

Bu Notlar Kaan Aslan Tarafından C ve Sistem Programcıları Derneđi'nde Verilen C++Uygulamaları Dersinde Tutulmuştur. Notlar Üzerinde Herhangi Bir Düzeltme Yapılmamıştır.

Notları Tutan Deniz Kürümođlu'na Teşekkür Ederiz.

Sınıfların Dosya Düzenlemesi

Normal olarak bir sınıf iki dosya biçiminde düzenlenir. Sınıfın ismi X olmak üzere X.h ve X.cpp dosyaları oluşturulur. X.h dosyası içerisine nesne belirtmeyen yani yer kaplamayan öğeler yerleştirilir. Örneğin;

- Sınıf bildirimi
- Sınıf ile ilgili çeşitli sembolik sabitler ve global const değişkenler
- inline fonksiyon tanımlamaları
- Sınıf ile ilgili arkadaş fonksiyonların prototipleri
- Diğer bildirimler

X.cpp dosyasına ise üye fonksiyon tanımlamaları yerleştirilebilir.

Ayrıca tüm dosyalara düzenli bir biçimde başlık bölümü eklemek faydalıdır. Tipik bir başlık bölümü şöyle olabilir.

```
/*-----  
File Name: X.cpp  
Author: Deniz Kürümoğlu  
Last Update: 20/06/2004  
  
Açıklama  
Copyright Bilgisi  
-----*/
```

Başlık dosyalarına kesinlikle include koruması yerleştirilmelidir. include korumasının tipik genel biçimi şöyledir:

```
#ifndef _X_H_  
#define _X_H_  
....  
#endif
```

Buradaki sembolik sabit ismi herhangi bir biçimde uydurulmuş bir isim olabilir. Şüphesiz bu ismin dosya isminden hareketle oluşturulması uygundur.

Pekçok derleyici sisteminin tümleşik çevresi (IDE) sınıf oluştururken .h ve .cpp dosyalarını kendisi yaratmaktadır.

Visual C++ derleyici sisteminde yeni bir sınıfın eklenmesi için otomatik dosya oluşturmak isteniyorsa menüden "Insert/New Class" seçilir. Çıkan diyalog penceresinde sınıfın ismi girilir. Sınıf başka bir sınıftan türetilcekse taban sınıfın ismi de belirtilir. Derleyici sistemi şu işlemleri yapar:

- Sınıfın ismi X olmak üzere X.h ve X.cpp dosyalarını oluşturur.
- X.h dosyası içerisinde include koruması uygular.

- Belirtilen isimde yalnızca içi boş bir biçimde başlangıç ve bitiş fonksiyonları bulunan bir sınıf oluşturur.
- X.cpp içerisinde X.h dosyasını include eder.
- X.cpp dosyasını proje dosyasına ekler.

Böylece programcının oluşturulan sınıfı kullanmak için tek yapacağı şey X.h dosyasını include etmektir.

Bazen birkaç sınıf için bir arada tek bir başlık dosyası ve .cpp dosyası oluşturulabilir.

Büyük projelerde ön işlemci zamanını azaltmak için ek include koruması uygulanabilir.

Bir dosya içerisinde başka bir dosyada bildirilmiş olan isimlere gereksinim duyulabilir. Örneğin, B sınıfının veri elemanı A sınıfı türünden bir sınıf nesnesi olsun. Biz her sınıfı ayrı iki dosya olarak düzenlediğimize göre B sınıfı derlenirken derleyicinin A sınıfının bildirimini görmesi gerekir. Programcının başlık dosyalarını kendi kendine yetebilir hale getirmesi gerekir. Genel olarak .h dosyası içerisindeki bildirimlerde kullanılan isimlerin bulunduğu başlık dosyaları yine .h dosyası içerisinde include edilmelidir. Yani örneğin bizde B.h dosyası içerisinde A.h dosyasını include etmeliyiz.

B.H

```
#ifndef _B_H_
#define _B_H_
#include "a.h"
class B{
    //...
private:
    A m_a;
};
#endif
```

Büyük projelerde çeşitli başlık dosyaları ve kaynak dosyalar farklı farklı dizinlerde olabilir. C ön işlemcileri dosya ismi "" ile belirtildiğinde onu önce bulunulan dizinde sonra da derleyicinin kendi belirlediği dizinde aramaktadır. Biz ön işlemcinin başka dizinlere de bakmasını isteyebiliriz.

Anahtar Notlar:

Gerek C standartlarında gerekse C++ standartlarında "" yada <> ile include etme işlemi sırasında ön işlemcinin hangi dizinlere bakacağı belirtilmemiştir. Standartlarda derleyicilerin kendi istedikleri dizinlere bakabileceği belirtilmiştir. Bugün kullanılan derleyici sistemlerinde ön işlemcinin bakacağı include dizinleri derleyici genelinde yada proje genelinde değiştirilebilmektedir. Örneğin, Visual C++ derleyici sisteminde derleyici genelinde bakılacak include dizinlerini değiştirmek yada ekleme yapmak için "Tools/Options/Directories" menüsü kullanılır. Benzer biçimde proje genelinde dizin ekleme işlemi için "Project/Settings/Category:Preprocessor/Additional include directories" kullanılabilir.

Şüphesiz dizin belirleme işlemi "" yada <> içerisinde tüm yol ifadesinin yazılmasıyla da sağlanabilir, fakat bu yöntem tercih edilmemelidir.

Ek include Korumasının Uygulanması:

Büyük projelerde ek include korumaları derleme zamanını kısaltabilmektedir. a.h, b.h ve c.h dosyaları kendi içerisinde x.h dosyasını include etmiş olsun. Bizde bu üç dosyayı include etmiş olalım. a.h dosyası içerisinde x.h dosyası derleme modülünde görülecek ve derleme modülüne verilecektir. Ön işlemci b.h ve c.h dosyalarında x.h dosyasının include edildiğini gördüğünde bu dosyanın önce içeriğini açacak ve include korumasından geri dönecektir. Ön işlemcinin dosyayı açıp onun içinden geri dönmesi aşağıdaki gibi ek korumayla engellenebilir.

```
#ifndef _B_H_
#include "a.h"
#endif
```

Başlık dosyalarında başka başlık dosyaları bu biçimde include edilebilir.

Sınıfların Yazımında Dikkat Edilecek Biçimsel Durumlar

Sınıfın veri elemanlarının özel ön ek yada son eklerle ayrıştırılması sağlanabilir. Örneğin, veri elemanlarını *m_* yada *d_* ile başlatmak yaygındır, sınıfın static veri elemanlarına da *ms_* gibi önekler verilebilir.

Sınıfın private üye fonksiyonları public ve protected fonksiyonlarından daha değişik isimlendirilebilir. Örneğin, private fonksiyonların ilk sözcüklerinin tamamı küçük harflerle yazılabilir, sonraki sözcüklerin ilk harfleri büyük harflerle yazılabilir.(Camel Casting) public üye fonksiyonların isimlerinin her sözcüğünün ilk harfi büyük harfle yazılabilir. (Pascal Casting)

Sınıf bölümlerinin public, protected, private sırasına göre düzenlenmesi daha uygundur. Herkesi ilgilendiren elemanların sınıf bildiriminin başında olması daha uygundur.

```
class Sample{
```

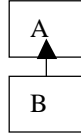
```
public:
    //...
protected:
    //...
private:
    //...
};
```

Sınıfların Sınıfları Kullanma Biçimleri

Bir sınıfın bir sınıfa bağımlılığı çeşitli düzeylerde olabilir. Bu bağımlılıklar UML sınıf diyagramlarında farklı biçimlerde temsil edilmektedir.

Türetme İlişkisi (Derivation):

Bir sınıf bir sınıftan türetilmişse bir türetme ilişkisi söz konusudur. Türetme ilişkisi sınıf diyagramlarında türemiş sınıftan taban sınıfa çekilen ok ile gösterilmektedir.



Türetme ilişkisine "is a" ilişkisi de denilmektedir.

İçerme İlişkisi (Composition):

Bir kavram yada nesne başka bir kavram yada nesneyi kapsıyorsa bu çeşit bir ilişki söz konusudur. Örneğin, bir sınıfın veri elemanı başka bir sınıf türünden nesne ise burada bir içerme ilişkisi vardır. İçerme ilişkisi sınıf diyagramlarında içerenden içerilene çekilen içi dolu yuvarlak yada carrow sembolü ile gösterilmektedir. Örneğin B sınıfının veri elemanı A sınıf türünden bir nesne olsun.

İçerme durumunda içeren nesne ile içerilen nesnenin ömürleri aynıdır.

Dışarıdan Bütünsel Kullanma İlişkisi (Aggregation):

Bir sınıf nenesi başka sınıf nesneleri tarafından içerme olmadan kullanılabilir. Örneğin, A sınıfına ilişkin bir nesne dinamik olarak yaratılmış olsun. Biz bu nesnenin adresini B ve C sınıfı türünden nesne yaratırken başlangıç fonksiyonuna geçirelim. Başlangıç fonksiyonu da bu adresleri de bir gösterici veri elemanında saklayabilir. B ve C sınıflarının üye fonksiyonları da bu gösterici yoluyla aynı nesneyi kullanabilir. Örneğin;

```
class B{
public:
    B(A *pA) : m_pA(pA)
    {}
    //...
private:
    A *m_pA
};
...
A *pA = new A();
...
```

```
B *pB = new B(pA);
```

...

Dışarıdan bütünsel kullanmanın içirme ilişkisinden farkı kullanılan nesnenin dışarıda daha önce kullandandan bağımsız olarak yaşaması ve belkide başka nesneler tarafından da paylaşılmasıdır. Dışarıdan bütünsel kullanma sınıf diyagramlarında genellikle kullandandan kullanılına doğru çekilen içi boş yuvarlak yada carrow ile temsil edilmektedir.

Dışarıdan Kısmi Kullanma (Association):

Bazen bir nesne sınıfın tüm üye fonksiyonları tarafından değil yalnızca bir yada birkaç üye fonksiyon tarafından kullanılır. Dahası bu fonksiyonlar aynı nesneyi değil farklı nesneleri de kullanıyor olabilir. Tipik olarak bir sınıfın bir üye fonksiyonunun parametresi başka bir sınıf türünden gösterici yada referans ise böyle bir durum söz konusudur.

```
class B{  
public:  
    void Func(A *pA);  
};
```

Bu durum sınıf diyagramlarında genellikle kullanan sınıftan kullanılan sınıfa doğru çekilen kesikli oklarla gösterilir.



Kavramların ve Nesnelerin Sınıflarla Temsil Edilmesi

Nesne yönelimli proje geliştirmesinde öncelikle ele alınacak projedeki tüm gerçek nesneler (yani örneğin eşyalar, aygıtlar gibi) ve kavramlar birer sınıfla temsil edilir. Bu işleme *dönüştürme (transformation)* denilmektedir. Örneğin, bir hastane için hasta takip programı yazılacak olsun. Projede konu olan gerçek nesnelerle kavramlar tespit edilir ve bunlar sınıflarla ifade edilir. Nesneler ve kavramlar şunlar olabilir:

- Doktorlar
- Hemşireler
- Hastalar
- İlaçlar
- Tıbbi Aygıtlar
- Hastanedeki Yataklar
- Muhasebe İşlemleri gibi...

Bu sınıflar arasındaki ilişkiler tespit edilmeye çalışılır. Örneğin, çalışanlar employee biçimde bir sınıfta tutulabilir ve doktorlar ve hemşireler bu sınıftan türetilir. Tıbbi aygıtlarda bir sınıfta türetilir. Hangi sınıflar hangi sınıfları kullanacaktır? Örneğin, doktorlar tıbbi aygıtları kullanabilirler. Hastalar, doktorları kullanırlar gibi...

Daha sonra sınıf diyagramları çizilerek temel belirlemeler daha somut hale getirilir ve kodlama aşamasına geçilebilir.

String Sınıfları

Hemen her nesne yönelimli programlama dilinde dilin kendi içerisinde ya da standart kütüphanesinde string işlemleri yapan bir string sınıfı vardır. Eskiden C++'da standart bir string sınıfı yoktu. Ancak STL kütüphanesinin C++'a eklenmesiyle string sınıfı da standart kütüphaneye katılmıştır.

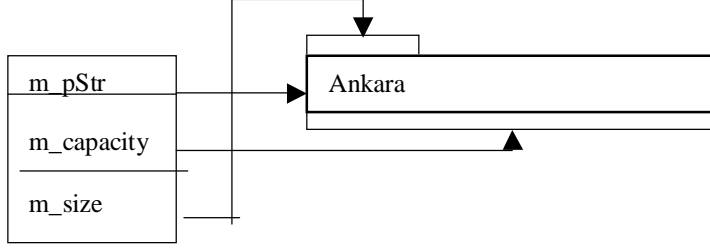
String sınıfları genellikle bir yazı üzerinde işlem yapmaya yarayan tüm üye fonksiyonları içerirler. Standart string sınıfı da bu bakımdan güçlü bir sınıftır.

Anahtar Notlar:

Programlama dillerinin standarnizasyonunda derleyici yazarlarını mümkün olduğu kadar serbest bırakacak bir esneklik öngörülmüştür. Yani örneğin bir sınıf standarnize edilirken sınıfın içsel yazımı üzerinde belirlemeler yapılmaz, yalnızca ara birim üzerinde belirlemeler yapılır. Sınıfın public ve protected bölümündeki üye fonksiyonların prototipleri ve bu fonksiyonların ne yapması gerektiği açıklanır. Yoksa bu sınıfın nasıl yazılması gerektiği, bu üye fonksiyonların nasıl algoritmalar kullanması gerektiği gibi içsel belirlemelere yer verilmez. Bu durumda örneğin, iki farklı firmanın String sınıfı da standartlara uygun olabilir, fakat bunların içsel yazımları tamamen birbirinden farklı olabilir.

İyi bir String sınıfının veri yapısı içerisinde neler olmalıdır? Bir kere iyi bir String sınıfı vektörel yapıya sahip olmalıdır, yani bir kapasite alanı olmalı, yazı bu alan içerisinde bu alandan daha kısa bir biçimde olmalıdır. Böylelikle yazıya küçük bazı eklemeler yapıldığında yeniden tahsisat yapılması engellenmiş olur. Yazının sonunda NULL karakterin bulunması gerekmeyebilir. Yazının sonunda NULL karakterin bulunması yazı adreslerinin dışarıya verilmesi durumunda fayda sağlamaktadır. Örneğin, Microsoft'un MFC kütüphanesi içerisindeki *CString* sınıfı yazıyı sonunda NULL karakter ile birlikte tutmaktadır. O halde tipik bir string sınıfının veri yapısı şöyle olabilir:

```
class String{  
    //...  
private:  
    char *m_pStr;  
    size_t m_capacity;  
    size_t m_size;  
};
```



Görüldüğü gibi *m_pStr* yazının tutulduğu heap alanını, *m_capacity* bu alanın uzunluğunu, *m_size* ise buradaki yazının uzunluğunu tutmaktadır. *m_size*, *m_capacity* değerine eriştiğinde alanda ne kadar büyütme yapılacağı sınıfı yazanlara bağlıdır. Şüphesiz sınıfı yazanlar bu büyütme miktarı için başta sistemdeki bellek miktarı olmak üzere bazı faktörleri gözönüne alacaklardır.

Standart string Sınıfı

Standart *string* sınıfı aslında tüm standart kütüphanelerde olduğu gibi template tabanlı bir sınıftır. Sınıfın template yapılmasının nedeni yazıyı oluşturan karakterlerin ASCII yada UNICODE olabilmesindendir.

Standart *basic_string* sınıfı, `<string>` başlık dosyası içerisinde yer almaktadır.

```
template<class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
class basic_string;
```

Görüldüğü gibi sınıfın üç template parametresi vardır. Açılım sırasında birinci template parametresi belirtilmek zorundadır. Bu template parametresi yazının her elemanının kaç byte ile tutulacağını belirtmektedir. Diğer template parametreleri daha sonra ele alınacaktır.

Tüm standart kütüphanenin *std* isim alanında olduğu unutulmamalıdır. *std* isim alanında *basic_string* sınıfının *char* türden ve *wchar_t* türünden açılımları sırasıyla *string* ve *wstring* olarak typedef edilmiştir.

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

basic_string Başlangıç Fonksiyonları

basic_string sınıfının pekçok faydalı başlangıç fonksiyonu vardır:

1. Default başlangıç fonksiyonu: Default başlangıç fonksiyonuyla bir nesne yaratıldığında kapasite değeri standartlara göre herhangi bir değerde olabilir, yani başlangıçta bir tahsisat yapılabilir ya da yapılmayabilir. Nesnenin tuttuğu string sıfır uzunluğunda boş bir stringtir.

2. Başka bir string nesnesinin belirli bir kısmından string yapan başlangıç fonksiyonu: Tipik kullanım şöyledir, "*string s(a, index, n);*" Burada *a* başka bir *basic_string* nesnesi, *index*, *a* stringinin içerisindeki karakter offseti, *n* ise o offsetten başlayarak ne kadar sayıda karakterin alınacağıdır. Üçüncü parametre default değer almıştır, belirtilmeyebilir. Belirtilmezse geri kalan tüm yazı anlamına gelir.

```
#include <iostream>
#include <string>
```



```
using namespace std;

int main()
{
    basic_string<char> s = "deniz";
    string d(s, 1);
    cout << d << endl;
    return 0;
}
```

Anahtar Notlar:

<string> başlık dosyası içerisinde ostream sınıfı türünden bir nesne ile bir stringi yazdırabilecek bir global operator fonksiyonu vardır. Örneğin,

```
string s = "ankara";
cout << s;
```

3. Bir göstericiyle belirtilmiş olan yazının ilk n karakterinden bir string oluşturan başlangıç fonksiyonu: Tipik kullanımı şöyledir;

```
char a[] = "ankara"
string s(a, 2);
```

4. Bir göstericiyle belirtilen yazıdan string oluşturan başlangıç fonksiyonu: Bu başlangıç fonksiyonu, yukarıdaki başlangıç fonksiyonunun ikinci parametresi olmayan biçimidir. Örneğin;

```
string s("ankara");
```

5. Diğer başlangıç fonksiyonları: Bu başlangıç fonksiyonları burada ele alınmayacaktır.

Sınıfın şüphesiz bir kopya başlangıç fonksiyonu vardır.

string Sınıfının Atama Operatör Fonksiyonları

1. Kopya Operatör Atama Fonksiyonu: İki string nesnesini birbirine atamakta kullanılan bir kopya atama operatör fonksiyonu vardır.

2. Bir göstericinin gösterdiği yerdeki yazıyı atayan bir atama operatör fonksiyonu da vardır: Örneğin;

```
string a("ankara");
a = "izmir";
```

3. Bir karakteri string'e atayan atama operatör fonksiyonu: Örneğin;

```
string a("ankara");
a = 'x';
```

Temel İşlemleri Yapan Üye Fonksiyonlar

string sınıfının içerisinde public bölümde *size_type* isimli bir typedef vardır. *size_type* default olarak *size_t* türündendir, *size_t* türü derleyicilerin çoğunda *unsigned int* türündendir. Ayrıca *string* sınıfının içerisinde *size_type* türünden *npos* ismiyle *static const* bir sabit tanımlanmıştır. Bu sabit çeşitli durumlarda başarısızlığı anlatmak için kullanılmaktadır. Standartlara göre bu sabitin değeri -1 dir. (*size_type* işaretli bir tür olduğuna göre -1 sayısı *size_type* türüne sığabilecek en büyük pozitif sayıdır.)

size ve lenght Fonksiyonları

string sınıfının tuttuğu yazının sonunda NULL karakter yoktur. Bu iki üye fonksiyon tamamen birbirinin eşdeğeridir, yazının karakter uzunluğunu verir.

```
size_type size() const;
```

```
size_type lenght() const;
```

max_size Fonksiyonu

Bu fonksiyon bir string nesnesinde tutulabilecek yazının o sistemdeki maximum uzunluğunu vermektedir.

```
size_type max_size() const;
```

capacity Fonksiyonu

Bu fonksiyon yazı için ayrılan alanın o anki uzunluğunu vermektedir.

```
size_type capacity() const;
```

resize Fonksiyonları

resize fonksiyonları standart kütüphane içerisindeki başka sınıflarda da benzer anlamlara gelen bir fonksiyondur. Genel olarak *resize* fonksiyonları veri yapısının kapasitesi üzerinde değil, o anda tutulan elemanın uzunluğu üzerinde etkili olurlar. Tabi şüphesiz *resize* işlemi arka planda kapasitenin değiştirilmesine de yol açabilir.

```
size_type resize(size_type n);
```

Parametre olarak girilen *n* sayısı o andaki yazının uzunluğundan küçükse yazı kırpılarak küçültülür. Eğer *n* sayısı o andaki yazının uzunluğundan büyük ise yazı genişletilir, genişletilen kısım NULL karakterlerle doldurulur. Örneğin;

```
string s = "ankara";
```

```
s.resize(2); //an
```

```
s.resize(10); //an\0\0\0\0\0\0\0
```

resize fonksiyonunun diğer biçimi şöyledir:

```
size_type resize(size_type n, charT c);
```

Fonksiyonun birinci parametresi yazının yeni uzunluğu, ikinci parametresi eğer yeni uzunluk yazının toplam uzunluğundan büyükse doldurulacak karakterdir.

