ios::scientific);

ostr << dec << 123.23;
ostr << endl << ends; // tamponun boş ile sonlandırıldığından emin olun

// sonuç katarını gösterin
cout << buf;

cout << ostr.pcount();

return 0;
}

2. /* dizin içeriğini diğerine kopyalamak için dizi tabanlı I/O kullan */
#include <iostream>
#include <strstream>
using namespace std;

char inbuf[] = "Bu bir denemedir of C++ array-based I/O";
char outbuf[255];

int main()
{
 istream istr(inbuf);
 ostream ostr(outbuf, sizeof outbuf);

 char ch;

 while(!istr.eof()) {
 istr.get(ch);
 if(!istr.eof()) ostr.put(ch);
 }

 ostr.put('\0'); // boş sonlanma

 cout << "Input: " << inbuf << '\n';
 cout << "Output: " << outbuf << '\n';

 return 0;
}

3. // Katarı float'a dönüştür
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
 float f;
 char s[] = "1234.564"; // float katar olarak gösterilmistir
 istream istr(s);

 // iç gösterimi kolay yolla dönüştür
 istr >> f;

 cout << "Converted form: " << f << '\n';
 return 0;
}

```

## Pekiştirme Testi: Bölüm 13

- Normal üye değişkeninden farklı, her nesnenin kendi kopyasına sahip olduğu durumda, **static** üye değişkeninin sadece bir kopyası bulunur ve o sınıfın bütün nesneleri tarafından paylaşıılır.
- Dizi tabanlı I/O kullanmak için `<sstream>`'i dahil edin.
- Hayır.
- `extern "C" int counter();`
- Bir dönüşüm fonksiyonu basitçe bir nesneyi diğer tipten uyumlu bir değere dönüştürür. Dönüşüm fonksiyonları genelde nesneleri iç veri tipleriyle uyumlu değerlere dönüştürmek için kullanılır.
- explicit** anahtar kelimesi sadece constructor'lara uygulanır. Kapalı constructor dönüşümlerini engeller.
- const** üye fonksiyonu onu uyarlan nesneyi değiştiremez.
- namespace** anahtar kelimesi ile bildirilen Bildirim bölgesi, bildirim yapılabılır bir bölge tanımlar.
- mutable** anahtar kelimesi, **const** üye fonksiyonuna, kendi sınıfına üye olan veriyi değiştirmeye imkanı verir.

## Bütünleştirme Testi: Bölüm 13

- Evet. Kapalı dönüşümün, constructor parametresinin tipi igeri azan yüklenmiş bir operatörün gösterdiği işlevle, aynı işlevi göstermesi durumunda, atama operatörünü aşırı yüklemeye gerek yoktur.
- Evet
- Yeni kütüphaneler kendi bildirim bölgeleri içerisinde barındırlar. Böylece isim ve diğer kod çakışmaları önlenmiş olur. Hatta bu özellikle yeni kütüphanelerle güncelltilmiş eski kodlara da uygulanabilir.

## Gözden Geçirme Testi: Bölüm 14

- Bildirim bölgeleri C++'a, isim çakışmalarını önlemek amacıyla değişken isimlerini yerelleştirmek için eklenmiştir. Üçüncü parti sunf kütüphanelerinin çoğalmasıyla isim çakışmaları ciddi bir sorun haline gelmiştir.
- Bir üye fonksiyonu **const** olarak belirtmek için, fonksiyon parametre listesini **const** anahtar kelimesi ile takip edin. İşte size bir örnek:

```
int f(int a) const;
```

3. Yanlış, **mutable**, bir üye değişkenin **const** üye fonksiyonu ile değiştirilebilmesini sağlar.
4. 

```
class X {
 int a, b;
public:
 X(int i, int j) { a = i, b = j; }
 operator int() { return a+b; }
};
```
5. Doğru
6. Hayır, **explicit** ifadesi **int**'den Demo'ya kendiliğinden dönüşümü engeller.

## 14.1. Alıştırmalar

1. Bir container diğer nesneleri taşıyan bir nesnedir. Algoritma bir container'ın içindelerini etkileyen bir rutindir. Bir iterator işaretçiyeye benzer.
2. İkili ve birdü.
3. iterator'ların beş türü vardır: rasgele erişim, çift yönlü, ileri, giriş, çıkış

## 14.3. Alıştırmalar

2. Vektörelde saklanan her nesne geçerli yapılandırıcıyı (constructor) sağlamalıdır.
3. 

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

class Coord {
public:
 int x, y;
 Coord() : x = y = 0; }
 Coord(int a, int b) : x = a, y = b; }

bool operator<(Coord a, Coord b)
{
 return (a.x+a.y) < (b.x+b.y);
}

bool operator==(Coord a, Coord b)
{
 return (a.x+a.y) == (b.x+b.y); }

int main()
{
 vector<Coord> v;
 int i;

 for(i=0; i<10; i++)
 v.push_back(Coord(i, i));
```

```

 for(i=0; i<v.size(); i++)
 cout << v[i].x << "," << v[i].y << " ";
 cout << endl;

 for(i=0; i<v.size(); i++)
 v[i].x = v[i].x * 2;

 for(i=0; i<v.size(); i++)
 cout << v[i].x << "," << v[i].y << " ";

 return 0;
}