Anahtar Notlar:

Standart kütüphane içerisindeki template sınıflar içerisindeki bazı fonksiyonlara aynı isimler verilmiştir. Örneğin, string sınıfının da resize isimli bir fonksiyonu vardır, vector sınıfının da. Aynı isimli fonksiyonlar o sınıflara özgü benzer işlemleri yapmaktadır. Örneğin;

- *size isimli fonksiyonlar; veri yapısı içerisindeki eleman sayısını elde etmekte kullanılır.*
- *capacity isimli fonksiyonlar; veri yapısı için tahsis edilmiş olan alanın uzunluğunu almakta kullanılır.*
- *resize isimli fonksiyonlar; size değerini büyütmek ya da küçültmek için kullanılırlar.*
- *reserve isimli fonksiyonlar; kapasiteyi değiştirmek için kullanılırlar. Genel olarak nesne tutan sınıflarda kapasite düşümü yapılmamaktadır. Yani kapasite arttırılmakta fakat azaltılamamaktadır.*

reserve Fonksiyonu

void reserve(size_type res_arg = 0);

Fonksiyonun parametresi eski kapasite değerinden küçükse genel olarak alan küçültülmesine gidilmez. Fakat standartlarda *vector* sınıfı için kesin bir küçültme yapılmayacağı söylenmişse de *string* için söylenmemiştir. Fonksiyon çağırıldıktan sonra kapasite değerinin parametreyle belirtilen değerde olması zorunlu değildir. Yani fonksiyon istenilenden daha büyük bir alanı tahsis edebilir, ya da istenilen kadar alanı tahsis edebilir.

clear Fonksiyonu

clear fonksiyonu da pekçok nesne tutan sınıfta var olan ortak bir fonksiyondur. Bu fonksiyon veri yapısının tüm elemanlarını siler. Silme işleminden sonra kesinlikle *size* değeri 0 olmaktadır.

void clear();

[] Operator Fonksiyonları ve at Fonksiyonları

Sınıfın belirli indexteki elemanını elde etmek için kullanılan `const` olan ve olmayan `[]` operator fonksiyonu ve *at* fonksiyonu vardır. `[]` operator fonksiyonu index kontrolünü yapmamaktadır.

Halbuki *at* fonksiyonları eğer *index* değeri *size* değerinden büyük ise *out_of_range* türüyle throw etmektedir.

+= Operatör Fonksiyonu

Sınıfın += operatör fonksiyonları yazının sonuna ekleme yapmak için kullanılmaktadır. bir string nesnesi içerisindeki yazıyı sona ekleyen, tek bir karakteri sona ekleyen ve gösterici yoluyla yazılmış bir yazıyı sona ekleyen biçimleri vardır.

append Fonksiyonları

append fonksiyonları da ekleme amaçlı kullanılmaktadır. Fakat *append* fonksiyonunun bir biçimi istenilen bir indexten sonraya ekleme yapabilmektedir.

insert Fonksiyonları

Sınıfın *insert* fonksiyonları tipik olarak bir *index*' e bir yazıyı insert etmek amacıyla kullanılır. Çeşitli biçimleri vardır. Örneğin;

- Belli bir *index*' e bir string nesnesinin içerisindeki yazıyı insert eden fonksiyon.
- Bir string nesnesi içerisindeki yazının belli bir *index*' inden itibaren *n* karakteri belirli bir *index*' e insert eden fonksiyon.
- Adres yoluyla verilmiş bir yazının başındaki *n* karakteri belirli bir *index*' e insert eden fonksiyon.
- Bir karakterden *n* tane belirli bir *index*' e insert eden fonksiyon.

using namespace std;

int main(void)

```
{  
    string s("deniz");  
    s.insert(s.size(), 2, 'a');  
    cout << s << endl;  
    return 0;  
}
```

Anahtar Notlar:

Standart kütüphane içerisinde pekçok sınıf ve sınıfların pekçok üye fonksiyonu vardır. Bu fonksiyonların parametrik yapısının tamamen bilinmesi çok zordur. Bu nedenle öğrenme sürecinde genel olarak hangi tür fonksiyonların bulunduğuna yoğunlaşılmalı ve gerekirse duruma göre fonksiyonlar referanslara başvurarak incelenmelidir.

erase Fonksiyonları

erase fonksiyonları genel olarak pekçok sınıfta aynı isimle bulunmaktadır. Bu fonksiyonlar veri yapısından bazı elemanları silmek için kullanılır. *string* sınıfının *erase* fonksiyonu belirli bir *index*'ten itibaren *n* tane karakteri silmekte kullanılır. *Index* pozisyonunun *size* değerinde \leq olması gerekmektedir. Aksi halde *out_of_range* türüyle *throw* işlemi oluşur. *erase* fonksiyonunun iki parametresi de default değer almaktadır. Bu default değerler kullanılırsa tüm elemanlar silinmektedir. İkinci parametre verilmezse geri kalanının hepsi anlaşılır.

```
using namespace std;
int main(void)
{
    string s("deniz");
    s.erase(2);
    cout << s << endl;
    return 0;
}
```

replace Fonksiyonları

replace fonksiyonları genel olarak bir yazının belli bir bölümünü silip onun yerine başka bir yazının belli bir bölümünü eklemek amacıyla kullanılır. Fonksiyonlar bir *index* numarasını, o *index* numarasından itibaren silinecek karakter sayısını ve yerleştirilecek yazıyı ve o yazının neresinin yerleştirileceği bilgisini almaktadırlar.

```
using namespace std;
int main(void)
{
    string s("deniz"), t("izmir");
    s.replace(2, 2, t);
    cout << s << endl;
    return 0;
}
```

copy Fonksiyonu

Bu fonksiyon bir yazının belirli bir bölümünü belirli bir *index*'ten itibaren kopyalar. *replace* fonksiyonundan farklı olarak bir silme yada insert işlemi yapmamaktadır.

data ve c_str Fonksiyonları

data fonksiyonu tahsis edilmiş olan tamponun başlangıç adresine geri dönmektedir. Alan içerisindeki yazının sonunda NULL karakter olmadığına göre bu adres standart C fonksiyonlarında kullanılamaz. Halbuki *c_str* fonksiyonu tahsis edilmiş alandaki yazıyı başka bir alana kopyalar. Sonuna NULL karakteri yerleştirir ve o yeni alanın adresiyle geri döner. Yani *c_str* fonksiyonunun verdiği adres standart C fonksiyonlarında kullanılabilir.

```
using namespace std;
```

```
int main(void)
{
    string s;
    s= "ankara"
    s.resize(3);
    cout << s.data << endl;
    return 0;
}
```

find ve rfind Fonksiyonları

Bu fonksiyonlar genel olarak string nesnesinin belirttiği yazı içerisinde başka bir yazıyı yada karakteri aramaktadır. Fonksiyon aradığını bulursa bulunduğu yerin index numarasıyla, bulamazsa *string::npos* değeriyle geri döner. *rfind* aramayı sondan itibaren, *find* baştan itibaren yapmaktadır. Fonksiyonlar aramanın başlatılacağı bir index numarasını da parametre olarak almaktadır.

```
using namespace std;
int main(void)
{
    string s;
    s= "ankara"
    string::size_type pos;
    if((pos = s.find("kara")) != string::npos)
        cout << "buldu: " << pos << endl;
    else
        cout << "bulamadı\n";
    return 0;
}
```

Örneğin, bir yazıyı arayalım, o yazıyı bulursak o yazı yerine başka bir yazıyı yerleştirelim.

```
using namespace std;
int main(void)
{
    string s;
    s= "ankara"
    string::size_type pos;
    if((pos = s.find("kara")) != string::npos)
        s.replace(pos, 4, "ak")
    cout << s << endl;
    return 0;
}
```

Burada "kara bulunmuş onun yerine "ak" yazısı yerleştirilmiştir.

Sınıf Çalışması:

Üç string nesnesi alınız, birinci string' e bir yazı yerleştiriniz. İkinci ve üçüncü string' lere sırasıyla aranacak ve değiştirilecek yazıları yerleştiriniz. Birinci string' teki yazı içerisinde ikinci string' teki tüm yazılar yerine üçüncü string' teki yazıları yer değiştiriniz.

```
#include <cstring>
#include <string>
#include <iostream>

using namespace std;
int main(void)
{
    string s = "bugun ay yil ay sali ay";
    string f = "ay";
    string r = "xx";

    string::size_type pos1 = 0;

    while ((pos1 = s.find(f, pos1)) != string::npos)
        s.replace(pos1, f.size(), r);

    cout << s << endl;
    return 0;
}
```

substr Fonksiyonu

Bu fonksiyon bir yazının belirli bir kısmından başka bir string nesnesi yapıp, o nesne ile geri dönmektedir. Fonksiyonun birinci parametresi başlangıç index numarası, ikinci parametresi o index' ten itibaren elde edilecek karakter sayısıdır. Her iki parametre de default değer alabilir. Birinci parametre yazılmazsa baştan itibaren, ikinci parametre yazılmazsa birinci parametreden sona kadar olan kısım anlaşılır.

```
#include <cstring>
#include <string>
#include <iostream>

using namespace std;
int main(void)
{
    string s = "c:\\a\\b\\c.bat";

    string::size_type pos1 = 0, pos2;
    string name;

    while ((pos2 = s.find('\\', pos1)) != string::npos){
        name = s.substr(pos1, pos2 - pos1);
        pos1 = pos2 + 1;
    }
```

```
        cout << name << endl;
    }
    if(pos2 != s.size()){
        name = s.substr(pos1);
        cout << name << endl;
    }
    return 0;
}
```

Karşılaştırma İşlemleri

Altı karşılaştırma operatörünün hepsi için operatör fonksiyonu yazılmıştır. Bu operatör fonksiyonlarının hem string biçimi hem de gösterici biçimi vardır. Yani, $s > t$ işlemi, $s > \text{"deniz"}$ işlemi ve $\text{"deniz"} > s$ işlemlerinin hepsi geçerlidir. Bu operatör fonksiyonlarının hepsi global operatör fonksiyonu olarak yazılmıştır.

Ayrıca sınıfın int geri dönüş değerine sahip çeşitli *compare* fonksiyonları da vardır. Bu fonksiyonlar genel olarak iki yazının belirli bölümlerini karşılaştıracak biçimdedir.

string Sınıfı için Okuma ve Yazma Fonksiyonları

string sınıfı kütüphanenin temel bir sınıfı olarak ele alınmaktadır. Bu nedenle *string* nesnesi ile bağlantılı << ve >> operator fonksiyonları da kütüphaneye eklenmiştir.

string nesnesi için okuma yapan >> operatör fonksiyonu aşağıdaki gibi kullanılabilir.

```
string s;
```

```
cin >> s;
```

Fakat bu operatör fonksiyonu tampondaki ilk boşluksuz karakter kümesini almaktadır.

string nesnesi içerisindeki yazıyı bastırmak << operatör fonksiyonu şöyle kullanılabilir.

```
string s;
```

```
cout << s;
```

Ayrıca kütüphanede tamamen *gets* fonksiyonu gibi bir satırlık bilgiyi alan ve bunu string nesnesi içerisine yerleştiren global bir *getline* fonksiyonu vardır. Aşağıdaki gibi kullanılır:

```
string s;
```

```
getline(cin, s);
```

Şüphesiz, *string* nesnesine ilişkin bu fonksiyonların hepsi <string> başlık dosyası içerisinde.

getline fonksiyonunun diğer bir biçimi belirli bir karakter görünce okumayı sonlandıran üç parametrelidir. Şöyle kullanılabilir:

```
string s;
```

```
getline(cin, s, 'x');
```

Burada kesim karakteri string'e yerleştirilmemektedir.

Örnek:

```
#include <iostream>
#include <string>
#include <cstring>
#include <windows.h>

typedef struct tagCMD {
    const char *pCmd;
    void (*pProc) ();
} CMD;

void Dir();
void Erase();
void Exit();
using namespace std;
const char *PROMPT = "CSD";
string g_cmd, g_param;
CMD g_commands[] = {{"dir", Dir}, {"erase", Erase}, {"exit", Exit}, {0, 0}};

int main(void)
{
    string cmdLine;
    string::size_type pos1, pos2;
    for (;;) {
        cout << PROMPT << "> ";
        getline(cin, cmdLine);
        pos1 = cmdLine.find_first_not_of("\t", 0);
        pos2 = cmdLine.find_first_of("\t", pos1);
        g_cmd = cmdLine.substr(pos1, (pos2 != string::npos) ? pos2 - pos1 :
string::npos);
        if (pos2 != string::npos) {
            pos1 = cmdLine.find_first_not_of("\t", pos2);
            g_param = cmdLine.substr(pos1);
        }
        else
            g_param.clear();
        int i;
        for (i = 0; g_commands[i].pCmd != 0; ++i) {
            if (g_cmd == g_commands[i].pCmd) {
                g_commands[i].pProc();
                break;
            }
        }
        if (g_commands[i].pCmd == 0) {
            cout << "bad command or file name!..\n";
        }
    }
}
```

```
        return 0;
    }
    void Dir()
    {
        cout << "dir\n";
    }
    void Erase()
    {
        cout << "erase\n";
    }
    void Exit()
    {
        exit(EXIT_SUCCESS);
    }
}
```

Standart Kütüphanenin Genel Yapısı

Standart kütüphane temel olarak bir grup *template sınıf* ve bir grup *template fonksiyon*dan oluşmaktadır, *template fonksiyon*lara algoritma da denilmektedir. *template sınıflar* da yine kendi araların çeşitli bölümlere ayrılarak incelenebilir.

- **Genel amaçlar için kullanılan yararlı bir takım sınıflar:** Örneğin; string sınıfı, auto_ptr gibi sınıflar.
- **Nesne tutan sınıflar:** Bu sınıflara İngilizce *container class* denilmektedir. (Java terminolojisinde bu tür sınıflara *collectin class* denilmektedir.) Örneğin; bağlı liste işlemleri yapan list sınıfı, dinamik dizi büyütme işlemleri yapan vector sınıfı gibi.
- **Teşhis amaçlı kullanılan sınıflar:** Kütüphanenin kendi içerisinde yapılan exception işlemleri yakalamak için kullanılan sınıflardır.
- **İterator sınıfları:** İterator işlemleri için kullanılan sınıflardır.

İterator Kavramı

Standart kütüphanenin en önemli kavramı *iterator* kavramıdır. Kütüphane içerisindeki tüm algoritmalar *iterator* kavramını destekleyecek biçimde tasarlanmışlardır.

Iterator, gösterici gibi davranan nesnelerdir. Bir nesnenin gösterici gibi davranması için ya gerçekten bir gösterici olması ya da gösterici operatorlerine ilişkin operator fonksiyonlarına sahip bir sınıf nesnesi olması gerekir. Görüldüğü gibi *iterator* programcı için gösterici gibi kullanılan bir kavramdır, fakat gerçekte bir gösterici olmayabilir.

*Iterator*ler, nesne tutan sınıfların dolaşılmasını sağlayan onların türü ne olursa olsun, sanki bir diziymiş gibi kullanılmasını sağlayan nesnelerdir.

Nesne tutan sınıfların hepsinde *iterator* isminde, public bölümünde bir typedef ismi vardır. Örneğin, `vector<int>::iterator`

```
list<int>::iterator
```

```
basic_string<char>::iterator
```

```
...
```

Sınıflar içerisindeki bu *iterator* typedef ismi bir gösterici türünü temsil etmektedir. Yani biz *iterator* türünden bir nesne tanımlarsak o nesne gösterici gibi davranacaktır.

Programcı herhangi bir nesne tutan sınıf türünden *iterator* nesnesi tanımladığında, bu *iterator* nesnesinin gerçek türünü bilmek zorunda kalmaz. Programcı için bu *iterator* nesnesinin gösterici gibi davranacak olması yeterlidir. Böylece tüm veri yapılarını dolaşmak için ortak bir tür ve davranış kalıbı elde edilmiş olur. Yani programcının elinde bir *iterator* varsa bu *iterator* vector sınıfına ilişkin olsa da list sınıfına ilişkin olsa da programcı tarafından aynı biçimde kullanılır.

Tüm nesne tutan sınıfların *begin* ve *end* isimli iki önemli public fonksiyonu vardır. bu fonksiyonlar geri dönüş değeri olarak o sınıf içerisinde bildirilen *iterator* türüne ilişkin bir değer vermektedir. *begin* fonksiyonunun verdiği *iterator*, nesne tutan sınıfın ilk elemanını temsil eden bir gösterici gibidir. Bir *iterator* * operatörüyle kullanırsak, sanki *iterator* bir göstericiymiş de bizde göstericinin gösterdiği yerdeki elemana erişiyormuşuz gibi iteratorun temsil ettiği elemana erişiriz.

```
basic_string<char> s("deniz");
```

```
basic_string<char>::iterator iter;
```

```
iter = s.begin();
```

```
assert(*iter=='d');
```

Şimdi, *iter* bizim için dizide bir elemanı gösteren gösterici gibidir. Dizin her elemanı da karakterlerden oluşmaktadır. O halde biz *++iter* dedikten sonra **iter* yaparsak sonraki karaktere erişiriz.

O halde bir *iterator* ile hangi türden nesne tutan bir sınıf olursa olsun, dolaşım şöyle yapılır.

1. O sınıf türünden bir *iterator* nesnesi tanımlanır.
2. Nesne tutan sınıfın *begin* üye fonksiyonuyla ilk elemana ilişkin *iterator* değeri alınır.
3. Bir döngü içerisinde ++ operatörüyle veri yapısı dolaşılır, * operatörüyle elemanlar elde edilir.

Burada tek problem veri yapısının sonuna gelindiğinin tespit edilmesidir. İşte nesne tutan sınıfların *end* üye fonksiyonları tuttukları son nesneden bir sonrakinin *iterator* değerini geri vermektedir. O halde tipik dolaşım şöyle yapılır:

```
for(iter = s.begin(); iter != s.end; ++iter){
```

```
    //...
```

```
}
```

```
#include <iostream>
```

```
#include <string>
```

```
#include <cstring>
```

```
#include <cctype>
```

```
#include <windows.h>
```

```
using namespace std;
int main(void)
{
    basic_string<char> s("deniz");

    basic_string<char>::iterator iter;

    for (iter = s.begin(); iter != s.end(); ++iter)
        cout << *iter << endl;

}
```

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <windows.h>
```

```
using namespace std;

int main(void)
{
    vector<char> s;

    s.push_back('k');
    s.push_back('a');
    s.push_back('a');
    s.push_back('n');

    vector<char>::iterator iter;

    for (iter = s.begin(); iter != s.end(); ++iter)
        cout << *iter << endl;

}
```

Nesne Tutan Fonksiyonların İterator Veren Fonksiyonları

Nesne tutan sınıfların *begin* ve *end* gibi iterator veren üye fonksiyonları kendi sınıfları içerisinde typedef edilmiş iterator türüne geri dönerler. O halde biz bu fonksiyonların geri dönüş değerlerini saklayabilmek için o sınıfların içerisinde belirtilen iterator türünü kullanmalıyız. Örneğin;

```
list<int> x;
list<int> iterator iter;
```

```
iter = x.begin;
```

Ayrıca `list<int>::iterator` türüyle `list<long>::iterator` türleri gerçekte aynı tür olmayabilir.

Iterator konusunun daha iyi anlaşılabilmesi için kendimiz bir *Array* sınıfı yazalım.

```
using namespace std;
```

```
template <class T>
class Array {
public:
    typedef T *iterator;
public:
    Array(int size) : m_size(size), m_pArray(new T[size])
    {}
    ~Array()
    {
        delete [] m_pArray;
    }
    iterator begin()
    {
        return m_pArray;
    }
    iterator end()
    {
        return m_pArray + m_size;
    }
    // ...
private:
    T *m_pArray;
    int m_size;
};
```

Şimdi iskelet bir bağlı liste sınıfını ele alarak iterator türünün normal bir gösterici değil de bir sınıf olabileceğini görelim.