```

## 14.4. Alıştırmalar

2. // Temelleri listeleyin

```

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

int main()
{
 list<char> lst; // boş liste oluştur
 int i;

 for(i=0; i<10; i++) lst.push_back('A'+i);

 cout << "Size = " << lst.size() << endl;
 list<char>::iterator p;

 cout << "Contents: ";
 for(i=0; i<lst.size(); i++) {
 p = lst.begin();
 cout << *p;
 lst.pop_front();
 lst.push_back(*p); // elamanı listenin sonuna koy
 }

 cout << endl;
 if(!lst.empty())
 cout << "List is not empty.\n";

 return 0;
}

```

Bu program aşağıdaki çıkışı verir:

```

Size = 10
Contents: ABCDEFGHIJ
List is not empty.

```

Bu yaklaşımında, elemanlar önden çıkarılmış fakat arkaya konmuştur. Böylelikle liste boşaltılmamıştır. Listeyi görüntüleyen döngü, size( )'ın elde ettiği listenin uzunluğunu ile kontrol edilmektedir.

3. // iki proje listesini birleştirin

```

#include <iostream>

```

```
#include <list>
#include <cstring>
using namespace std;

class Project {
public:
 char name[40];
 int days_to_completion;
 Project() {
 strcpy(name, "");
 days_to_completion = 0;
 }

 Project(char *n, int d) {
 strcpy(name, n);
 days_to_completion = d;
 }

 void add_days(int i) {
 days_to_completion += i;
 }

 void sub_days(int i) {
 days_to_completion -= i;
 }

 bool completed() { return !days_to_completion; }

 void report() {
 cout << name << ": ";
 cout << days_to_completion;
 cout << " days left.\n";
 }
};

bool operator<(const Project &a, const Project &b)
{
 return a.days_to_completion < b.days_to_completion;
}

bool operator>(const Project &a, const Project &b)
{
 return a.days_to_completion > b.days_to_completion;
}

bool operator==(const Project &a, const Project &b)
{
 return a.days_to_completion == b.days_to_completion;
}

bool operator!=(const Project &a, const Project &b)
{
 return a.days_to_completion != b.days_to_completion;
}

int main()
{
 list<Project> proj;
 list<Project> proj2;

 proj.push_back(Project("Compiler", 35));
```

```

proj.push_back(Project("Spreadsheet", 190));
proj.push_back(Project("STL Implementation", 1000));

proj2.push_back(Project("Database", 780));
proj2.push_back(Project("Mail Merge", 50));
proj2.push_back(Project("COM Objects", 300));

proj.sort();
proj2.sort();

proj.merge(proj2); //projeleri birleştirir

list<Project>::iterator p = proj.begin();
/* display projects */
while(p != proj.end()) {
 p->report();
 p++;
}

return 0;
}

```

## 14.5. Alıştırmalar

2. // İsimlerin ve telefon numaralarının haritası

```

#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class name {
 char str[20];
public:
 name() { strcpy(str, ""); }
 name(char *s) { strcpy(str, s); }
 char *get() { return str; }
};

// isim nesnelerine bağlı olarak data sz tanımınıza
bool operator<(name a, name b)
{
 return strcmp(a.get(), b.get()) < 0;
}

class phonenum {
 char str[20];
public:
 phonenum() { strcpy(str, ""); }
 phonenum(char *s) { strcpy(str, s); }
 char *get() { return str; }
};

int main()
{
 map<name, phonenum> m;

 // put names and phone numbers into map
 m.insert(pair<name, phonenum>(name("Joe"), phonenum("555-4323")));
 m.insert(pair<name, phonenum>(name("Tom"), phonenum("555-9784")));
}