```
template <typename T>
class List {
private:
    struct Node {
        T m_val;
        Node *m_pNext;
        Node(const T &val) : m_val(val)
        {}
        void Func();
    };
private:
```

```
class ListIterator {
public:
    ListIterator(Node *pNode = 0) : m_pNode(pNode)
    {}
    ListIterator & operator ++()
    {
        m_pNode = m_pNode->m_pNext;

        return *this;
    }

    T operator *()
    {
        return m_pNode->m_val;
    }

    bool operator !=(const ListIterator &iter)
    {
        return m_pNode != iter.m_pNode;
    }

private:
    Node *m_pNode;
};

public:

List() : m_pHead(0), m_pTail(0), m_size(0)
{}
void AddHead(const T &val)
{
    Node *pNode = new Node(val);

    if (m_pHead == 0)
        m_pTail = pNode;

    pNode->m_pNext = m_pHead;
    m_pHead = pNode;

    ++m_size;
}

typedef ListIterator iterator;

iterator begin()
{
    return iterator(m_pHead);
}
```

```

    iterator end()
    {
        return iterator(0);
    }

private:
    Node *m_pHead;
    Node *m_pTail;
    int m_size;
};

```

Standart Kütüphanedeki Algoritmaların İteratorlerle İlişkisi

Standart kütüphanedeki algoritma dediğimiz fonksiyonlar hem dizilerle hem de nesne tutacak sınıflarla çalışacak biçimde tasarlanmışlardır. Yani bu fonksiyonlar içerisinde iteratörlerin desteklediği operatörler kullanılmıştır. Bu algoritmalar genel olarak bir diziyi parametre olarak alacakları zaman onun ilk elemanının ve son elemanından bir sonraki ilk elemanının adresini alırlar. Böylece bu fonksiyonlara sıradan diziler de geçilebileceği gibi nesne tutan sınıflarda geçirilebilmektedir. Şüphesiz bu template fonksiyonlar her farklı iteratör türü için derleyici tarafından yeniden yazılmaktadır.

İteratörlerin Sınıflandırılması

Bir iteratör ile hangi operatörleri kullanabiliriz? Bir iteratör bir gösterici gibi davrandığına göre göstericilerle kullanabildiğimiz her operatörü iteratör türüne de uygulayabilir miyiz? İşte bir iteratörü hangi operatörlere uygulayacağımız, o iteratörün türüne bağlı olarak değişebilmektedir. İteratörler destekledikleri operatörlere göre beş gruba ayrılırlar.

1. Girdi İteratörleri (Input Iterators): Girdi iteratörleri yalnızca ileri doğru operatörüyle ilerlemeye izin veren, * operatörüyle elemanı elde edebilen fakat onu değiştiremeyen iteratörlerdir. Girdi iteratörleri şu operatörleri destekler: *iter, iter++, iter->eleman, ++iter, iter1 == iter2, iter1 != iter2

2. Çıktı İteratörleri (Output Iterators): Bu iteratörleri tamamen girdi operatörlerini içermekle birlikte veri yapısındaki elemanı da değiştirebilirler. Şu operatörleri destekler: *iter, iter++, iter->eleman, ++iter, iter1 == iter2, iter1 != iter2

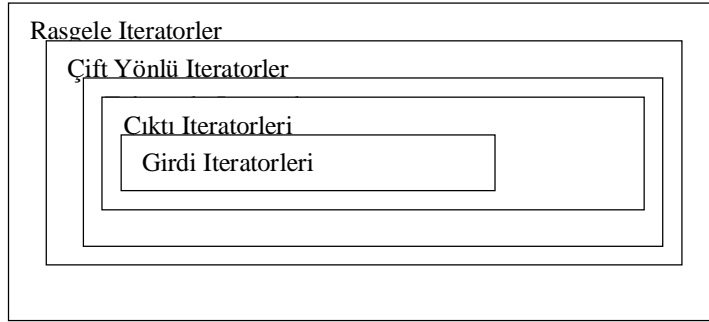
3. Tek Yönlü İlerleyen İteratörler (Forward Iterators): Bu iteratör grubu da yetenek olarak çıktı iteratörlerini kapsamaktadır. Tipik olarak veri yapısında tek yönlü ilerleme için kullanılır. Çıktı iteratöründen farklı olarak iki iteratörün atamasına olanak sağlayan atama operatörünü de destekler. Şu operatörleri destekler: *iter, iter++, iter->eleman, ++iter, iter1 == iter2, iter1 != iter2, iter1 = iter2

4. Çift Yönlü İlerleyen İteratörler (Bidirectional Iterators): Bu iteratör grubu da tek yönlü ilerleyen iteratör grubunu yetenek olarak kapsamaktadır. Fakat ayrıca -- operatörünü desteklediğinden dolayı geriye doğru gitmeye de olanak sağlar. Şu operatörleri destekler: *iter, iter++, iter->eleman, ++iter, iter1 == iter2, iter1 != iter2, iter1 = iter2, --iter, iter--

5. Rasgele Erişimli Iteratorler (Random Access Iterators): Rasgele erişimli iteratorler yetenek olarak çift yönlü ilerleyen iteratorleri kapsamaktadır. Bu iterator grubu normal bir göstericinin desteklediği her türlü operatörü desteklemektedir. Örneğin, bu iteratorler ile biz [] operatörünü kullanarak veri yapısının herhangi bir elemanına çok hızlı erişebiliriz.

Rasgele erişimli iteratorler tamamen bir gösterici gibi davranabilen iteratorlerdir. Destekledikleri operatörler şunlardır: tüm çift yönlü ilerleyen iteratorün desteklediği operatörler, iter[n], iter += n, iter -= n, n+iter, iter+n, iter-n, iter1-iter2, iter1>iter2, iter<iter2, iter1>=iter2, iter<=iter2

Görüldüğü gibi iterator grupları birbirlerini kapsar niteliktedir.



Iterator Grupları ve Standart Kütüphane

Standart kütüphanedeki tüm global template fonksiyonların hangi türden iteratorlere gereksinim duyduğu belirlenmiştir. Örneğin, *find* isimli fonksiyon parametre olarak girdi iteratoru alır. O halde bu fonksiyon yazılırken yalnızca iteratorlerinin desteklediği operatörler kullanılmıştır. Fakat örneğin *binary_search* isimli fonksiyon rasgele erişimli iteratorlere gereksinim duymaktadır. Biz bu fonksiyona çift yönlü ilerleyen iteratorleri parametre olarak veremeyiz. Muhtemelen bu fonksiyonun içerisinde [] operatörü kullanılmıştır. Gerek standartlarda gerekse derleyicilerin dökümanlarında, global template fonksiyonların prototiplerinde bu fonksiyonların hangi türden iteratorlere gereksinim duyduğu anlaşılabilmektedir. Örneğin, standartlarda *find* fonksiyonu aşağıdaki prototiple belirtilmiştir.

```
template<class InputIterator, classT>
```

```
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

Ayrıca standart kütüphanede nesne tutan sınıfların *begin* ve *end* gibi iterator veren fonksiyonlarının hangi türden iterator verdiği açıkça belirtilmiştir. Örneğin, *List* sınıfının iteratorleri *çift yönlü ilerleyen iteratorler (bidirectional iterators)* biçimindedir. Bu durumda biz *List* sınıfının iteratorlerini *find* fonksiyonunda kullanabiliriz ama *binary_search* fonksiyonunda kullanamayız. Başka bir deyişle *List* sınıfı üzerinde ikili arama yapamayız. Fakat örneğin, standartlarda *Vector* sınıfının iteratorlerinin *rasgele erişimli iteratorler* olduğu belirtilmiştir. O halde biz *Vector* sınıfını ikili arama işleminde kullanabiliriz.

Algoritmanın Karmaşıklığı

Bilindiği gibi algoritmanın karmaşıklığı, algoritmaları hız bakımından birbirleriyle kıyaslamak için kullanılan bir kavramdır. Bu kavrama göre algoritma içerisinde onu iyi temsil eden bir işlem seçilir. O işlemten en kötü olasılıkla ya da ortalama olasılıkla kaç defa yapılması gerektiği

hesaplanır. Bu hesap değeri ‘O’ notasyonu denilen bir notasyonla belirtilmektedir. Örneğin, dizinin uzunluğu n olmak üzere bir elemanı sıralı aramanın ortalama karmaşıklığı O notasyonuna göre;

$$\frac{n+1}{2}$$

Şüphesiz O notasyonu ile değerin bulunması zor olabilmektedir. Bu nedenle algoritmalar karmaşıklık katagorilerine ayrılarak ele alınabilirler. Tipik karmaşıklık katagorileri şunlardır:

1. **Sabit Karmaşıklık:** İşlem için döngü gerekmiyorsa basit bir takım yalın işlemlerle sonuca gidiliyorsa bu tür algoritmalara sabit karmaşıklığa sahip algoritmalar denir. Sabit karmaşıklık O notasyonunda “O(n)=1” şeklinde gösterilebilir.
2. **Logaritmik Karmaşıklık:** Bu tür algoritmalarda döngü vardır ama dizinin uzunluğu n olmak üzere $\log_2 n$ kadar dönmektedir. O notasyonunda “O(n)= $\log_2 n$ ” şeklinde gösterilebilir.
3. **Doğrusal Karmaşıklık:** Bir algoritmada iç içe döngü yoksa fakat bir ya da birden fazla tekil döngüler varsa bu karmaşıklığa doğrusal karmaşıklık denir. O(n) = n ile gösterilir.
4. **Doğrusal Logaritmik Karmaşıklık:** Bir algoritmada logaritmik tane tekil döngü varsa bu türden karmaşıklığa sahiptir. O(n)=nlogn ile gösterilir.
5. **Karesel Karmaşıklık:** Bir algoritmada iç içe iki döngü varsa (buna ek olarak tekil döngülerde olabilir.) karesel karmaşıklığa sahiptir. O(n)= n^2 ile gösterilir.
6. **Küpsel Karmaşıklık:** Bir algoritmada iç içe üç döngü varsa küpsel karmaşıklığa sahiptir. O(n)= n^3 ile gösterilir.
7. **Diğer Karmaşıklıklar:** Bu grupların dışında kalan karmaşıklıklarda vardır. Örneğin O(n)= 2^n biçiminde üstel bir karmaşıklık söz konusu olabilir.

Standart Kütüphane ve Algoritmanın Karmaşıklığı

Standartlarda standart kütüphanedeki global template fonksiyonlar ve sınıfların üye fonksiyonları yalnızca prototip verilerek parametrik bakımdan açıklanmışlardır. Yani açıklama yazıma ilişkin değildir, arayüze ilişkindir. Fakat ayrıca algoritmik fonksiyonlar için gereken minimum karmaşıklıkta belirtilmiştir. Yani örneğin, derleyicileri yazanlar sort template fonksiyonunu prototipte belirtilen parametrik yapıya sahip olmak üzere istedikleri gibi yazabilirler. Fakat kullandıkları algoritmanın nlogn karmaşıklığında olması gerekir. Şüphesiz, aynı karmaşıklığa sahip olan fonksiyonlarda yinede kalite farkı söz konusu olabilmektedir.

Vector Sınıfı

Dinamik büyütülen dizileri temsil eden sınıflara *Vector* denilmektedir. *Vector* sınıfı bir dizi gibi kullanılır. Ancak diziden farklı olarak eleman insert edildiğinde ya da sonuna eleman eklendiğinde dizi de otomatik olarak büyütülmektedir. Programcı *Vector* sınıfını kullanırken dizinin büyütülmesi işlemini tamamen sınıfa bırakır.

Bir *Vector* sınıfının tipik yazımı şöyle gerçekleştirilir:

Sınıfa tahsis edilen alanın başlangıç adresini tutan bir gösterici, tahsis edilen alanın uzunluğunu tutan bir veri elemanı ve o anda dizi içerisindeki eleman sayısını tutan başka bir veri elemanı yerleştirilir.

```
template<class T>
class Vector{
    //...
private:
    T *m_pVector;
    size_t m_capacity;
    size_t m_size;
};
```

Nesne yaratıldığında henüz bir tahsisat yapılmayabilir. Tahsisat ilk eleman eklendiğinde yapılabilir. Önce *Vector* alanı için *m_capacity* kadar bir yer tahsis edilir, sonra eleman eklendikçe *m_size* değeri artırılır. *m_size*, *m_capacity* değerine eriştiğinde eleman ekleyen ya da insert eden üye fonksiyonlar yeniden tahsisat işlemi yaparak *m_capacity* değerini yükseltirler. Genellikle *Vector*' lerin genişletilmesinde iki kat genişletme yöntemi kullanılmaktadır.

Standart vector Sınıfının Temel Özellikleri

Standartlara göre *vector* sınıfındaki elemanlar ardışıl olarak tutulmak zorundadır. (Yani *vector*' un elemanları tamamen dizilerde olduğu gibi peşi sıra gitmektedir. *vector* elemanlarının ardışıl olma zorunluluğu standartların 1998 versiyonunda belirtilmemiştir, bu özelliğin belirtilmemiş olması bir böcek kabul edilmekteydi. 2003 standartlarında düzeltme yapılırken bu durum açıkça belirtilmiştir.) Standart *vector* sınıfında kapasite küçültülmesine asla gidilmemektedir. Yani biz *vector* deki elemanların büyük çoğunluğunu silsek bile daha önceden tahsis edilmiş olan alanlar geri bırakılmaz.

Vector sınıfının iteratorleri *rasgele erişimli* iteratorlerdir. Yani biz *vector* türünden bir nesnenin elemanına sanki bir diziymiş gibi [] operatörüyle doğrudan erişebiliriz. *vector*' ün sonuna eleman eklenmesi ya da sonundaki elemanın silinmesi sabit zamanlı yani hızlı bir işlemdir. Fakat araya eleman ekleme ya da aradan eleman silme doğrusal karmaşıklığa sahiptir. Ayrıca *vector* sınıfında başa eleman ekleyen klasik *push_front* fonksiyonu ve baştaki elemanı silen *pop_front* fonksiyonu bulunmamaktadır.

Vector sınıfının bildirimi <vector> başlık dosyası içerisinde yer almaktadır.

```
using namespace std;
```

```
int main(void)
{
    vector<int> a;

    for (int i = 0; i < 100; ++i)
        a.push_back(i);

    for (int k = 0; k < a.size(); ++k)
        cout << a[k] << "\n";
}
```

```
    return 0;  
}
```

Vector Sınıfının Template Parametreleri

vector sınıfının iki template parametresi vardır. birinci template parametresi *vector*' ün tutacağı elemanların türünü belirtir ve kesinlikle yazılmak zorundadır. İkinci template parametresi tahsisatlar için kullanılacak *Allocator* sınıfını belirtmektedir. Bu parametre belirtilmeyebilir, bu durumda tahsisatlar için standart *allocator* sınıfı kullanılır.

Anahtar Notlar:

Standart kütüphanede nesne tutan sınıfların hepsi Allocator denen bir template parametresi almaktadır. Allocator kavramı ileride ele alınacağı için o konuya gelene kadar herhangi bir açıklama yapılmayacaktır.

```
template<class T, class Allocator = allocator<T> >  
class vector{  
    //...  
};
```

vector Sınıfının Üye Fonksiyonları

push_back ve pop_back Fonksiyonları:

vector' ün sonuna *push_back* fonksiyonuyla eleman eklenebilir, *pop_back* fonksiyonuyla da sonundaki eleman silinebilir.

```
void push_back(const T& x);  
void pop_back();
```

insert Fonksiyonları:

Sınıfın üç *insert* isimli fonksiyonu vardır.

```
iterator insert(iterator position, const T& x);
```

Bu fonksiyon belirli bir iterator pozisyonuna insert eder. Örneğin,

```
vector<int> a;
```

```
for (int i = 0; i < 10; ++i)  
    a.push_back(i);
```

```
a.insert(a.begin() + 5, 500);
```

Burada ekrana şunlar çıkacaktır:

```
0
1
2
3
4
500
5
6
7
8
9
```

vector sınıfının iteratorleri rasgele erişimli olduğu için "*a.begin()+5*" işlemi ilk elemanın iteratoründen beş sonraki elemanın iterator değerini verir. Fonksiyon insert işleminin yapıldığı iterator değerine geri döner.

```
void insert(iterator position, size_type n, const T& x);
```

size_type sınıf içerisinde default *size_t* türüne typedef edilmiş olan bir tür ismidir. Bu fonksiyon belirtilen iterator fonksiyonuna n tane eleman insert eder.

```
vector<int> a;
```

```
for (int i = 0; i < 10; ++i)
    a.push_back(i);
```

```
a.insert(a.begin() + 5, 3, 500);
```

```
template <class InputIterator>
```

```
void insert(iterator position, InputIterator first, InputIterator last);
```

Görüldüğü gibi bu fonksiyon template sınıfın template fonksiyonudur. Belirli bir iterator pozisyonuna başka bir nesne tutan sınıfın iki iterator tutan elemanlarını insert eder.

Anahtar Notlar:

Standart kütüphanede her zaman iki iterator ile bir aralık belirtildiği zaman ilk iteratorun belirttiği eleman bu aralığa dahildir, ama ikinci iteratorun belirttiği alan dahil değildir.

Anahtar Notlar:

Bir sınıfın template üye fonksiyonuna sahip olması durumuna ingilizce "member template" denilmektedir. Template bir sınıfta template üye fonksiyona sahip olabilir. Bu fonksiyonun dışarıda tanımlanması şöyle yapılır.

```
template <class T>
class Sample {
public:
    template <class A>
    void Func();
    // ...
};
```

```
template <class T>
template <class A>
void Sample<T>::Func()
{
    // ...
}
```

Template bir sınıfın içerisinde template olmayan bir sınıf template sınıfın template parametresini kullanabilir. Böyle bir sınıfın üye fonksiyonu dışarıda aşağıdaki gibi yazılır:

```
template <class T>
class A {
public:
    class B {
    public:
        void Func();
    };
};
```

```
template <class T>
void A<T>::B::Func()
{
    // ...
}
```

Template bir sınıfın içerisindeki template bir sınıfın template üye fonksiyonu olabilir. Bu template üye fonksiyon içerisinde söz konusu tüm template parametreleri kullanılabilir. Fonksiyonun dışarıda tanımlanması şöyle yapılmalıdır.

```
template <class T1>
class A {
public:
    template <class T2>
    class B {
    public:
        template <class T3>
        void Func();
    };
};
```

```

};

};

template <class T1>
template <class T2>
template <class T3>
void A<T1>::B<T2>::Func()
{
    // ...
}

```

Bu fonksiyonla başka bir sınıf içerisinde nesne tutan sınıfların tüm elemanları aşağıdaki işlemle vector'e insert edilmesi mümkündür.

```
using namespace std;
```

```

int main(void)
{
    vector<int> a;
    int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    a.insert(a.begin(), b, b + 10);

    for (int i = 0; i < 10; ++i)
        cout << a[i] << endl;

    return 0;
}

```

Sınıf Çalışması:

1' den 1000' e kadar sayılar içerisinde üçe tam bölünenleri bir *vector*' de toplayınız ve bunları yazdırınız.

```
using namespace std;
```

```

int main (int)
{
    vector<int> a;
    int i;

    for(i= 1; i<=1000; i++)
        if((i%3)==0)
            a.push_back(i);

    for(i = 0; i < a.size(); i++)
        cout<< a[i]<<endl;

    return 0;
}

```

```
}
```

erase Fonksiyonları:

erase fonksiyonları ile aradan eleman silinebilir. *erase* fonksiyonlarının iki biçimi vardır.

```
iterator erase(iterator position);
```

```
iterator erase(iterator first, iterator last);
```

Birinci fonksiyon iterator ile belirtilen elemanı, ikinci fonksiyon iki iterator ile belirtilen aralıktaki tüm elemanları siler.

Ayrıca vector sınıfında tüm elemanları silen bir *clear* fonksiyonu vardır. Yani:

```
a.clear();
```

ile

```
a.erase(a.begin(), a.end());
```

eşdeğerdir.

reserve Fonksiyonu:

reserve fonksiyonu kapasite arttırımı için kullanılmaktadır.

```
void reserve(size_type n);
```

Fonksiyonun parametresi yeni kapasite değerini belirtmektedir. Burada belirtilen değer o andaki kapasite değerinden büyük değilse yeniden tahsisat yapılmaz. Yani bu fonksiyon ile alan küçültmesi yapılamamaktadır.

reserve fonksiyonu özellikle vector'ün bir dizi gibi kullanıldığı durumlarda tercih edilmektedir. Yani biz uzunluğunu bildiğimiz bir dizi açmak isteyelim, fakat bu işlemde vector nesnesini kullanalım. Bu durumda işin başında *reserve* fonksiyonuyla bir kez tahsisat yapmak ve sonra [] operatörüyle vector'ü dizi gibi kullanmak mümkün olur. Örneğin;

```
vector<int> a;
```

```
a.reserve(100);
```

```
for(int i; i < 100; ++i)
```

```
    a[i] = i;
```

Şüphesiz yukarıdaki işlemlerle vector'ün size değeri 0' da kalacaktır. [] operatörü size değeriyle ilgili bir işlem yapmamaktadır. Sınıfın size değeri ancak *push_back* ve *insert* fonksiyonlarıyla değişebilmektedir.