```

```

m.insert(pair<name, phonenum>(name("Larry"), phonenum("212 555-9372")));
m.insert(pair<name, phonenum>(name("Tod"), phonenum("01 11 232-4232")));

// bir isim ver, telefon numarasını bul
char str[80];
cout << "Enter name: ";
cin >> str;

map<name, phonenum>::iterator p;

p = m.find(name(str));
if(p != m.end())
 cout << "Phone Number: " << p->second.get();
else
 cout << "Name not in map.\n";

return 0;
}

```

3. Evet.

## 14.6. Alıştırmalar

- // Sırt algoritması kullanarak vektör sıralama

```

#include <iostream>
#include <vector>
#include <cstdlib>
#include <algorithm>
using namespace std;

int main()
{
 vector<char> v;
 int i;

 // rastgele karakterlerden oluşan vector oluştur
 for(i=0; i<10; i++)
 v.push_back('A'+(rand()%26));

 cout << "Original contents: ";
 for(i=0; i<v.size(); i++)
 cout << v[i] << " ";

 cout << endl << endl;

 // vektörü sırala
 sort(v.begin(), v.end());

 cout << "Sorted contents: ";
 for(i=0; i<v.size(); i++)
 cout << v[i] << " ";

 return 0;
}

```
- // İki listeyi birleştirme algoritması: kullanarak birleştir.

```

#include <iostream>
#include <list>
#include <algorithm>

```

```

using namespace std;

int main()
{
 list<char> lst1, lst2, lst3(20);
 int i;

 for(i=0; i<10; i+=2) lst1.push_back('A'+i);
 for(i=1; i<11; i+=2) lst2.push_back('A'+i);

 cout << "Contents of lst1: ";
 list<char>::iterator p = lst1.begin();
 while(p != lst1.end()) {
 cout << *p;
 p++;
 }
 cout << endl << endl;

 cout << "Contents of lst2: ";
 p = lst2.begin();
 while(p != lst2.end()) {
 cout << *p;
 p++;
 }
 cout << endl << endl;

 // Sıradaki üç listenin birleşimi
 merge(lst1.begin(), lst1.end(), lst2.begin(), lst2.end(), lst3.begin());

 cout << "Merged List: ";
 p = lst3.begin();
 while(p != lst3.end()) {
 cout << *p;
 p++;
 }

 return 0;
}

```

## 14.7. Alıştırmalar

1. 

```

#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
 list<string> str;

 str.push_back(string("one"));
 str.push_back(string("two"));
 str.push_back(string("three"));
 str.push_back(string("four"));
 str.push_back(string("five"));
 str.push_back(string("six"));
 str.push_back(string("seven"));
 str.push_back(string("eight"));
 str.push_back(string("nine"));
}
```

```
str.push_back(string("ten"));

str.sort(); // vektörü sırala

list<string>::iterator p = str.begin();
while(p != str.end()) {
 cout << *p << " ";
 p++;
}
return 0;
}

2. #include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
 string str;

 cout << "Enter a string: ";
 cin >> str;

 int i = count(str.begin(), str.end(), 'e');

 cout << i << " of them are e.\n";

 return 0;
}

3. #include <iostream>
#include <string>
#include <algorithm>
#include <cctype>
using namespace std;

int main()
{
 string str;

 cout << "Enter a string: ";
 cin >> str;

 int i = count_if(str.begin(), str.end(), islower);

 cout << i << " of them are lowercase.\n";

 return 0;
}

4. Katar basic_string'in özelleştirilmiş halidir.
```

## Pekiştirme Testi: Bölüm 14

1. STL çeşitli veri yapıları ve algoritmaların açılmış (debugged) sürümlerini sağlar. STL sınıfları şablon haline getirildiğinden her türü ver ile kullanılabilir.

2. Bir container diğer nesneleri tutan nesnedir. Iterator işaretçilere benzer. Bir algoritma container'ların içeriğine etki eder.
- 3.
- ```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main()
{
    vector<int> v(10);
    list<int> lst;
    int i;

    for(i=0; i<10; i++) v[i] = i;

    for(i=0; i<10; i++)
        if(!(v[i]&2)) lst.push_back(v[i]);

    list<int>::iterator p = lst.begin();
    while(p != lst.end()) {
        cout << *p << " ";
        p++;
    }

    return 0;
}
```
4. string tipi katarların operatörler ile manipüle edilmesine imkan verir. Ancak, string nesneleri ile çalışmak boy ile sonlandırılmış konularla çalışmaktan kader verimli değildir.
5. Bir predicate (boolean), true veya false döndüren bir funkciodur.