Programcı vector'ü dizi gibi değilde normal bir biçimde kullanacak olsa bile yeniden tahsisat sayısını azaltmak için işin başında *reserve* fonksiyonunu kullanabilir. Örneğin;

```
vector<int> a;
```

```
a.reserve(100);
```

```
for (int i = 0; i < 100; ++i)
```

```
    a.push_back(i);
```

Burada *reserve* çağırması kaldırılabilir, fakat bu durumda daha fazla tahsisat yapılacaktır. Şüphesiz, programcı özel durumlar söz konusu olmadıktan sonra bu kadar ince düşünmek zorunda değildir.

resize Fonksiyonu:

resize fonksiyonu vector içerisinde tutulan gerçek elemanların sayısını değiştirmek için kullanılır. Örneğin, vector içerisinde 100 eleman bulunuyor olsun, o andaki kapasite miktarı da 300 olsun. Biz *resize* fonksiyonu ile size değerini 200'e yükseltirsek diziye 100 eleman eklemiş oluruz. tabi bu durumda bir kapasite yükseltmesi yapılmaz, fakat biz size değerini 500'e çıkartmak istersek *resize* fonksiyonu önce kapasiteyi 500'e göre ayarlayacak daha sonra 400 yeni eleman ekleyecektir. Eğer *resize* fonksiyonuyla size değerini düşürürsek bu işlemde kesinlikle kapasite etkilenmez. Sanki sondan eleman silmiş gibi oluruz.

```
void resize(size_type sz, T c= T());
```

Fonksiyonun ikinci parametresi eğer size değeri yükseltiliyorsa anlamlıdır ve doldurulacak elemanı belirtir. Görüldüğü gibi bu parametre hiç yazılmayabilir, bu durumda sınıflar için default başlangıç fonksiyonu ile elde edilen değer, normal türler için ise 0 değeri söz konusu olur. Standartlarda fonksiyonun yaptığı işlem şöyle açıklanmıştır.

```
if (sz > size())
    insert(end(), sz-size(), c);
else if (sz < size())
    erase(begin()+sz, end());
else
    ; //do nothing
```

vector Sınıfının Dizi gibi Kullanılması

vector'ün elemanlarının ardışıklığı standartların 2003 versiyonunda garanti altına alınmıştır. *v*, *T* türünden bir vector nesnesi olmak üzere *&v[0]* *vector*'ün tuttuğu elemanlara ilişkin dizinin başlangıç adresini vermektedir. Yani *&v[0]* ifadesi *T** türündendir ve *vector*'ün tuttuğu dizinin ilk elemanının adresini verir. Programcı bu adresten başlayarak *v.size()* kadar elemanı kullanabilir. Aslında bu adresten itibaren *v.capacity()* kadar eleman tahsis edilmiş durumdadır. Örneğin;

```
vector<int> v;
...
Func(&v[0], v.size());
```

yada örneğin;

```
vector<char> str;
str.reserve(20);
gets(&str[0]);
puts(&str[0]);
```


Özetle programcı `&v[0]` adresinden `v.capacity()` değerine kadar olan elemanları gönül rahatlığıyla kullanabilir. Tabi eğer vector'ü hem dizi gibi hem de vector gibi kullanacaksa `&v[0]` adresinden `v.size()` tane elemanı kullanması daha uygundur.

vector sınıfının tuttuğu elemanların ardışıklığı garanti edilmiştir. Fakat vector sınıfının iterator türünün bir gösterici olacağının garantisi yoktur. Şüphesiz, derleyicilerin büyük çoğunluğunda vector sınıfının iteratorleri bir adres türündendir. Fakat genel olarak bunun bir garantisi yoktur. Dolayısıyla `&v[i]` ifadesini, vector'ün i. elemanına ilişkin bir iterator ifadesi olarak kullanmamalıyız. vector'ün i. elemanına ilişkin iterator ifadesini "`v.begin() + i`" biçiminde ifade etmeliyiz.

Anahtar Notlar:

Nesne tutan fonksiyonların insert ve erase fonksiyonları genel olarak iteratorle işlem yaparlar. Örneğin, biz vector'ün iki elemanı arasını silmek istesek ilk elemanın dahil olduğu ikinci elemanın dahil olmadığı bir iterator aralığı vermemiz gerekir.

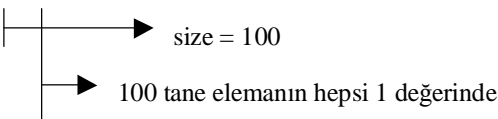
vector Sınıfının Başlangıç Fonksiyonları

vector sınıfının kopya başlangıç fonksiyonu dışında üç önemli başlangıç fonksiyonu vardır.

1. Default başlangıç fonksiyonu olarak kullanılan başlangıç fonksiyonu: Bu başlangıç fonksiyonu kullanılırsa size değeri 0 olur. capacity değeri hakkında bir şey söylenmemiştir. (Genellikle başlangıçta bir tahsisat yapılmamaktadır.)

2. Belirli bir elemandan n tane yaratmak için kullanılan başlangıç fonksiyonu: Bu başlangıç fonksiyonunda elemanın değerini belirten ikinci parametre default değer almıştır. Örneğin,

```
vector<int> v(100, 1);
```



The diagram illustrates a vector object. A vertical line represents the vector. A horizontal arrow points to the right from the top of the line, labeled "size = 100". Another horizontal arrow points to the right from the bottom of the line, labeled "100 tane elemanın hepsi 1 değerinde".

Örneğin;

```
vector<int> v(100, 1); //100 tane eleman default 0 değerine sahip.
```

3. Herhangi bir dizinin ya da nesne tutan sınıfın iki iterator arasından vector yapan iterator fonksiyonu: Örneğin;

```
list<int> l;
```

...

```
vector<int> v(l.begin(), l.end());
```

ya da örneğin;

```
int a[]={3, 5, 9, 1, 7}
```

```
vector<int> v(a, a+5);
```

Ters Yönde İlerleyen İteratorler

Ters yönde ilerleyen iteratörler (reverse iterators) ++ uygulandığında geriye giden, -- uygulandığında ileriye giden iteratörlerdir. Yani biz bir nesne tutan sınıfın son elemanına ilişkin ters yönde ilerleyen bir iteratör alsak, bu iteratörü artırarak artırarak ilk elemana kadar geriye doğru ilerleriz.

Ters yönde ilerleyen iteratörler ileriye doğru işlem yapan algoritmaların ters yönde de çalışmasını sağlamak için düşünülmüştür.

Nesne tutan sınıfların *reverse_iterator* isminde ters yönde ilerleyen bir iteratör tür ismi vardır. Örneğin, biz *vector* sınıfının ters yönde ilerleyen bir iteratörünü şöyle tanımlayabiliriz.

```
vector<int> :: reverse_iterator iter;
```

Nesne tutan sınıfların *rbegin* ve *rend* isimli üye fonksiyonları ters yönde ilerleyen iteratörler belirler.

```
iterator begin();
```

```
iterator end();
```

```
reverse_iterator rbegin();
```

```
reverse_iterator rend();
```

rbegin, son elemana ilişkin iteratör değerini; *rend* ise ilk elemandan bir önceki elemana ilişkin iteratör değerini verir. Örneğin;

```
vector<int> a;
```

```
for (int i = 0; i < 10; ++i)  
    a.push_back(i);
```

```
vector<int> b(a.rbegin(), a.rend());
```

```
for (int i = 0; i < b.size(); ++i)  
    cout << b[i] << endl;
```

Burada *b* vektörü, *a* vektörünün elemanlarının ters sırada oluşturulmasıyla elde edilmiştir.

```
#include <cstring>
```

```
#include <cctype>
```

```
#include <vector>
```

```
#include <list>
```

```
#include <windows.h>
```

```
using namespace std;
```

```
void Disp(const int *pArray, int size)  
{  
    for (int i = 0; i < size; ++i)  
        cout << pArray[i] << "\n";  
}
```

```
int main(void)  
{
```

```
vector<int> a;

for (int i = 0; i < 10; ++i)
    a.push_back(i);

vector<int>::reverse_iterator riter = a.rbegin();

cout << *riter << endl;
++riter;
cout << *riter << endl;

return 0;
}
```

const İteratorler

const iterator gösterdiği yer const olan bir gösterici gibi davranan iteratordür. Yani *iter* bir const iterator olmak üzere **iter* ile belirtilen nesneyi değiştiremeyiz. Hem normal iteratorler const olabilir, hem de ters yönde ilerleyen iteratorler const olabilir. Nesne tutan sınıfların *const_iterator* ve *const_reverse_iterator* isimli türleri de vardır.

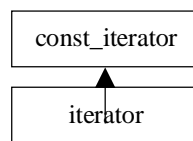
Bir iterator *const_iterator* türüne atanabilir, fakat bunun tersi mümkün değildir. Benzer biçimde bir *reverse_iterator* de *const_reverse_iterator* türüne atanabilir ama tersi mümkün değildir.

Nesne tutan sınıfların const iterator veren üye sınıfları yoktur. Zaten böyle bir şeye gereksinim de duyulmaz. const iterator için yine *begin*, *end* ve *rbegin*, *rend* fonksiyonlarını kullanabiliriz.

iterator ve *reverse_iterator* türleri farklı türlerdir, yani birbirlerine atanamazlar.

Anahtar Notlar:

Pekçok derleyici sisteminde iterator bir sınıf ile temsil edilecekse const_iterator sınıfından türetilir.



Böylece iterator, const_iterator türüne atanabilmekte fakat tersi mümkün olmamaktadır. citer, const bir iterator olsun;

**citer = x;*

işleminin engellenmesi gerekir. Bu engelleme şöyle yapılabilir:

```
const T &operator *()
{
    //...
}
```

Bu durumda tipik bir nesne tutan sınıfın dört çeşit iterator türü vardır.

1. **iterator**
2. **const_iterator**
3. **reverse_iterator**
4. **const_reverse_iterator**

Ayrıca sınıfların yine iterator veren dört üye fonksiyonu vardır.

1. **iterator begin();**
2. **iterator end();**
3. **reverse_iterator rbegin();**
4. **reverse_iterator rend();**

const iteratorler tamamen const göstericilere benzerler. Yani const göstericiler neden kullanılırlarsa const iteratorler de o nedenlerle kullanılırlar.

Normal diziler, rasgele erişimli iterator gereksinimlerini karşılarlar. Fakat ters yönde ilerleyen iterator gereksinimlerini karşılayamazlar.

Fonksiyon Nesneleri (Function Object/Functor)

Fonksiyon nesneleri (function object/functor), fonksiyon gibi kullanılan sınıf nesneleridir. Bilindiği gibi bir sınıf nesnesinin fonksiyon gibi kullanılabilmesi için fonksiyon çağırma operator fonksiyonunun yazılmış olması gerekir. Örneğin;

```
X f;  
f();           //eşdeğeri:  f.operator()();
```

Bir template fonksiyon bir fonksiyon parametresi istediğinde biz ona gerçek bir fonksiyonu verebileceğimiz gibi bir fonksiyon nesnesini de verebiliriz. Örneğin,

```
template <typename T>  
class Functor {  
public:  
    void operator()(T a)  
    {  
        cout << a << endl;  
    }  
};  
...  
int a[] = {1, 2, 3, 4, 5};
```

```
for_each(a, a + 5, Functor<int>());
```

Görüldüğü gibi burada fonksiyonun son parametresinde bir fonksiyon adresi değil bir sınıf nesnesi verilmiştir. *foreach* fonksiyonu şöyle yazılmış fonksiyondur.

```
template <typename InputIterator, typename F>  
F myfor_each(InputIterator first, InputIterator last, F f)  
{
```

```
for (; first != last; ++first)
    f(*first);
return f;
}
```

Genel olarak standart kütüphanedeki algoritma dediğimiz global template fonksiyonların bir fonksiyon parametresi istemesi durumunda bir fonksiyon değil bir fonksiyon nesnesi kullanmak daha etkindir.

Karşılaştırma Fonksiyonları

Standart kütüphanedeki bazı fonksiyonlar karşılaştırma amacıyla kullanacakları bir fonksiyonu parametre olarak alırlar. Böyle fonksiyonlara *karşılaştırma fonksiyonları (predicate)* denilmektedir. Bu tür fonksiyonların geri dönüş değerleri bool türünden ya da bool türüne doğrudan dönüştürülebilir türde olması gerekmektedir. Karşılaştırma fonksiyonları, *tek parametrelili (uniary predicate)* ya da *çift parametrelili (binary predicate)* olabilir.

karşılaştırma fonksiyonu alan bir fonksiyona örnek olarak *count_if* fonksiyonu verilebilir. *count_if* fonksiyonu aşağıdaki gibi tanımlanmış bir fonksiyondur.

```
template <typename InputIterator, unaryPredicate f>
std::size_t count_if(InputIterator first, InputIterator last, UnaryPredicate f)
{
    std::size_t n = 0;
    for (; first != last; ++first)
        if(f(*first))
            ++n;
    return n;
}
```

count_if fonksiyonuna karşılaştırma amacıyla ya bir fonksiyon ya da fonksiyon gibi davranan bir sınıf nesnesi verilmelidir.

ALGORİTMALAR

Standart kütüphanedeki global template fonksiyonlara algoritma denilmektedir. Bu fonksiyonlar *<algorithm>* başlık dosyasında tanımlanmıştır. Algoritmik fonksiyonlar standartlarda 25. bölümde ele alınmaktadır. Standartlarda algoritmalar tek tek ele alınıp açıklanmış ve algoritma karmaşıklıklarının ne olması gerektiği belirtilmiştir.

Algoritmalar çeşitli yazarlar tarafından değişik biçimde sınıflandırılmaktadır. Kaba bir sınıflandırma şöyle olabilir:

1. Veri yapısı üzerinde değişiklik yapmayan algoritmalar(Nonmodifying Algorithms)
2. Veri yapısı üzerinde değişiklik yapan algoritmalar (Modifying Algorithms)
3. Silme işlemleri yapan algoritmalar (Removing Algorithms)
4. Nesnelerin yerlerini değiştiren algoritmalar (Mutating Algorithms)

5. Sort işlemi yapan algoritmalar (Sorting Algorithms)
6. Sayısal işlem yapan algoritmalar (Numeric Algorithms)

Veri Yapısı Üzerinde Değişiklik Yapmayan Algoritmalar(Nonmodifying Algorithms)

for_each Fonksiyonu:

Bu fonksiyon iki iterator arasındaki elemanlar için belirlenen fonksiyonu çağırır.

```
template<class InputIterator, class Function>
```

```
Function for_each(InputIterator first, InputIterator last, Function f);
```

Fonksiyonun son parametresinde belirtilen fonksiyonun iteratorün belirttiği türden bir parametresi olmalıdır. Bu fonksiyonun parametresi referans olabilir. Fonksiyon son parametresinde belirtilen tür ile geri dönmektedir.

```
#include <iostream>
```

```
#include <string>
```

```
#include <cstring>
```

```
#include <cctype>
```

```
#include <vector>
```

```
#include <algorithm>
```

```
#include <list>
```

```
#include <windows.h>
```

```
using namespace std;
```

```
class Functor {
```

```
public:
```

```
    void operator () (int a)
```

```
    {
```

```
        cout << a << endl;
```

```
    }
```

```
};
```

```
void Square(int &a)
```

```
{
```

```
    a = a * a;
```

```
}
```

```
void Disp(int a)
```

```
{
```

```
    cout << a << endl;
```

```
}
```

```
int main(void)
```

```
{
```

```
    int a[] = {1, 112, 35, 42, 5};
```

```
    for_each(a, a + 5, Square);
```

```
    for_each(a, a + 5, Disp);
```

```
    return 0;
}
```

Örnek2

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>

using namespace std;

class Functor {
public:
    Functor() : m_total(0)
    {}
    void operator () (int a)
    {
        m_total += a;
    }
    void Disp() const
    {
        cout << m_total << endl;
    }
private:
    int m_total;
};

int main(void)
{
    int a[] = {1, 112, 35, 42, 5};
    Functor result;

    result = for_each(a, a + 5, Functor());
    result.Disp();

    return 0;
}
```

Sınıf Çalışması:

for_each fonksiyonunu kullanarak bir dizide istenilen bir elemanı arayan programı yazınız. Aranacak eleman dışarıdan değiştirilebilmelidir.

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>

using namespace std;

class Functor {
public:
    Functor(int a) : m_a(a), m_p(0)
    {}
    Functor()
    {}
    void operator () (int &a)
    {
        if (a == m_a)
            m_p = &a;
    }
    void Disp() const
    {
        if (m_p != 0)
            cout << "buldu: " << *m_p << endl;
        else
            cout << "bulamadi!..\n";
    }
private:
    int m_a;
    int *m_p;
};

int main(void)
{
    int a[] = {1, 112, 35, 42, 5};
    Functor result;

    result = for_each(a, a + 5, Functor(35));
    result.Disp();

    return 0;
}
```



```
}  
  
vector olursa;  
#include <iostream>  
#include <string>  
#include <cstring>  
#include <cctype>  
#include <vector>  
#include <algorithm>  
#include <list>  
#include <windows.h>  
  
using namespace std;  
  
class Functor {  
public:  
    Functor(int a) : m_a(a), m_p(0)  
    {}  
    Functor()  
    {}  
    void operator () (int &a)  
    {  
        if (a == m_a)  
            m_p = &a;  
    }  
    void Disp() const  
    {  
        if (m_p != 0)  
            cout << "buldu: " << *m_p << endl;  
        else  
            cout << "bulamadi!..\n";  
    }  
private:  
    int m_a;  
    int *m_p;  
};  
  
int main(void)  
{  
    int a[] = {1, 112, 35, 42, 5};  
    vector<int> v(a, a + 5);  
    Functor result;  
  
    result = for_each(v.begin(), v.end(), Functor(5));  
    result.Disp();  
  
    return 0;  
}
```

```
}
```

find Fonksiyonu:

Bu fonksiyon veri yapısı içerisinde bir elemanı aramak için kullanılır. Eleman bulunursa elemanın veri yapısı içerisindeki iterator değeriyle, eleman bulunamazsa son elemandan bir sonraki elemanın iterator değeriyle geri dönlür.

```
template<class InputIterator, class T>
```

```
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

Örnek;

```
#include <iostream>
```

```
#include <string>
```

```
#include <cstring>
```

```
#include <cctype>
```

```
#include <vector>
```

```
#include <algorithm>
```

```
#include <list>
```

```
#include <windows.h>
```

```
using namespace std;
```

```
int main(void)
```

```
{
```

```
    int a[] = {1, 112, 35, 42, 5};
```

```
    vector<int> v(a, a + 5);
```

```
    vector<int>::iterator result;
```

```
        result = find(v.begin(), v.end(), 112);
```

```
    if (result != v.end())
```

```
        cout << "buldu:" << *result << endl;
```

```
    else
```

```
        cout << "bulamadi\n";
```

```
    return 0;
```

```
}
```

find fonksiyonunun karşılaştırmada == operatörünü kullandığı standartlarda garanti altına alınmıştır. Yani eğer veri yapısı sınıflardan oluşuyorsa bizim bu sınıflar türünden == operatör fonksiyonu yazmamız gerekir.

```
#include <iostream>
```

```
#include <string>
```

```
#include <cstring>
```

```
#include <cctype>
```

```
#include <vector>
```

```
#include <algorithm>
```

```
#include <list>
```

```
#include <windows.h>
```

```
using namespace std;

class Sample {
public:
    Sample(int a) : m_a(a)
    {}
    bool operator ==(const Sample &r)
    {
        return m_a == r.m_a;
    }
    friend ostream &operator <<(ostream &os, const Sample &r);
private:
    int m_a;
};

ostream &operator <<(ostream &os, const Sample &r)
{
    os << r.m_a;

    return os;
}

int main(void)
{
    vector<Sample> v;

    for (int i = 0; i < 100; ++i)
        v.push_back(Sample(i));

    vector<Sample>::iterator result;

    result = find(v.begin(), v.end(), Sample(145));

    if (result != v.end()) {
        cout << *result << endl;
    }
    else {
        cout << "bulamadi!..\n";
    }
    return 0;
}
```

Anahtar Notlar:

Standart kütüphanedeki bir template fonksiyonun iterator üzerinde hangi operatörleri uyguladığı iteratorün türüne göre anlaşılabilir. Örneğin, iteratorün türü girdi iteratorü ise fonksiyonun iterator üzerinde örneğin -- operatörünü kullanmadığını biliriz. Ancak fonksiyonların iteratorün gösterdiği tür üzerinde hangi operatörleri uyguladığı nasıl anlaşılacaktır? İşte standartlarda iki olasılık üzerinde durulmuştur.

1. *Yalnızca == operatörünü kullanan fonksiyonlar (Equality comparable)*
2. *Yalnızca < operatörünü kullanan fonksiyonlar (Less than comparable)*

Aslında yalnızca < operatörü ile tüm karşılaştırmalar yapılabilir. İşte standart kütüphanede pekçok fonksiyon ya yalnızca == ya da yalnızca < operatörü kullanarak belirlenmişlerdir. Böylece programcının sınıf için tek bir operatör fonksiyonu yazması yeterli olmaktadır.

Anahtar Notlar:

Standart kütüphane içerisindeki bazı fonksiyonların isimlendirilmesinde _if son eki kullanılmıştır. Örneğin, find isimli bir fonksiyonun yanı sıra find_if isimli bir fonksiyon daha vardır. Bu _if son ekli fonksiyonlar karşılaştırma için operatör değil, bir fonksiyon (predicate) kullanırlar. Standartlarda bu fonksiyonun hangi karşılaştırmalarda kullanılacağı belirtilmiştir. Genel olarak bu fonksiyonlar ya == karşılaştırmalarında (equality comparable) ya da < karşılaştırmalarında (less than comparable) kullanılmaktadır.

find_if Fonksiyonu:

Bu fonksiyon find fonksiyonunun karşılaştırma fonksiyonu alan biçimidir. Karşılaştırma fonksiyonu == durumunu anlamak için kullanılacaktır. Yani karşılaştırma fonksiyonunun bir parametresi olmalı bu fonksiyon eğer programcı aranan şeyin bulunduğu inanılıyorsa true değerine, bulunmadığına inanılıyorsa false değerine geri dönmelidir. Örneğin, Visual C++ 7.0 derleyicisinde bu template fonksiyon şöyle yazılmıştır.

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

```
template<class _InIt, class _Pr>
inline find_if(InIt _First, _InIt _Last, _Pr _Pred)
{
    for(; _First != _Last; ++_First)
        if(_Pred(*_First))
            break;
    return (_First);
}
```

Örnek;

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>
```

```
using namespace std;
```

```
class Sample {
public:
    Sample(int a) : m_a(a)
    {}
    bool operator ==(const Sample &r)
    {
        return m_a == r.m_a;
    }
    friend ostream &operator <<(ostream &os, const Sample &r);
    friend bool Compare(const Sample &r);
private:
```

```
    int m_a;
};
```

```
ostream &operator <<(ostream &os, const Sample &r)
{
    os << r.m_a;

    return os;
}
```

```
bool Compare(const Sample &r)
{
    if(r.m_a == 100)
        return true;
    return false;
}

int main(void)
{
    vector<Sample> v;

    for (int i = 0; i < 105; ++i)
        v.push_back(Sample(i));

    vector<Sample>::iterator result;

    result = find_if(v.begin(), v.end(), Compare);

    if(result != v.end()) {
        cout << *result << endl;
    }
    else {
        cout << "bulamadi!..\n";
    }

    return 0;
}
```

Örnek;

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>
```

```
using namespace std;
```

```
class Sample {
public:
    Sample(int a) : m_a(a)
    {}
    bool operator ==(const Sample &r)
    {
        return m_a == r.m_a;
```

```
    }  
    friend ostream &operator <<(ostream &os, const Sample &r);  
private:  
    int m_a;  
};
```

```
class Functor {  
public:  
    Functor(const Sample &s) : m_s(s)  
    {}  
    bool operator()(const Sample &s)  
    {  
        return m_s == s;  
    }  
private:  
    Sample m_s;  
};  
ostream &operator <<(ostream &os, const Sample &r)  
{  
    os << r.m_a;  
  
    return os;  
}
```

```
int main(void)  
{  
    vector<Sample> v;  
  
    for (int i = 0; i < 105; ++i)  
        v.push_back(Sample(i));  
  
    vector<Sample>::iterator result;  
  
    result = find_if(v.begin(), v.end(), Functor(Sample(100)));  
  
    if (result != v.end()) {  
        cout << *result << endl;  
    }  
    else {  
        cout << "bulamadi!..\n";  
    }  
  
    return 0;  
}
```

Örnek;

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>
```

```
using namespace std;
```

```
class Functor {
public:
    Functor(const string &r) : m_s(r)
    {
        for (int i = 0; i < m_s.size(); ++i)
            m_s[i] = toupper(m_s[i]);
    }
    bool operator()(string &s)
    {
        string temp(s);

        for (int i = 0; i < m_s.size(); ++i)
            temp[i] = toupper(m_s[i]);

        return temp == m_s;
    }
private:
    string m_s;
};
```

```
int main(void)
{
    vector<string> s;

    s.push_back("ali");
    s.push_back("veli");
    s.push_back("selami");

    vector<string>::iterator result;

    result = find_if(s.begin(), s.end(), Functor("ALI"));

    if(result != s.end())
        cout << "buldu: " << *result;
    else
```



```

        cout << "bulamadi\n";

    return 0;
}

```

find_first_of Fonksiyonları

Bu fonksiyonlar bir iterator aralığı ile verilmiş dizide başka bir iterator aralığı verilmiş bir dizinin içerisindeki herhangi bir elemanı aralar. Bulurlarsa ilk buldukları yerin iterator değeri ile bulamazlarsa aranan dizinin son iteratoründen sonraki iterator değeriyle geri dönerler. Fonksiyonun normal ve iki parametrelili karşılaştırma fonksiyonu alan biçimleri vardır.

```

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1

```

```

find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2);

```

```

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1

```

```

find_first_of(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2,
              ForwardIterator2 last2, BinaryPredicate pred);

```

Örneğin,

```

#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>

```

```

using namespace std;

```

```

class Functor {

```

```

public:

```

```

    Functor(int x) : m_x(x)

```

```

    {}

```

```

    bool operator() (int a, int b)

```

```

    {

```

```

        return a - b == m_x;

```

```

    }

```

```

private:

```

```

    int m_x;

```

```

};

```

```

int main(void)

```

```

{

```

```

    int a[10] = {3, 6, 8, 8, 5, 9, 2, 6, 8, 4};

```

```
int b[] = {9, 8, 6};
vector<int> v(a, a + 10);
vector<int>::iterator iter;

iter = find_first_of(v.begin(), v.end(), b, b + 3);

if (iter != v.end())
    cout << *iter << endl;
else
    cout << "cannot find!..\n";

return 0;
}

#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>

using namespace std;

bool Compare(char a, char b)
{
    return a == b;
}

int main(void)
{
    string s1 = "01234567";
    char s2[] = "234";
    string::iterator iter;

    iter = find_first_of(s1.begin(), s1.end(), s2, s2 + 3, Compare);
    if (iter != s1.end()) {
        string result (iter, s1.end());
        cout << "Buldu: " << result;
    }
    else
        cout << "bulamadi!..\n";
    return 0;
}
```

find_end Fonksiyonları

Bu fonksiyonlar tamamen *find_first_of* fonksiyonu gibidir. Fakat dizilim birden fazla yerde varsa ilk bulunan değil son bulunan yerin iterator değeriyle geri dönerler.

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);
```

```
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);
```

adjacent_find Fonksiyonları

Bu fonksiyonlar iterator aralığıyla verilmiş olan bir dizilimde yanyana iki eleman aynı ise ilk yanyana iki elemanın bulunduğu iterator değerini elde eder. Eğer, yanyana aynı iki elemandan yoksa son elemandan bir sonraki iterator değeriyle geri dönülür.

```
template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);
```

```
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                            BinaryPredicate pred);
```

Fonksiyonun ikinci biçimi yanyana elemanın eşitliğini sorgulamak için karşılaştırma fonksiyonu almaktadır.

Sınıf Çalışması:

Fonksiyon nesnesi kullanarak int türden bir vektör içerisinde yanyana iki eleman arasında +n fark bulunan ilk ikiliyi tespit ediniz.

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>
```

```
using namespace std;
```

```
class Functor {
public:
    Functor(int x) : m_x(x)
    {}
}
```

```
bool operator() (int a, int b)
{
    return a - b == m_x;
}
private:
    int m_x;
};

int main(void)
{
    int a[10] = {3, 6, 8, 3, 5, 9, 2, 6, 8, 4};
    vector<int> v(a, a + 10);
    vector<int>::iterator result;

    result = adjacent_find(v.begin(), v.end(), Functor(5));
    if (result != v.end())
        cout << "found: " << *result << endl;
    else
        cout << "cannot find!..\n";
    return 0;
}
```

count Fonksiyonları

Bu fonksiyonlar bir dizilimde belirli bir elemandan kaç tane bulunduğunu elde etmektedir.

```
template<class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);
```

count_if fonksiyonu, *count* fonksiyonunun karşılaştırma fonksiyonunu alan biçimidir.

```
template<class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);
```

Anahtar Notlar:

Nesne tutan sınıflarda standart *difference_type* isiminde bir *typedef* ismi vardır, *difference_type* işaretli olmak zorundadır. Fakat hangi türe *typedef* edileceği derleyiciyi yazanlara bırakılmıştır. Tipik olarak bu tür iki iterator arasındaki eleman sayısını temsil etmek için düşünülmüştür.

equal Fonksiyonları

Bu fonksiyonlar iki dizilimin birbirine eşit olup olmadığına bakmaktadır. Yani kabaca iki diziliminde karşılıklı elemanları birbirine eşitse dizim birbirine eşittir.

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);
```

Görüldüğü gibi ikinci dizilim için bir aralık değil, yalnızca başlangıç iterator değeri verilmiştir. İkinci dizilimin bitiş iterator değerinin verilmesine gerek yoktur. fonksiyon elemanları karşılıklı olarak == operatörü ile karşılaştırmakta eğer tüm elemanlar birbirine eşitse true değerine geri dönmektedir.

search Fonksiyonları

Bu fonksiyonlar bir dizilim içerisinde başka bir dizilimi aramak için kullanılır.

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
search(ForwardIterator1 first1, ForwardIterator1 last1,
       ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
search(ForwardIterator1 first1, ForwardIterator1 last1,
       ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);
```

Fonksiyon birinci iterator aralığıyla belirtilen dizilimde ikinci iterator aralığıyla belirtilen dizilimi arar. Bulursa bulduğu yerin iterator değeriyle bulamazsa aranan dizilimdeki son elemandan bir sonraki elemanın iterator değeriyle geri döner.

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>
```

```
using namespace std;
```

```
class Functor {
public:
    Functor(int x) : m_x(x)
    {}
    bool operator() (int a, int b)
    {
```

```

        return a - b == m_x;
    }
private:
    int m_x;
};
int main(void)
{
    int a[10] = {3, 6, 8, 8, 5, 9, 2, 6, 8, 4};
    int b[] = {8, 8, 5};
    vector<int> v(a, a + 10);
    vector<int>::iterator iter;

    iter = search(v.begin(), v.end(), b, b + 3);

    if (iter != v.end())
        cout << *iter << endl;
    else
        cout << "cannot find!..\n";
    return 0;
}

```

3. Veri Yapısı Üzerinde Değişiklik Yapan Fonksiyonlar

copy Fonksiyonu:

Bu fonksiyon bir dizilimdeki iki iterator arasını başka bir veri yapısına kopyalar.

```

template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result);

```

Bu fonksiyonla farklı türden nesne tutan sınıflar arasında kopyalama yapabiliriz. Şüphesiz bu işlemden önce hedef veri yapısında yeterli alanın tahsis edilmiş olması gerekmektedir. Fonksiyon kopyalamanın yapıldığı dizilimde kopyalanan elemanlardan sonraki iterator değerine geri döner.

```

#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>

```

```
using namespace std;
```

```

class Functor {
public:
    Functor(int x) : m_x(x)
    {}
}

```

```
bool operator() (int a, int b)
{
    return a - b == m_x;
}
private:
    int m_x;
};

template <typename InputIterator, typename OutputIterator>
OutputIterator mycopy(InputIterator first, InputIterator last, OutputIterator result)
{
    while (first != last) {
        *first = *result;
        ++first;
        ++result;
    }
    return result;
}

int main(void)
{
    int a[10];
    vector<int> v;

    for (int i = 0; i < 10; ++i)
        v.push_back(i);
    mycopy(v.begin(), v.end(), a);

    if (equal(v.begin(), v.end(), a))
        cout << "success\n";
    else
        cout << "failed!..\n";
    return 0;
}
```

Anahtar Notlar:

++ ve -- operatörlerine izin veren iterator türleri bu operatörlerin hem ön ek hem de son ek biçimlerini bulundurmaktadır. Ancak son ek ++ ve -- operatörlerini yazımı zaman bakımından daha maaliyetlidir.

copy_backward Fonksiyonu:

Bu fonksiyon tıpkı *copy* fonksiyonu gibi kopyalama yapar. Fakat kopyalamayı sondan başa doğru gerçekleştirmektedir.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2
copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
    BidirectionalIterator2 result);
```

Fonksiyonun parametreleri sondan başa girilmelidir. Örneğin;

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <list>
#include <windows.h>
```

```
using namespace std;
```

```
class Functor {
public:
    Functor(int x) : m_x(x)
    {}
    bool operator() (int a, int b)
    {
        return a - b == m_x;
    }
private:
    int m_x;
};
```

```
template <typename InputIterator, typename OutputIterator>
OutputIterator mycopy(InputIterator first, InputIterator last, OutputIterator result)
{
    while (first != last) {
        *first = *result;
        ++first;
        ++result;
    }
    return result;
}

int main(void)
{
    int a[5] = {1, 2, 3, 4, 5};
    int b[5];
    copy_backward(a, a + 5, b + 5);
}
```



```
    for (int i = 0; i < 5; ++i)
        cout << b[i] << endl;
    return 0;
}
```

transform Fonksiyonları:

Bu fonksiyonlar en çok kullanılan algoritmalarındandır. Bir dizilimdeki elemanlar üzerinde belirli işlemleri uygulayarak onları güncellemek için kullanılır. İki biçimi vardır:

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator
transform(InputIterator first, InputIterator last,
OutputIterator result, UnaryOperation op);
```

Fonksiyon bir iterator aracılığıyla alınan dizinin içindeki elemanlara bir işlem uygulayarak bir dizilime kopyalar. Kaynak dizilimle hedef dizilim aynı olabilir. Fonksiyonun son parametresinin dizilime ilişkin türden parametre ve geri dönüş değerine sahip bir fonksiyon olması gerekmektedir. Fonksiyon kopyalamanın yapıldığı dizide kopyalamanın sonundaki elemanın iterator değeriyle geri döner.

transform fonksiyonunu şöyle yazabiliriz.

```
using namespace std;
```

```
class Functor {
public:
    Functor(int x) : m_x(x)
    {}
    bool operator() (int a, int b)
    {
        return a - b == m_x;
    }
private:
    int m_x;
};

template <typename InputIterator, typename OutputIterator>
OutputIterator mycopy(InputIterator first, InputIterator last, OutputIterator result)
{
    while (first != last) {
        *first = *result;
        ++first;
        ++result;
    }
    return result;
}

int Square(int a)
{

```

```
        return a * a;
    }
    template<class InputIterator, class OutputIterator, class UnaryOperation>
    OutputIterator mytransform(InputIterator first, InputIterator last,
                              OutputIterator result, UnaryOperation op)
    {
        while (first != last) {
            *result = op(*first);
            ++result;
            ++first;
        }
    }
    int main(void)
    {
        int a[6] = {1, 2, 3, 4, 5};
        int b[6];
        transform(a, a + 5, b, Square);
        for (int i = 0; i < 5; ++i)
            cout << b[i] << endl;
        return 0;
    }
```

transform fonksiyonunun ikinci biçimi iki dizilimin karşılıklı elemanlarını işleme sokarak başka bir dizilime aktarır.

```
template<class InputIterator1, class InputIterator2, class OutputIterator, class
BinaryOperation>
OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, OutputIterator result,
          BinaryOperation binary_op);
```

Burada fonksiyonun son parametresi iki parametrelili bir fonksiyondur.

Sınıf Çalışması:

10 elemanlı int türden a ve b dizileri alınır. Bu diziden karşılıklı elemanları en küçük olanı kullanarak *transform* fonksiyonuyla bir c dizisi oluşturulur.

```

#include <cstdio>
#include <iostream>
#include <vector>
#include <algorithm>

int smaller(int a, int b)
{
    if (a < b)
        return a;
    return b;
}

using namespace std;

int main()
{
    int a[5] = {3, 9, 7, 5};
    int b[5] = {8, 6, 5, 4};
    int c[5];

    /*
     * iki dizinin ortak en küçük elemanlarını 3. diziye yazar.
     */
    transform(a, a + 5, b, c, smaller);
    for (int i = 0; i < 5; i++)
        cout << c[i] << endl;

    return 0;
}

```

Standart Kütüphanedeki Fonksiyon Nesneleri

Standart kütüphanede bazı temel işlemleri yapmak için çeşitli fonksiyon sınıfları mevcuttur. Böylelikle programcının bazı basit işlemler için ayrıca bir fonksiyon ya da fonksiyon sınıfı yazmasına gerek kalmaz. Standart fonksiyon sınıfları <utility> başlık dosyasında tanımlanmışlardır.

Standart kütüphanedeki *plus*, *mines* gibi fonksiyon sınıfları toplama çıkartma gibi temel işlemleri yapmaktadır. Örneğin, *plus* gerçekte template bir sınıftır, bu sınıfın iki parametre alan bir fonksiyon çağırma operatör fonksiyonu vardır. Örneğin biz

plus <int>()

ifadesini bir algoritmaya *f* parametresi olarak geçirelim. Algoritma içerisinde *f(a, b)* gibi bir çağırma aslında *plus* sınıfının fonksiyon çağırma operatör fonksiyonunun çağırılması anlamına gelecektir. *plus* sınıfının iki parametrelili fonksiyon çağırma operatör fonksiyonu geri dönüş değeri olarak bu iki parametresinin toplamını vermektedir. Bu durumda örneğin:

transform(a, a + 5, b, c, plus<int>());

a' nın ve b' nin karşılıklı elemanları toplanarak c' ye yazılacaktır.

Standart kütüphanedeki fonksiyon nesne sınıfları sembolik olarak *unary_function* ve *binary_function* sınıflarından türetilmiştir. Bu sınıflar veri elemanına sahip olmayan yalnızca typedef isimlerine sahip olan algılamayı güçlendirmek için kullanılan sınıflardır.

```
template <class Arg, class Result>
struct unary_function {
typedef Arg argument_type;
typedef Result result_type;
};
```

```
template <class Arg1, class Arg2, class Result>
struct binary_function {
typedef Arg1 first_argument_type;
typedef Arg2 second_argument_type;
typedef Result result_type;
};
```

Bu durumda örneğin standartlarda *plus* sınıfı şöyle bildirilmiştir.

```
template <class T> struct plus : binary_function<T,T,T> {
T operator()(const T& x, const T& y) const;
};
```

Standart kütüphanede bulunan temel fonksiyon nesne sınıfları şunlardır.

plus:

```
template <class T> struct plus : binary_function<T,T,T> {
T operator()(const T& x, const T& y) const;
};
```

mines:

```
template <class T> struct minus : binary_function<T,T,T> {
T operator()(const T& x, const T& y) const;
};
```

multiplies:

```
template <class T> struct multiplies : binary_function<T,T,T> {
T operator()(const T& x, const T& y) const;
};
```

divides:

```
template <class T> struct divides : binary_function<T,T,T> {
T operator()(const T& x, const T& y) const;
};
```

modulus:

```
template <class T> struct modulus : binary_function<T,T,T> {
T operator()(const T& x, const T& y) const;
```

```
};
```

negate:

```
template <class T> struct negate : unary_function<T,T> {  
    T operator()(const T& x) const;  
};
```

equal_to, not_equal_to, less, greater_equal, less_equal, greater:

```
template <class T> struct ... : binary_function<T,T,bool> {  
    bool operator()(const T& x, const T& y) const;  
};
```

logical_and, logical_or:

```
template <class T> struct logical_xxx : binary_function<T,T,bool> {  
    bool operator()(const T& x, const T& y) const;  
};
```

logical_not:

```
template <class T> struct logical_not : unary_function<T,bool> {  
    bool operator()(const T& x) const;  
};
```

örnek:

```
transform(a, a + 5, a, logical_not<int>());
```

replace Fonksiyonları

replace fonksiyonlarının amacı dizilimdeki çeşitli değerleri başka değerlerle değiştirmektir, çeşitli biçimleri vardır.

```
template<class ForwardIterator, class T>  
void replace(ForwardIterator first, ForwardIterator last, const T& old_value,  
             const T& new_value);
```

Bu fonksiyon bir iterator aralığı ile belirtilen dizilimdeki belirli değerleri başka değerlerle yer değiştirir.

```
int main(void)
{
    vector<string> v;

    v.push_back("ali");
    v.push_back("veli");
    v.push_back("mehmet");
    v.push_back("ali");
    v.push_back("salih");

    replace(v.begin(), v.end(), string("ali"), string("recep"));

    for (int i = 0; i < v.size(); ++i)
        cout << v[i] << endl;

    return 0;
}
```

Örnek;

```
using namespace std;
```

```
class Functor {
```

```
public:
```

```
    Functor(int x) : m_x(x)
```

```
    {}
```

```
    bool operator() (int a, int b)
```

```
    {
```

```
        return a - b == m_x;
```

```
    }
```

```
private:
```

```
    int m_x;
```

```
};
```

```
template <typename InputIterator, typename OutputIterator>
```

```
OutputIterator mycopy(InputIterator first, InputIterator last, OutputIterator result)
```

```
{
```

```
    while (first != last) {
```

```
        *first = *result;
```

```
        ++first;
```

```
        ++result;
```

```
    }
```

```
    return result;
```

```
}
```

```
int Square(int a)
```

```
{
```

```
    return a * a;
```

```
}
```

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator mytransform(InputIterator first, InputIterator last,
                          OutputIterator result, UnaryOperation op)
```

```
{
    while (first != last) {
        *result = op(*first);
        ++result;
        ++first;
    }
}
```

```
template <class T>
```

```
class Complex {
```

```
public:
```

```
    Complex(T real, T imag) : m_real(real), m_imag(imag)
    {}
```

```
    bool operator ==(const Complex &r) const
    {
```

```
        return m_real == r.m_real && m_imag == r.m_imag;
```

```
    }
```

```
    template <class M>
```

```
    friend ostream &operator << (ostream &os, const Complex<M> &r);
```

```
private:
```

```
    T m_real, m_imag;
```

```
};
```

```
template <class T>
```

```
ostream &operator << (ostream &os, const Complex<T> &r)
```

```
{
    os << r.m_real << "+" << r.m_imag << "i";
```

```
    return os;
```

```
}
```

```
int main(void)
```

```
{
```

```
    vector<Complex<int> > v;
```

```
    v.push_back(Complex<int>(10, 2));
```

```
    v.push_back(Complex<int>(11, 4));
```

```
    v.push_back(Complex<int>(12, 2));
```

```
    v.push_back(Complex<int>(10, 2));
```

```
    v.push_back(Complex<int>(10, 2));
```

```
    replace(v.begin(), v.end(), Complex<int>(10, 2), Complex<int>(99, 99));
```

```
for (int i = 0; i < v.size(); ++i)
    cout << v[i] << endl;
return 0;
}
```

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last, Predicate pred,
    const T& new_value);
```

Bu biçimde == karşılaştırması tek parametre alan bir fonksiyona bırakılmıştır.

Anahtar Notlar:

copy son ekli fonksiyonlar elde edilen sonucun başka bir dizilime kopyalanacağı anlamına gelir. Örneğin, replace fonksiyonu dizilimdeki x değerlerini y yapmaktadır. O halde replace_copy fonksiyonu x değerlerini y yaparak başka bir dizilime kopyalar. Ayrıca bir de copy_if son eki vardır. Bu fonksiyonlar hem karşılaştırma fonksiyonu alırlar hem de sonucu başka bir dizilime kopyalarlar.

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator
replace_copy(InputIterator first, InputIterator last, OutputIterator result,
    const T& old_value, const T& new_value);
```

```
template<class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator
replace_copy_if(Iterator first, Iterator last, OutputIterator result,
    Predicate pred, const T& new_value);
```

replace_copy_if dizilimdeki bir değeri karşılaştırma fonksiyonuna sokarak başka bir dizilime yerleştirmektedir.

fill Fonksiyonları:

Bu fonksiyonlar bir veri yapısını belirli değerlerle doldurmak için yapılır.

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value);
```


generate Fonksiyonları:

Bu fonksiyonlar bir dizilimdeki değerleri bir fonksiyona sokarak güncellerler.

```
template<class ForwardIterator, class Generator>
```

```
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

```
template<class OutputIterator, class Size, class Generator>
```

```
void generate_n(OutputIterator first, Size n, Generator gen);
```

Buradaki fonksiyonun parametresi yoktur, geri dönüş değeri de dizilimde tutulan değer türünden olmalıdır.

Silme İşlemi Yapan Algoritmalar

Silme yapan fonksiyonlar *remove* ismiyle başlatılmışlardır. Bu fonksiyonlar aslında gerçek bir silme yapmazlar yalnızca silinecek elemanları dizilimin sonunda toplarlar ve genel olarak sonda topladıkları bu silinecek elemanlara başını gösteren iterator değeriyle geri dönerler. Böylece programcı nesne tutan sınıflarda remove işlemlerinden sonra gerçek bir erase işlemiyle ilgili elemanları silebilmektedir.

remove Fonksiyonları:

Bu fonksiyonlar, silinecek elemanları dizilimin sonuna alarak silinme noktasına ilişkin iterator değeriyle geri döner.

```
template<class ForwardIterator, class T>
```

```
ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T& value);
```

Bu fonksiyon dizinin içindeki value parametresine verilen değeri == ile arar, bulursa onu dizilimin sonuna alır. Standartlara göre silinecek elemanların dizilimin sonuna alınması biçiminin bir garantisi yoktur. Ancak pek çok algoritma silinecek elemandan sonraki elemanların silinecek eleman üzerine kaydırılması yöntemini izlemektedir. Ancak standartlarda silme işleminden önce ve sonraki durumda silinmeyen elemanların birbirlerine göre olan durumunun korunacağı garanti altına alınmıştır. Örneğin; int bir vector-den 2 olan elemanları silerek çıkartalım:

```
int main()
{
    int a[10] = {2, 3, 5, 7, 12, 2, 8, 2, 58,
22};
    int *p;
    p = remove(a, a + 10, 2);
    for (int i = 0; i < 10; i++)
        cout << a[i] << " ";
    return 0;
}
```

```
int main()
{
    int a[10] = {2, 3, 5, 7, 12, 2, 8, 2, 58,
22};
    vector<int> v(a, a + 10);
    vector<int>::iterator iter;
    iter = remove(v.begin(), v.end(), 2);
    v.erase(iter, v.end());
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    return 0;
}
```

`remove_if` fonksiyonu `remove` fonksiyonunun tek parametrelili karşılaştırma fonksiyonu alan biçimidir.

```
template<class ForwardIterator, class Predicate>
```

```
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, Predicate Pred);
```

Bir dizilimde belli bir koşulu sağlayan elemanların silinmesi için `remove_if` kullanılabilir.

SINIF ÇALIŞMASI:

Bir dizilimdeki elemanlardan küçük olanları silen programı `remove_if` kullanarak yazınız. Karşılaştırma fonksiyonu olarak template bir fonksiyon nesnesi alınız. Bu fonksiyon nesne sınıfının başlangıç fonksiyonu, dizilimin ilk ve sondan bir sonraki elemanının iteartor değerlerini parametre olarak alsın. Oratlamayı bularak ortalamadan küçük olanları silsin. Fonksiyon nesnesine ilişkin sınıfın tek yönde ilerleyen bir template iterator parametresi ile eleman türünü belirten bir template parametresi olması gerekir.

```
template <class FI, class T>
```

```
class Functor {
```

```
public:
```

```
    Functor(FI first, FI last)
```

```
    {
```

```
        T sum = 0;
```

```
        double count = 0;
```

```
        while (first != last){
```

```
            sum += *first;
```

```
            ++count;
```

```
            ++first;
```

```
        }
```

```
        m_aveg = sum / count;
```

```
    }
```

```
    bool operator()(const T &r)
```

```
    {
```

```
        return r < m_aveg;
```

```
    }
```

```
private:
```

```
    double m_aveg;
```

```
};
```

```
int main()
```

```
{
```

```
    int a[10] = {4, 6, 2, 43, 65, 12, 31, 13, 24, 5};
```

```
    vector<int> v(a, a + 10);
```

```
    vector<int>::iterator iter;
```

```
    iter = remove_if(v.begin(), v.end(), Functor< vector<int>::iterator, int>(v.begin(), v.end()));
```

```
    v.erase(iter, v.end());
```

```
    for (int i = 0; i < v.size(); i++)
```

```
        cout << v[i] <<" ";
```

```
    return 0;
}
```

remove fonksiyonunun da remove_copy ve remove_copy_if isminde türleri vardır.

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last, OutputIterator result,
const T& value);
```

unique Fonksiyonları:

Bu fonksiyonlar yan yana aynı olan elemanlar varsa onlardan 1 tane bırakacak biçimde diğerlerini mantıksal silerler. Bu fonksiyonlar da remove fonksiyonlarında olduğu gibi mantıksal bir silme yaparlar ve gerçekten silinmesi gereken noktanın iterator değeriyle geri dönerler.

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);
```

Bu fonksiyonlar özellikle sort işleminden sonra uygulanacak her elemandan tek kopya kalmasını sağlamak için kullanılır.

```
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last, BinaryPredicate pred);
```

```
iter = unique(v.begin(), v.end());
```

unique fonksiyonunun 2 de copy son ekli biçimi vardır.

```
template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last, OutputIterator result,
BinaryPredicate pred);
```

4-) Nesnelerin Yerlerini Değiştiren (Mutating) Algoritmalar:

Bu algoritmalar yalnızca elemanların yerlerini değiştirirler, değerlerini değil.

reverse Fonksiyonları:

reverse fonksiyonları bir dizilimi ters-düz etmek için kullanılır. İki biçimi vardır:

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
OutputIterator result);
```

rotate Fonksiyonları:

Bu fonksiyonlar döndürme işlemi yaparlar. Döndürme işlemi kaydırılanların dizinin başına yada sonuna alınması işlemidir.

Örneğin;

1 2 3 4 5 6 7 8 9

bu dizilimin sola doğru 1 kez döndürülmesiyle şu dizilim elde edilir.

2 3 4 5 6 7 8 9 1

rotate fonksiyonları üst üste n defa döndürme yapabilen fonksiyonlardır.

```
template<class ForwardIterator>
```

```
void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```

Fonksiyonun 1 ve 3. parametreleri, dizilimin başını ve sonunu belirlemede kullanılır. 2. parametresi aslında döndürme sayısı ve yönü üzerinde etkili olmaktadır. Döndürüldükten sonra ilk eleman 2. parametreyle belirtilen iteratordeki eleman olacaktır.

Bu fonksiyon ile sağa öteleme işlemi fazla sayıda sola öteleme yapılarak gerçekleştirilir.

rotate fonksiyonunun copy son ekli biçimi de vardır.

```
template<class ForwardIterator, class OutputIterator>
```

```
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result);
```

random_shuffle Fonksiyonları:

Bu fonksiyonlar bir dizideki elemanları karıştırmak için kullanılmaktadır. Örneğin bir kağıt oyununda kağıtların karıştırılması için bu fonksiyon kullanılabilir. Karıştırma algoritmaları n kere dizinin rastgele iki elemanının yer değiştirmesi esasına dayanır. Fonksiyonun iki biçimi vardır. 1. biçiminde rastgele sayıları üretmek için rand fonksiyonu kullanılır. 2. biçimde fonksiyon rastgele sayı üretmek için gereken rastsal sayı fonksiyonunu dışarıdan alır.

```
template <class RandomAccessIterator>
```

```
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

```
template <class RandomAccessIterator, class RandomNumberGenerator>
```

```
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator &rand);
```

SINIF ÇALIŞMASI:

Bir iskambil kağıdını Card isimli yapıyla aşağıdaki gibi ifade edelim.

```
enum {KUPA, KARO, MACA, SINEK};
```

```
enum {AS, IKI, UC, .., VALE, KIZ, PAPAZ};
```

```
struct Card {
```

```
    Card(int tur, int renk) : m_tur(tur), m_renk(renk)
```

```
{}
```

```
int m_tur, m_renk;
```

```
};
```

Card türünden bir vector içerisine sırasıyla 52 kartı yerleştiriniz. Bu kartları random_shuffle ile karıştırınız ve 4 oyuncuya 13'er tane dağıtınız. Her oyuncu için oyuncu1, oyuncu2, oyuncu3, oyuncu4 biçiminde 4 ayrı vector nesnesi alınacaktır. Kart dağıtılırken ana desteden çıkarılıp oyuncuların bulunduğu nesnelere yerleştirilecektir. Sonra oyunculara dağıtılan kartlar gösterilecektir.

```
enum {KUPA, KARO, MACA, SINEK};
```

```
enum {AS, IKI, UC, DORT, BES, ALTI, YEDI, SEKIZ, DOKUZ, ON, VALE, KIZ, PAPAZ};
```

```
struct Card {
```

```
    Card(int tur, int renk) : m_tur(tur), m_renk(renk)
```

```
{}
```

```
int m_tur, m_renk;
```

```

};

void Disp(const Card &c)
{
    cout << c.m_tur << " " << c.m_renk << endl;
}

int main()
{
    vector<Card> deste;
    vector<Card> oyuncu[4];
    Card kagit(0, 0);

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 13; j++) {
            kagit.m_tur = j;
            kagit.m_renk = i;
            deste.push_back(kagit);
        }

    random_shuffle(deste.begin(), deste.end());

    for (int i = 0; i < 52; i++)
        Disp(deste[i]);

    int p = 0;
    for (int i = 0; i < 13; i++)
        for (int j = 0; j < 4; j++)
            oyuncu[j].push_back(deste[p++]);

    for (int i = 0; i < 4; i++){
        cout << endl << "oyuncu " << i << endl;
        for (int k = 0; k < 13; k++)
            Disp(oyuncu[i][k]);
    }
    return 0;
}

```

partition Fonsiyonları:

Bu fonksiyonlar, dizilimin belirli bir kritere uygun olan değeri ile uygun olmayan değerlerini birbirinden ayırırlar. Kritere uygun olan değerler dizinin solunda, uygun olmayanlar sağında bulunur.

```

template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition (BidirectionalIterator first, BidirectionalIterator last,
Predicate pred);

```

Fonksiyonun 3. parametresi kriteri belirlemektedir. Karşılaştırma fonksiyonunun geri dönüş değeri true ise ölçüte uygunluğu, false ise uygun olmama durumunu belirtir. Örneğin; bir dizinin tek ve çift elemanlarını ayırılım:

```

bool Func(int a)

```

```

{
    return a %2 == 0;
}

int main()
{
    int a[10] = {1, 2, 4, 6, 76, 35, 13, 7, 9, 11};
    partition(a, a + 10, Func);
    for (int i = 0; i < 10; i++)
        cout << a[i] << endl;
}

```

Fonksiyon, ayrılma noktasının iterator değeriyle geri döner.

```
template<class BidirectionalIterator, class Predicate>
```

```
BidirectionalIterator stable_partition (BidirectionalIterator first, BidirectionalIterator last,
Predicate pred);
```

Bu fonksiyon partition fonksiyonunun eleman sıralarının muhafaza edilmiş halidir. partition fonksiyonu koşulu sağlayanları sola atar fakat, sola attıklarının görelî sıraları gerçek dizilimden farklı olabilir. Ancak stable_partition fonksiyonu bunu garanti eder.

Anahtar Notlar:

stable ön ekli fonksiyonlar görelî yerleri muhafaza eden fonksiyonlardır. Şüphesiz fonksiyonların stable biçimlerinin karmaşıklığı daha yüksektir.

nth_element Fonksiyonları:

Bu fonksiyonlar, bir dizilimdeki elemanları öyle yer değiştirir ki belirli bir iterator pozisyonunun solunda olan her eleman, sağında olan her elemandan küçük yada ona eşit olur. Fonksiyonlardaki iterator, büyüklerin ve küçüklerin düğüm noktasını belirtir.

```
template<class RandomAccessIterator>
```

```
void nth_element (RandomAccessIterator firts, RandomAccessIterator nth,
RandomAccessIterator last);
```

Fonksiyonunun 2. biçimde küçüklük, büyüklük ilişkisi karşılaştırma fonksiyonu ile belirlenir. Karşılaştırma fonksiyonundan true ile geri dönülmesi küçüklük, false ile dönülmesi büyüklük anlamına gelir.

```
template<class RandomAccessIterator, class Compare>
```

```
void nth_element(RandomAccessIterator firts, RandomAccessIterator nth,
RandomAccessIterator last, Compare comp);
```

5-) Sort İşlemi Yapan (Sorting) Algoritmalar:

Standart kütüphanede bir grup sort fonksiyonu vardır. Bu fonksiyonların karmaşıklıkları $O(n) = n \log n$ biçimindedir. Tipik olarak quick sort ile gerçekleşir.

En basit sort fonksiyonu şöyledir:

```
template<class RandomAccessIterator>
```

```
void sort (RandomAccessIterator first, RandomAccessIterator last);
```

Bu sort fonksiyonunda < operatörü kullanılmıştır.

SINIF ÇALIŞMASI:

Bir yapı alınız. Bu yapı için < operator fonksiyonunu yazınız ve bu yapı türünden bir diziyi sort fonksiyonu ile sort ediniz.

```
struct Yapi {
    Yapi()
    {}
    Yapi(int x) : m_x(x)
{}
bool operator < (const Yapi& a) const
{
    return m_x < a.m_x;
}
int m_x;
};
int main()
{
    Yapi yapi[5];
    for (int i = 0; i < 5; i++)
        yapi[i].m_x = rand() % 10 + 1;
    sort(yapi, yapi + 5);
    for (int i = 0; i < 5; i++)
        cout << yapi[i].m_x << " ";
    return 0;
}
```

sort fonksiyonunun 2. biçiminde < operatorunun işlemini bir fonksiyonun yapması istenmektedir. Yani fonksiyonun 2. parametresi olmalı, geri dönüş değeri bool olmalıdır. Eğer büyükten küçüğe dizme isteniyorsa yine bu fonksiyonlar ayarlama yapılabilir. Yani örneğin biz bu fonksiyon içerisinde < operatorunun işlevini yerine getirirsek dizi küçükten büyüğe, > büyüktür operatorunun işlevini yerine getirirsek dizi büyükten küçüğe sıralanacaktır.

```
bool Comp(int a, int b)
{
    return a > b;
}
```

```
sort(a, a + 10, Comp);
```

```
template<class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Daha açık bir anlatımla, sort işlemi, bir grup yer değiştirme işlemidir. Fonksiyon içerisinde bu karşılaştırma fonksiyonu çağrılmış karşılaştırma fonksiyonu true ise yer değiştirme yapılmamış, karşılaştırma fonksiyonu false ise yer değiştirme yapılmamış, false ise yapılmıştır.

stable_sort Fonksiyonu:

Bu fonksiyonlar sıraya dizmekle birlikte aynı olan elemanları gerçek dizideki göreceli yerlerini muhafaza eder.

```
template<class RandomAccessIterator >
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

partial_sort Fonksiyonları:

Bu fonksiyonlar bir dizilimin en büyük yada en küçük ilk n elemanlarını sırasıyla elde etmek için kullanılır.

```
template<class RandomAccessIterator,>
void partial_sort(RandomAccessIterator first, RandomAccessIterator last);
```

Fonksiyonun 2. parametresi aslında en küçük yada en büyük kaç elemanın elde edileceğini belirlemekte kullanılır. 1. ve 2. parametre ([first, middle)) arasında kalan elemanlar bu sayıyı belirtmektedir. Fonksiyonun karşılaştırma fonksiyonlu biçimi de vardır.

```
template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Anahtar Notlar:

sort fonksiyonlarında büyükten küçüğe sort etmek için yeniden bir karşılaştırma fonksiyonu yazmak yerine kütüphanedeki default template fonksiyon nesne sınıfları kullanılabilir. Örneğin; greater fonksiyonu iki parametre alan ve bu iki parametreyi > ile karşılaştıran bir fonksiyon nesne sınıfıdır. Anımsanacağı gibi bu template sınıfların parametre türlerini anlatan tek bir template parametresi vardır.

```
partial_sort(a, a + 3, a + 10, greater<int>());
```

Ayrıca fonksiyonun copy son ekli aşağıdaki biçimleri de vardır.

```
template<class InputIterator, class RandomAccessIterator >
RandomAccessIterator partial_sort_copy(InputIterator first, InputIterator last,
RandomAccessIterator result_first, RandomAccessIterator result_last);
```

```
template<class InputIterator, class RandomAccessIterator, class Compare >
RandomAccessIterator partial_sort_copy(InputIterator first, InputIterator last,
RandomAccessIterator result_first, RandomAccessIterator result_last, Compare comp);
```

Sıralı Diziler Üzerinde İşlem Yapan Özel Fonksiyonlar

Standart kütüphanedeki bazı algoritmalar önkoşul olarak dizilimi sıralı olmasını istemektedir.

merge Fonksiyonları:

merge işlemleri özellikle tek hamlede sıraya dizilemeyecek büyük dizilerin parça parça sıraya dizildikten sonra birleştirilmesi anlamına gelir.

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, Output Iterator result)
```

Fonksiyon, sıralı iki dizilimi alarak 3.dizilimde birleştirir. Fonksiyonun geri dönüş değeri hedef dizide birleştirmeden sonra kalınan noktayı belirtir.


```
template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
InputIterator2first2, InputIterator2 last2, Output Iterator result, Compare Comp);
```

1. fonksiyon < operatorunu kullanır.
2. 2. fonksiyonda < operatorunun işlemi fonksiyona yüklenmiştir.

Permütasyon Oluşturan Fonksiyonlar:

Bir dizideki elemanların her türlü permütasyonunu elde eden next_permutation fonksiyonları vardır.

```
template<class BidirectionalIterator>
bool next_permutation (BidirectionalIterator first, BidirectionalIterator last)
```

Fonksiyon her çağrıldığında verilen dizinin yeni bir permütasyonunu oluşturur. Fonksiyonun bir döngü içinde çağrıldığını düşünelim. Yeniden ilk noktaya gelindiğinde fonksiyon false ile, diğer durumlarda true ile geri döner. Peki fonksiyon ilk duruma gelindiğini nasıl anlar? İşte programcı işe başlarken dizilimi sıraya dizilmiş bir biçimde verir. Fonksiyon her permütasyonda eleman sıralamasına bakar ve bunun küçükten büyüğe sort edilip edilemediğine bakar. Küçükten büyüğe sort olmamış durumda ise false ile geri döner.

Örnek:

```
int main()
{
    int a[3] = {1, 2, 3};
    do {
        for (int i = 0; i < 3; i++)
            cout << a[i] << " ";
        cout << "\n";
    } while(next_permutation(a, a + 3));

    return 0;
}
```

pair Sınıfı

pair sınıfı bir çift bilgiyi tutmakta kullanılan yararlı bir sınıftır. Standart kütüphane içerisinde bazı sınıflar içerisinde *pair* sınıfı dolaylı bir biçimde kullanılmaktadır. *pair* sınıfının bildirimi şöyledir:

```
template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;
    pair();
    pair(const T1& x, const T2& y);
    template<class U, class V> pair(const pair<U, V> &p);
};
```

Ayrıca standart kütüphanede aynı türden iki pair nesnesini karşılaştıran operator fonksiyonları da vardır. Karşılaştırma işleminde first, second elemanından daha yüksek anlamlı değer belirtir.

Ayrıca *make_pair* isimli fonksiyon bir pair nesnesi elde etmek için kullanılabilir.

```
template <class T1, class T2>
pair<T1, T2> make_pair(T1 x, T2 y);
```

Iteratorlerin Ayrıntılı İncelenmesi

Bilindiği gibi iterator kavramı T türünden bir gösterici gibi kullanılabilen genel bir kavramdır. İterator düz bir gösterici olabileceği gibi bir sınıfta olabilir. Hangi iterator kategorilerinin hangi operatörleri desteklediği standartlarda belirlenmiştir. Normal göstericiler iterator olarak kullanılırsa rastgele erişimli (Random Access Iterator) olarak ele alınırlar. Standartlarda "24.1 Iterator Requirements" başlığı altında iterator kategorilerinin desteklemesi gerektiği operatorler belirtilmiştir. i, T türünden bir iterator olsun(yani *i T türünden olsun). Eğer T bileşik bir türse i iteratorü için ok(->) operatörü de tanımlı olmak zorundadır.

Örneğin;

```
#include <iostream>
#include <algorithm>
#include <string>
#include <vector>
#include <list>
#include <iomanip>
#include <fstream>
#include <sstream>
#include <windows.h>
```

```
using namespace std;
```

```
struct Person {
    Person(const char *name, int no) : m_name(name), m_no(no)
    {}
    string m_name;
    int m_no;
};

int main()
{
    vector<Person> v;

    v.push_back(Person("Kaan Aslan", 123));
    v.push_back(Person("Ali Serçe", 321));
    v.push_back(Person("Necati Ergin", 875));

    vector<Person>::iterator iter = v.begin();

    cout << (*iter).m_name << "\n";
    cout << iter->m_name << "\n";

    return 0;
}
```

İterator Sınıflarının Yazılması

Bir sınıfa iterator desteği vermek bazen gerekli olabilir. Yani örneğin yeni bir nesne tutan sınıf tasarlasak, bu sınıfın algoritmalarda kullanılabilmesi için sınıfa iterator desteğinin verilmesi gerekir. Sınıfa iterator desteği vermek için sınıfın iterator kategorisi ne olacaksa ona uygun olarak *begin*, *end*, *rbegin*, *rend* gibi iterator veren fonksiyonları yazmak gerekir. Daha sonra iterator türünün normal bir gösterici mi yoksa bir sınıf mı olması gerektiğine karar verilir. Eğer iterator türü bir sınıf ise sınıf gerçek sınıfın içerisinde bildirilerek gizlenebilir.

İterator sınıfının hangi operatorleri destekleyeceği iterator kategorisine bağlıdır. Fakat standartlara göre her iterator sınıfının *value_type*, *difference_type*, *pointer*, *reference*, *iterator_category* isimli türlere sahip olması gerekir. Yani bizim iterator sınıfı için bu türleri typedef etmemiz gerekir. Bu isimlerin typedef edilmesini kolaylaştırmak için <iterator> başlık dosyası içerisinde *iterator* isimli bir sınıf düşünülmüştür. Böylece programcı iterator sınıfı yazarken bu tür isimlerini typedef etmek yerine sınıfını iterator sınıfından türetebilir. İterator sınıfı şöyledir:

```
namespace std {
template<class Category, class T, class Distance = ptrdiff_t,
class Pointer = T*, class Reference = T&>
struct iterator {
typedef T value_type;
typedef Distance difference_type;
typedef Pointer pointer;
typedef Reference reference;
typedef Category iterator_category;
};
}
```

İterator kategorisi *input_iterator_tag*, *output_iterator_tag*, *forward_iterator_tag*, *bidirectional_iterator_tag*, *random_access_iterator_tag* içi boş sembolik sınıflardan biri olarak girilmelidir.

İterator sınıfı nasıl ilerleme yapacaktır? İlerleme işlemi için iterator sınıfının bazı bilgilere gereksinimi olabilir. En genel olarak gerçek sınıfın nesne adresi iterator sınıfına geçirilebilir ve bu sınıfa arkadaşlık verilebilir. Örneğin; A gerçek sınıf, AIter ise bu sınıfın iterator sınıfı olsun, düzenleme şöyle yapılabilir:

```
class A {
private:
    friend class AIter;
    class AIter : public iterator<input_iterator_tag, T> {
    public:
        AIter(A *pA) : m_pA(pA)
        {}
        // ...
    private:
        A *m_pA;
    };
public:
    typedef AIter iterator;
    iterator begin()
    {
        return iterator(this);
    }
}
```

```
// ...
};
```

Dizindeki Dosyaları Dolaşan Örnek Bir İterator Destekli Sınıf

Bilindiği gibi DOS ve Windows sistemleri bir dizin içerisindeki dosyaları tek yönlü dolaşmaya izin veren sistem fonksiyonları bulundurmaktadır. Microsoft derleyicilerinde bu fonksiyonlar `_findfirst` ve `_findnext` fonksiyonlarıdır.

Anahtar Notlar:

Microsoft C++ 6.0 derleyicisinin özellikle template konusunda standart uyumu zayıftır. Eksik noktaların çoğu 7.0 (.NET içerisindeki versiyon) versiyonunda düzeltilmiştir. 6.0' daki kimi bozukluklarında service packlerle düzeltildiği bilinmektedir.

Örnek uygulamamızda dolaşımı sağlayan gerçek sınıf *DirWalk* sınıfı, iterator sınıfı ise *DirWalkIter* sınıfıdır.

```
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <cstring>
#include <io.h>
```

```
using namespace std;
```

```
class DirWalk {
```

```
private:
```

```
    friend class DirWalkIter;
```

```
    class DirWalkIter : public iterator<input_iterator_tag, _finddata_t> {
    public:
```

```
        DirWalkIter()
```

```
        {
```

```
            m_fd = -1;
```

```
        }
```

```
        DirWalkIter(long fd, const _finddata_t &current) : m_fd(fd),
m_current(current)
```

```
        {}
```

```
        ~DirWalkIter()
```

```
        {
```

```
            if (m_fd != -1)
```

```
                ::_findclose(m_fd);
```

```
        }
```

```
        bool operator ==(const DirWalkIter &iter) const
```

```
        {
```

```
            return memcmp(&m_current, &iter.m_current, sizeof(m_current)) ==
```

```
0;
```

```

    }
    bool operator !=(const DirWalkIter &iter) const
    {
        return memcmp(&m_current, &iter.m_current, sizeof(m_current)) != 0;
    }
    _finddata_t operator *()
    {
        return m_current;
    }
    DirWalkIter &operator ++()
    {
        if (_findnext(m_fd, &m_current) == -1)
            m_current = ms_null;
        return *this;
    }
    DirWalkIter operator ++(int)
    {
        DirWalkIter temp(*this);
        if (::_findnext(m_fd, &m_current) == -1)
            m_current = ms_null;
        return temp;
    }
    _finddata_t *operator ->()
    {
        return &m_current;
    }
private:
    long m_fd;
    _finddata_t m_current;
    static _finddata_t ms_null;
};
public:
    DirWalk(const char *pszPath) : m_path(pszPath)
    {
    }
    typedef DirWalkIter iterator;
    iterator begin()
    {
        long fdh;
        _finddata_t fd;
        fdh = ::_findfirst(m_path.c_str(), &fd);
        return DirWalkIter(fdh, fd);
    }
    iterator end()
    {
        _finddata_t fd = {0};
        return iterator(-1, fd);
    }
private:

```

```

        string m_path;
};
_finddata_t DirWalk::DirWalkIter::ms_null = {0};
void Disp(const _finddata_t &fd)
{
    cout << fd.name << "\n";
}
int main(void)
{
    DirWalk dir("c:\\windows\\*.");
    for_each(dir.begin(), dir.end(), Disp);
    return 0;
}
/*
int main(void)
{
    DirWalk dir("c:\\windows\\*.");
    vector<_finddata_t> v;
    copy(dir.begin(), dir.end(), back_inserter(v));
    for (int i = 0; i < v.size(); ++i)
        cout << v[i].name << "\n";
    return 0;
}
*/

```

Uygulamada Microsoft'un `_findfirst` ve `_findnext` fonksiyonları kullanılabilir. `iterator ++` operator fonksiyonuyla artırıldığında `_findnext` fonksiyonu çağırılarak dizin içerisindeki sonraki elemana geçilebilir. `DirWalk` sınıfının `iterator` veren `begin` fonksiyonu dizinin ilk elemanına ilişkin `iterator` değerini verir. `end` üye fonksiyonunun son elemandan bir sonraki elemanın `iterator`unu vermesi gerekir, fakat dizin için böyle bir eleman söz konusu değildir. O halde son elemandan bir sonraki elemanı temsil eden kukla bir eleman kullanılabilir.

Ters Yönde İlerleyen İteratorler (Reverse Iteratorler)

Ters yönde ilerleyen `iterator`ler aslında normal `iterator`ler kullanılarak oluşturulabilir. Gerçekte standartlarda `reverse_iterator` isimli bir `template` sınıf tanımlanmıştır. Yani biz normal bir `iterator` sınıfı yazmış isek `reverse_iterator` sınıfı yazmamıza gerek yoktur. Kütüphane içerisindeki `reverse_iterator` `template` sınıfı bu amaçla kullanılabilir. kısacası aslında `reverse iterator` işlemleri normal `iterator`leri kullanan bir sarma sınıf yoluyla yapılabilir. Şüphesiz bir `iterator`ün ters yönde ilerleyebilmesi için onun çift yönlü ya da rastgele erişimli olması gerekir.

```

class MyIterator{
    //...
};
reverse_iterator<MyIterator>
riter(MyIterator());

```

Programcı daha ayrıntılı bir takım işlemler yapmak istiyorsa `reverse_iterator` sınıfından türetmeler yapabilir. Bu durumda

```

using namespace std;
struct Person {

```

```

    Person(const char *name, int no) : m_name(name), m_no(no)
    {}
    string m_name;
    int m_no;
};

int main()
{
    vector<int> a;
    for (int i = 0; i < 10; ++i)
        a.push_back(i);
    //vector<int>::reverse_iterator riter;
    reverse_iterator<vector<int>::iterator> riter;
    return 0;
}

```

→ İki aynı anlamda

Özetle ters yönde ilerleyen iteratorler için bir sınıf ayrıca yazmaya gerek yoktur. Bir iteratörden ters yönde ilerleyen sınıfı elde eden *reverse_iterator* sınıfı standart olarak vardır.

Bir sınıfa reverse iterator desteği vermek isteyelim. Bu durumda yapılacak şey sınıfa normal iterator işlemlerini yapan bir iterator sınıfı yazmak ve standart *reverse_iterator* sınıfını kullanarak gerçek sınıfın *reverse_iterator* türünü *typedef* etmektir. Örneğin,

```

class A{
private:
    class Alter{
        //...
    };
public:
    typedef Alter iterator;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    //...
};

```

back_insert_iterator Sınıfı ve back_inserter Fonksiyonu

Normal olarak algoritmaların çoğu kopyalama esasıyla çalışır. Örneğin, *copy* fonksiyonu bir dizilimdeki bazı değerleri başka bir dizilime kopyalar. *transform* fonksiyonu bir dizilimdeki değerleri bir işleme soktukten sonra başka bir dizilime kopyalar. Burada hedef dizilimlerin önceden tahsis edilmiş olması gerekir.

back_insert_iterator sınıfı basit bir sarma sınıfıdır. Başlangıç fonksiyonu parametre olarak bir nesne tutan sınıf türünden nesne alır. Bu nesnenin adresini sınıfın *protected* bölümündeki bir nesneye yerleştirir. Sınıfın *++* ve *** operator fonksiyonları hiçbir şey yapmaz yalnızca nesnenin kendisiyle geri döner. Fakat sınıfın atama operator fonksiyonu *protected* bölümünde sakladığı nesne adresiyle *push_back* işlemi yapmaktadır. Böylece algoritmalarındaki *** ve *++* operatorleri bir etkiye sahip olmazken atama operatorü sona ekleme anlamına gelecektir.

```

namespace std {
template <class Container>
class back_insert_iterator :
public iterator<output_iterator_tag,void,void,void,void> {
protected:
    Container* container;
public:

```

```

typedef Container container_type;
explicit back_insert_iterator(Container& x);
back_insert_iterator<Container>&
operator=(typename Container::const_reference value);
back_insert_iterator<Container>& operator*();
back_insert_iterator<Container>& operator++();
back_insert_iterator<Container> operator++(int);
};
template <class Container>
back_insert_iterator<Container> back_inserter(Container& x);
}

```

back_insert_iterator sınıfı bir template sınıftır. Sınıfın template parametresi nesne tutan sınıfın türünü belirtmektedir. Örneğin;

```

using namespace std;
struct Person {
    Person(const char *name, int no) : m_name(name), m_no(no)
    {}
    string m_name;
    int m_no;
};
int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> v;
    copy(a, a + 10, back_insert_iterator<vector<int>>>(v));
    for (int i = 0; i < v.size(); ++i)
        cout << v[i] << endl;
    return 0;
}

```

Burada v için başlangıçta yer ayrılmamıştır. Kopya algoritması adeta mod değiştirmekte ve insert işlemi yapmaktadır. Aşağıdaki örnekte iki dizinin karşılıklı elemanları toplanarak bir vektöre insert edilmiştir.

```

int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> v;
    transform(a, a + 10, b, back_insert_iterator<vector<int>>>(v),
        plus<int>());
    for (int i = 0; i < v.size(); ++i)
        cout << v[i] << endl;
    return 0;
}

```

back_insert_iterator işlemini kolaylaştırmak için *back_inserter* fonksiyonu düşünülmüştür. *back_inserter* fonksiyonu şöyle yazılmıştır.

```

template <class Container>
back_insert_iterator<Container> back_inserter(Container& x)
{
    return back_insert_iterator<Container>(x);
}

```



```
}
```

Bu durumda:

```
back_inserter(v);
```

işlemi ile

```
back_insert_iterator<vector<T> > (v);
```

aynı anlamdadır. O halde bir önceki işlem şöyle basitleştirilebilir.

```
int main()
```

```
{
```

```
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
    int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
    vector<int> v;
```

```
    transform(a, a + 10, b, back_inserter(v), plus<int>());
```

```
// copy(a, a + 10, back_inserter(v));
```

```
    for (int i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << endl;
```

```
    return 0;
```

```
}
```

front_insert_iterator Sınıfı front_inserter Fonksiyonu

back_insert_iterator sınıfının ve back_inserter fonksiyonunun veri yapısının önüne eklemek için kullanılan biçimleridir. Kullanım biçimi tamamen benzerdir. Örneğin biz copy işlemini bir yapısının önüne ekleme yapmak için kullanabiliriz.

```
int a[5] = {100, 200, 300, 400, 500};
```

```
list<int> lst;
```

```
for (int i = 0; i < 10; ++i)
```

```
    lst.push_back(i);
```

```
copy(a, a + 5, front_inserter(lst));
```

Burada dizi içerisindeki değerler bağlı listenin önüne insert edilmiştir.

```
int main()
```

```
{
```

```
    int a[5] = {100, 200, 300, 400, 500};
```

```
    list<int> lst;
```

```
for (int i = 0; i < 10; ++i)
```

```
    lst.push_back(i);
```

```
copy(a, a + 5, front_inserter(lst));
```

```
for (list<int>::iterator iter = lst.begin(); iter != lst.end(); ++iter)
```

```
    cout << *iter << "\n";
```

```
return 0;
```

```
}
```

insert_iterator Sınıfı ve inserter Fonksiyonu

insert_iterator sınıfı ve *inserter* fonksiyonu da benzer mantıkla çalışmaktadır. Fakat bu sınıf ve fonksiyon başa ya da sona insert etmek yerine belli bir iterator pozisyonuna insert işlemi yapar. *insert_iterator* sınıfının başlangıç fonksiyonun bir nesne tutan sınıf dışında bir de iterator nesnesi vardır, insert işlemi bu iteratorün gösterdiği yere yapılır. *inserter* fonksiyonu da iki parametrelidir. Örneğin;

```
int main()
{
    int a[5] = {100, 200, 300, 400, 500};
    vector<int> v;
    for (int i = 0; i < 10; ++i)
        v.push_back(i);
    copy(a, a + 5, inserter(v, v.begin() + 2));
    for (int i = 0; i < v.size(); ++i)
        cout << v[i] << endl;
    return 0;
}
```

advance Fonksiyonu

iter bir iterator olmak üzere bu iteratorun gösterdiği elemandan n sonraki elemanın iterator değeri nasıl elde edilir? Eğer iterator rasgele erişimli iterator ise iterator + ve - operatorlerini desteklediği için doğrudan "*iter + n*" işlemi yapılabilir. Örneğin v bir vektör nesnesi olmak üzere bu nesnenin üçüncü indisli elemanına ilişkin iterator değeri "*v.begin()+3*" ile elde edilebilir. Fakat iterator rasgele erişimli değilse bir iteratorün gösterdiği elemandan n sonraki elemanın iterator değerini bulmak için bir döngü içerisinde ilerlemek zorunludur.

```
while(n-- > 0)
    ++iter;
```

İşte bu işlem *advance* fonksiyonuyla yapılabilir.

```
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);
```

Örnek;

```
int main()
{
    int a[5] = {100, 200, 300, 400, 500};
    list<int> l;

    copy(a, a + 5, back_inserter(l));

    list<int>::iterator iter = l.begin();
    advance(iter, 3);

    l.insert(iter, 100000);

    for (list<int>::iterator iter = l.begin(); iter != l.end(); ++iter)
        cout << *iter << endl;

    return 0;
}
```

distance Fonksiyonu

distance fonksiyonu da eğer iterator rasgele erişimli değilse iki iterator arasındaki eleman sayısını vermektedir. Örneğin, *distance(x.begin(),x.end())* fonksiyonunun geri dönüş değeri dizilimdeki eleman sayısıdır.

```
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

istream ve ostream Iteratorleri

istream_iterator isimli iterator sınıfı bir iterator özelliği göstermesinin yanısıra *istream* okuması da yapar. Benzer biçimde *ostream_iterator* sınıfı da yazma işlemi yapan bir iterator sınıfıdır. *istream_iterator* sınıfının bildirimi şöyledir:

```
template <class T, class charT = char, class traits = char_traits<charT>,
class Distance = ptrdiff_t>
class istream_iterator:
public iterator<input_iterator_tag, T, Distance, const T*, const T&> {
...
};
```

Görüldüğü gibi *istream_iterator* sınıfının dört template parametresi vardır ama yalnızca ilk parametresi belirtilmek zorundadır. Bu parametre okumada hangi türden bilginin okunacağını belirtir. Örneğin:

```
istream_iterator<int> a(cin);
```

Burada *cin* nesnesi kullanılarak her defasında *int* bir bilgi okunmak istenmektedir.

istream_iterator sınıfının iki başlangıç fonksiyonu vardır: default başlangıç fonksiyonu ve *istream&* parametrelili başlangıç fonksiyonu. Sınıfın default başlangıç fonksiyonu *end* iteratorü etkisi yaratmaktadır. Yani dosyanın sonuna gelindiğini belirlemekte kullanılır. Parametrelili başlangıç fonksiyonu okumada hangi *istream* nesnesinin kullanılacağını belirtir. Örneğin:

```
istream_iterator<int> a(cin);
fstream f(...);
istream_iterator< long> b(f);
```

Birinci örnekte *cin* nesnesi kullanılarak yani klavyeden *int* türden bilgiler okunmak istenmektedir. İkinci örnekte *f* nesnesi kullanılarak, yani dosyadan okuma yapılacaktır. Okunacak bilgi *long* türündendir.

Örneğin, bir dosyada aralarına boşluk bırakılmış biçimde tam sayılar bulunsun. *istream_iterator* türünden iterator bu tam sayıları okumakta kullanılan bir iterator olabilir. Sınıfın *** operator fonksiyonu en son okunan değeri verir. Sınıfın önek ve sonek *++* operator fonksiyonları sonraki elemanı gerçek anlamda okur. Yani iteratorun ilerletilmesi sıradaki değerin okunması *** işlemi ise okunan değerin verilmesi anlamındadır. Sınıfın parametrelili başlangıç fonksiyonu ilk elemana konumlandırma yaptığı için okumaya da yol açar. Sonek *++* operatorü yeni bir değer okumakla birlikte eski okunan değerin iteratorüne geri döner. Yani biz **iter++;* gibi bir işlemle yeni bir değer okunmakla birlikte önceki okunan değer elde edilmektedir.

istream_iterator türünden bir iterator ile okuma yapılırken, dosyanın sonuna gelinmişse ya da okuma hatası oluşmuşsa sınıfın başlangıç fonksiyonunun ürettiği iterator değeri elde edilir.

Örneğin;

```
#include <iostream>
#include <string>
#include <cstring>
#include <cctype>
#include <vector>
#include <algorithm>
#include <utility>
#include <list>
#include <iterator>
#include <windows.h>
```

```
using namespace std;
```

```
int main(void)
{
    istream_iterator<int> iter(cin);
    while (iter != istream_iterator<int>()) {
        cout << *iter << " ";
        ++iter;
    }
    return 0;
}
```

Burada döngü dosya sonuna gelene kadar okuma işlemi bir biçimde başarısız olana kadar devam etmektedir. İlk okuma standartlara göre parametrelili başlangıç fonksiyonu tarafından ya da * operatorünün ilk kullanıldığı noktada yapılabilir. Diğer okumalar ++iter işlemiyle yapılmaktadır.

Aşağıdaki örnekte EOF tuşuna basılana kadar klavyeden girilenler vektöre yerleştirilmektedir.

```
vector<int> v;
copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));
```

Burada EOF tuşuna basılana kadar kaynak iteratörden yani klavyeden v vektörüne kopyalama yapılmaktadır. *back_inserter* sayesinde kopyalama işlemi sona ekleme işlemine dönüştürülmüştür.

```
using namespace std;
```

```
int main(void)
{
    vector<int> v;
    copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << "\n";
}
```

Sınıf Çalışması:

Klavyeden girilen int türden sayılar içerisinde 100' e eşit olanların sayısını yalnızca *count* algoritmasını kullanarak bulunuz.

```
using namespace std;
```

```
int main(void)
{
    size_t c;
    cout << count(istream_iterator<int>(cin), istream_iterator<int>(), 100) << endl;
    return 0;
}
```

Dosyadan okumak için;

```
using namespace std;
```

```
int main(void)
{
    ifstream f("x.txt");
    if (!f) {
        cout << "Cannot open file!..\n";
        exit(EXIT_FAILURE);
    }
    cout << count(istream_iterator<int>(f), istream_iterator<int>(), 100) << endl;
    return 0;
}
```

Sınıf Çalışması:

Yazılan bir cümlede içerisinde a geçen sözcüklerin sayısını bulan programı *count_if* fonksiyonunu kullanarak en basit biçimde yazınız.

```
using namespace std;
```

```
bool contains(const string &s)
```

```
{
    return s.find('a') != string::npos;
}
```

```
int main(void)
```

```
{
    cout << count_if(istream_iterator<string>(cin), istream_iterator<string>(), contains) << endl;
    return 0;
}
```

ostream_iterator sınıfının bildirimi şöyledir:

```
template <class T, class charT = char, class traits = char_traits<charT> >
```

```
class ostream_iterator:
```

```
public iterator<output_iterator_tag, void, void, void, void> {
```

```
...
};
```

Görüldüğü gibi sınıfın üç template parametresi vardır. Yalnızca ilk parametresinin yazılması zorunludur. Bu parametre yazdırılacak bilginin türünü temsil etmektedir. Sınıfın başlangıç fonksiyonu iki tanedir:

1. *ostream_iterator(ostream & s);*

2. *ostream_iterator*(*ostream_type*& s, *const char** delimiter);

Parametredeki *ostream* nesnesi yazdırmada kullanılacak nesneyi belirtir. İkinci parametredeki yazı yazma sonrasında ayıraç olarak kullanılacak karakterleri belirtir.

ostream_iterator sınıfının önek ve sonek ++ iterator fonksiyonları ve * operator fonksiyonu hiçbir şey yapmaz. *this değeriyle yani iteratorun kendi değeriyle geri döner. Fakat atama operator fonksiyonu atanan değeri başlangıç fonksiyonu ile belirtilen nesneyi kullanarak yazdırır sonra ayıraç karakterlerini yazdırır. Örneğin;

```
ostream_iterator<int> iter(cout, "\\n");
```

```
iter = 100;
```

```
*iter = 200;
```

```
++iter;
```

```
*iter++ = 300;
```

"iter = 100;" işleminde 100 değeri *cout* ile ekrana yazılır. Sonra cursor aşağı satırın başına geçerilir. "*iter = 200;" işleminde *iter ile iter arasında bir fark yoktur. ++iter etkisizdir. "*iter++ = 300;" işleminde *iter++' dan sonuçta iter elde edileceği için bu işleminde "iter= 300;" den bir farkı yoktur.

Bir dizilimi tek hamlede ekrana yazdırmak için aşağıdaki kalıp çok sık kullanılmaktadır:

```
vector<int> v;
```

```
...
```

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\\n"));
```

Dizide Kullanımı:

```
int a[] = {1, 2, 3, 4, 5};
```

```
...
```

```
copy(a, a + 5, ostream_iterator<int>(cout, "\\n"));
```

Sınıf Çalışması:

İki dizilimin karşılıklı elemanlarının toplamını ekrana yazdıran ifadeyi yazınız.

```
int a[] = {1, 2, 3, 4, 5};
```

```
int b[] = {1, 2, 3, 4, 5};
```

```
transform(a, a + 5, b, ostream_iterator<int>(cout, "\\n"), plus<int>());
```

b dizisini a dizisinin negatifiyle dolduran program:

```
int a[] = {1, 2, 3, 4, 5};
```

```
int b[5];
```

```
transform(a, a + 5, b, negate<int>());
```

```
copy(b, b + 5, ostream_iterator<int>(cout, "\\n"));
```

Yazılan sözcüğü ters yazan program:

```
using namespace std;
```

```
string rev(const string &s)
```

```
{
```

```
    string dest = s;
```

```
    reverse(dest.begin(), dest.end());
```

```
    return dest;
```

```
}
```

```
int main(void)
```

```

{
    transform(istream_iterator<string>(cin), istream_iterator<string>(),
              ostream_iterator<string>(cout, " "), rev);
    return 0;
}

```

Sınıf Çalışması:

Klavyeden girilen bir cümlemin sözcüklerini ters sırada yazdıran programı yazınız.
using namespace std;

```

string rev(const string &s)
{
    string dest = s;

    reverse(dest.begin(), dest.end());
    return dest;
}

int main(void)
{
    vector<string> v;
    copy(istream_iterator<string>(cin), istream_iterator<string>(),
          back_inserter(v));
    copy(v.rbegin(), v.rend(), ostream_iterator<string>(cout, " "));
    return 0;
}

```

deque Sınıfı

deque vektöre benzeyen temel bir veri yapısıdır. İsmi ingilizce "double ended queue" sözcüklerinden gelmektedir.

Tasarımsal olarak bir vektör tipik olarak tek parçalı dinamik tahsis edilmiş bir dizidir. Halbuki *deque* bloklardan oluşturulmuş bir bağlı liste biçimindedir. *deque* sınıfının iteratorleri vektörde olduğu gibi rastgele erişimli (random access) iteratorlerdir. Yani [] operatörü ile tıpkı vektörde olduğu gibi biz *deque* 'in de bir elemanına erişebiliriz. *deque* veri yapısı bloklu bağlı liste olduğu için herhangi bir pozisyona yapılan *insert* ve *delete* işlemleri veri yapısının tümünü etkilemez. Kaydırma işlemi yalnızca tek bir bloğu etkileyecek biçimde daha hızlı yapılabilir, yani genel olarak *deque* de *insert* ve *delete* işlemleri *vector* sınıfına göre daha etkin yapılmaktadır. Fakat standartlara göre başa ve sona yapılan *insert* ve *delete* işlemleri ek maliyetli sabit zamanlıdır, araya aypılan *insert* ve *delete* işlemleri doğrusal zamanlıdır. Buradan da anlaşılacağı gibi *deque* sınıfında *vector* sınıfında olmayan *push_front*, *pop_front* ve *front* fonksiyonları vardır.

```

namespace std {
template <class T, class Allocator = allocator<T> >
class deque {
...
};

```

push_front öne ekleme için, *pop_front* önden silme yapmak için *front* ise öndeki elemanı almak için kullanılır. *deque* sınıfının *insert* ve *erase* fonksiyonları tamamen *vector*

sınıftakilerle aynı parametrik yapıya sahiptir. *deque* sınıfında *capacity* kavramı yoktur, yani bu isimde bir fonksiyon mevcut değildir, fakat *size* kavramı vardır. Anımsanacağı gibi *vector* sınıfından eleman silsek bile tahsis edilen alan küçültülmemektedir, oysa *deque* sınıfı küçültme yapabilir.

deque Sınıfı Ne zaman Tercih Edilmelidir?

deque sınıfı tercih bakımından en çok *vector* sınıfıyla karışabilir. Eğer rasgele erişim isteniyorsa tercih *vector* ya da *deque* sınıfı olaraktır. Bu durumda programcı sistemin diğer özelliklerine bakmalıdır. Örneğin, sistemde hem başa hem de sona ekleme olayları sıkça gerçekleşiyorsa *deque* sınıfı yerine *vector* tercih edilmelidir. Ardışıl alan problemi söz konusu ise tercih *deque* sınıfından yana olmalıdır. *deque* sınıfının genel olarak belleği daha etkin kullandığı fakat *vector*'den biraz daha yavaş olduğu dikkate alınmalıdır.

deque sınıfının da klasik üç başlangıç fonksiyonu vardır. Default başlangıç fonksiyonu olarak kullanılabilecek fonksiyon boş bir *deque* yaratır. Belili bir *size* değerine sahip *deque* yaratılabilir ya da iki iteratorle belirtilen bir dizilimden *deque* oluşturulabilir.

deque sınıfı `<deque>` başlık dosyası içerisindedir.

```
#include <deque>
```

```
using namespace std;
```

```
int main(void)
```

```
{
```

```
    deque<int> a;
```

```
    for (int i = 1; i <= 10; ++i)
        a.push_back(i);
```

```
    for (int k = 100; k <= 1000; k += 100)
        a.push_front(k);
```

```
    copy(a.begin(), a.end(), ostream_iterator<int>(cout, "\n"));
```

```
    return 0;
```

```
}
```

list Sınıfı

Nesne tutan *list* sınıfı klasik bir bağlı liste sınıfıdır. *list* sınıfı klasik nesne tutan sınıfların tüm özelliklerini taşımaktadır. İteratorleri çift yönlü ilerleyen iteratorlerdir, yani biz bu iteratorlerle ++ ve -- operatorlerini kullanarak bağlı listede dolaşabiliriz.

list sınıfında *push_back* fonksiyonu sona ekleme yapmak için, *pop_back* fonksiyonu sondaki elemanı silmek için, *push_front* fonksiyonu başa eleman eklemek için, *pop_front* fonksiyonu ise baştaki elemanı silmek için kullanılır. *front* fonksiyonu baştaki elemanı, *back* fonksiyonu sondaki elemanı elde eder. Başa ve sona yapılan tüm işlemler hızlı yani sabit zamanlıdır. *list* sınıfının da iterator pozisyonuna insert işlemi yapan iki iterator arasındaki bilgileri silen *insert* ve *erase* fonksiyonları vardır. Sınıfın [] operator fonksiyonu yoktur.

Yine *size* fonksiyonu bağlı listedeki eleman sayısını elde etmek için kullanılır. Sınıfın *capacity* fonksiyonu yoktur.

Nesne Tutan Sınıfların Standartlarda Ele Alınması

Standartlarda nesne tutan sınıflar 23. bölümde ele alınmıştır. Burada sınıfların tek tek ele alınarak tüm üye fonksiyonlarının açıklanması yerine daha çok ortak özelliklere dayalı bir açıklama yoluna gidilmiştir. Bölümün başında tüm nesne tutan sınıfların sahip olması gereken ortak fonksiyonlar ve tür isimleri "*Container Requirements*" başlığında ele alınmıştır. Örneğin, buradaki anlatıma göre *begin* ve *end* iterator veren fonksiyon tüm nesne tutan sınıflarda olmak zorundadır. Yine tüm nesne tutan sınıfların başlangıç fonksiyonu, kopya başlangıç fonksiyonu olmak zorundadır. Ayrıca tüm nesne tutan sınıfların tutulan elemanın türünü gösteren *value_type* isimli bir typedef ismi, *iterator* ve *const_iterator* isimli typedef isimleri, *size_type* isimli genel bir typedef ismi, iki iterator arasındaki eleman sayısını göstermekte kullanılacak *difference_type* typedef ismi bulunmak zorundadır. Tüm nesne tutan sınıflarda bulunması gerekenler tablo-65' te özet olarak belirlenmiştir.

expression	return type	assertion/note pre/post-condition	complexity
<code>X::value_type</code>	T	T is Assignable	compile time
<code>X::reference</code>	lvalue of T		compile time
<code>X::const_reference</code>	const lvalue of T		compile time
<code>X::iterator</code>	iterator type pointing to T	any iterator category except output iterator. convertible to <code>X::const_iterator</code> .	compile time
<code>X::const_iterator</code>	iterator type pointing to const T	any iterator category except output iterator.	compile time
<code>X::difference_type</code>	signed integral type	is identical to the difference type of <code>X::iterator</code> and <code>X::const_iterator</code>	compile time
<code>X::size_type</code>	unsigned integral type	<code>size_type</code> can represent any non-negative value of <code>difference_type</code>	compile time
<code>X u;</code>		post: <code>u.size() == 0</code> .	constant
<code>X();</code>		<code>X().size() == 0</code> .	constant
<code>X(a);</code>		<code>a == X(a)</code> .	linear
<code>X u(a);</code> <code>X u = a;</code>		post: <code>u == a</code> . Equivalent to: <code>X u; u = a;</code>	linear
<code>(&a) -> ~X();</code>	void	note: the destructor is applied to every element of a; all the memory is deallocated.	linear
<code>a.begin();</code>	iterator; const_iterator for constant a		constant
<code>a.end();</code>	iterator; const_iterator for constant a		constant
<code>a == b</code>	convertible to bool	<code>==</code> is an equivalence relation. <code>a.size() == b.size()</code> <code>&& equal(a.begin(), a.end(), b.begin())</code>	linear
<code>a != b</code>	convertible to bool	Equivalent to: <code>!(a == b)</code>	linear
<code>a.swap(b);</code>	void	<code>swap(a,b)</code>	(Note A)

Standartlarda daha sonra geriye doğru iterator hareketi sağlayan sınıflar reversible container olarak isimlendirilmiştir ve bu sınıfların bulundurma gereken üye fonksiyonlar ve typedef isimleri de "*Reversible Container Requirements*" başlığı altında ele alınmıştır.

Standartlarda nesne tutan sınıflar *dizisel (sequences)*, *ilişkisel (associative)* olmak üzere ikiye ayrılmıştır. *vector*, *deque* ve *list* sınıfları dizisel nesne tutan sınıflardır. Dizisel özellik gösteren nesne tutan sınıfların bulundurma gereken üye fonksiyonlar "*Sequence Requirements*" başlığı altında ele alınmıştır. Bu durumda örneğin *vector* hem "*container requirement*" başlığında belirtilen özellikleri hem "*Reversible Container Requirements*" başlığı altında altındaki özellikleri hem de "*sequence requirements*" başlığı altındaki özellikleri barındırmak zorundadır. Bu durumda insert ve erase fonksiyonlarının hepsi birer

dizisel sınıf olan vector, deque ve list sınıflarında aynı biçimdedir. Yani vector' un insert fonksiyonlarının parametrik yapısıyla, list sınıfının insert fonksiyonlarının parametrik yapısı aynı biçimdedir.

En sonunda ilişkisel nesne tutan sınıfların ortak barındırması gereken üye fonksiyonlarda "Associative Container Requirements" başlığı altında ele alınmıştır.

İlişkisel Nesne Tutan Sınıflar

Nesne tutan sınıflar *dizisel (sequence)* ve *ilişkisel (associative)* olmak üzere ikiye ayrılmaktadır. vector, list ve deque sınıfları dizisel sınıflardır. Fakat set, multiset, map ve multimap sınıfları ilişkisel nesne tutan sınıflardır.

İlişkisel nesne tutan sınıflarda anahtar ve değer kavramları vardır. Değer, bir anahtar bilgiye göre saklanır ve ona göre de geri alınır. set, multiset, map ve multimap sınıflarında anahtar bilgiler sınıf içerisinde sıralı bir biçimde tutulmaktadır.

Standartlarda ilişkisel nesne tutan sınıfların ortak özellikleri "Associative Container Requirements" başlığı altında ele alınmıştır.

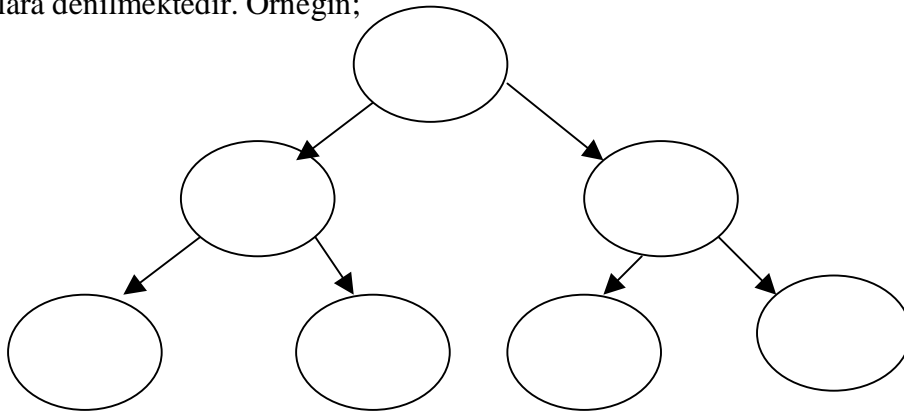
İlişkisel nesne tutan sınıfların iteratorleri çift yönde ilerleyen (bi-directional) iteratorlerdir. *i* ve *j* ilişkisel nesne tutan sınıflara ilişkin iki iterator olsun. *j*, *i*' den daha sonra erişilebilir ise **i* değeri **j* değerinden default olarak daha küçüktür. Başka bir deyişle bilgiler bu nesne tutan sınıflarda algoritmik bir sıra da oluşturmaktadır.

İlişkisel nesne tutan sınıflarda anahtar bilginin türünü belirten *key_type* isimli bir typedef ismi anahtar değerlerin karşılaştırılmasında kullanılan *key_compare* typedef ismi bulunmaktadır.

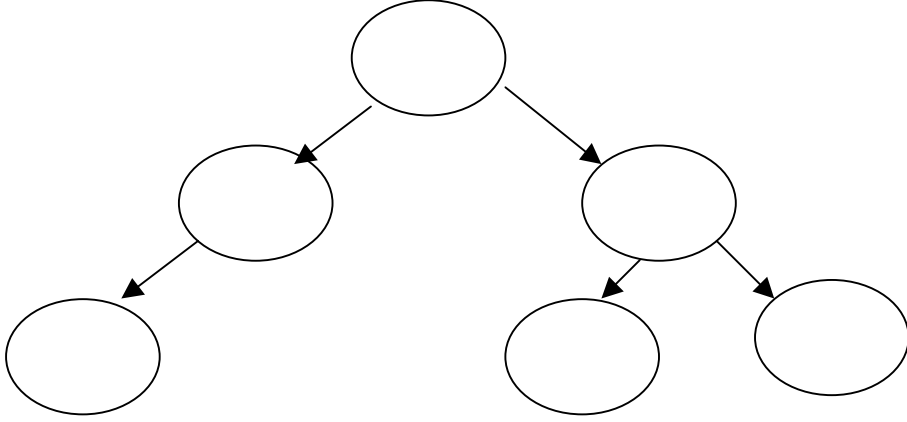
İlişkisel nesne tutan sınıflar için nasıl bir veri yapısı ve algoritma kullanılacağı standartlarda belirtilmemiştir. Fakat bu sınıflar tipik olarak dengelenmiş ikili ağaç (balanced binary tree) veri yapısını kullanmaktadır.

İkili Ağaçlar ve Dengelenmiş İkili Ağaçlar

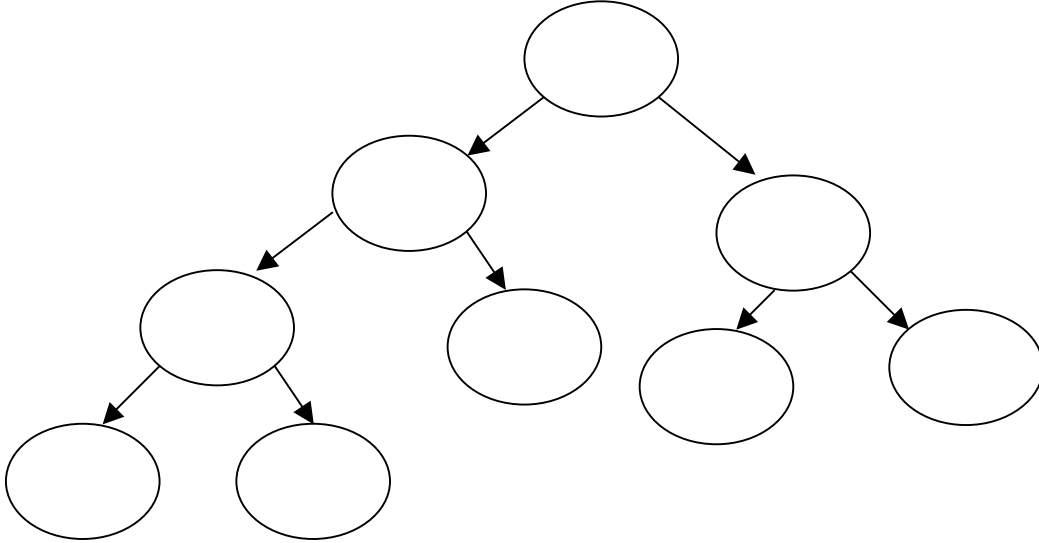
Elemanların sıralı bir biçimde ifade edilebildiği genel veri yapılarına *liste (list)* denilebilmektedir. Bu açıdan bakıldığında diziler bile bir listedir. Önceki elemanın sonraki elemanı gösterdiği özel listelere *bağlı listeler (linked lists)* denilmektedir. Bir elemanın tek bir eleman tarafından gösterildiği fakat o elemanın birden fazla elemanı gösterebildiği genel listelere ise *ağaç* denilmektedir. Ağaçların elemanlarına *düğüm (node)* denilebilmektedir. Gösteren düğüme *üst düğüm (parent node)*, gösterilen düğüme *alt düğüm (child node)* denir. *İkili ağaç (binary tree)* özel bir ağaçtır, her düğümün en fazla iki alt düğüme sahip olduğu özel ağaçlara denilmektedir. Örneğin;



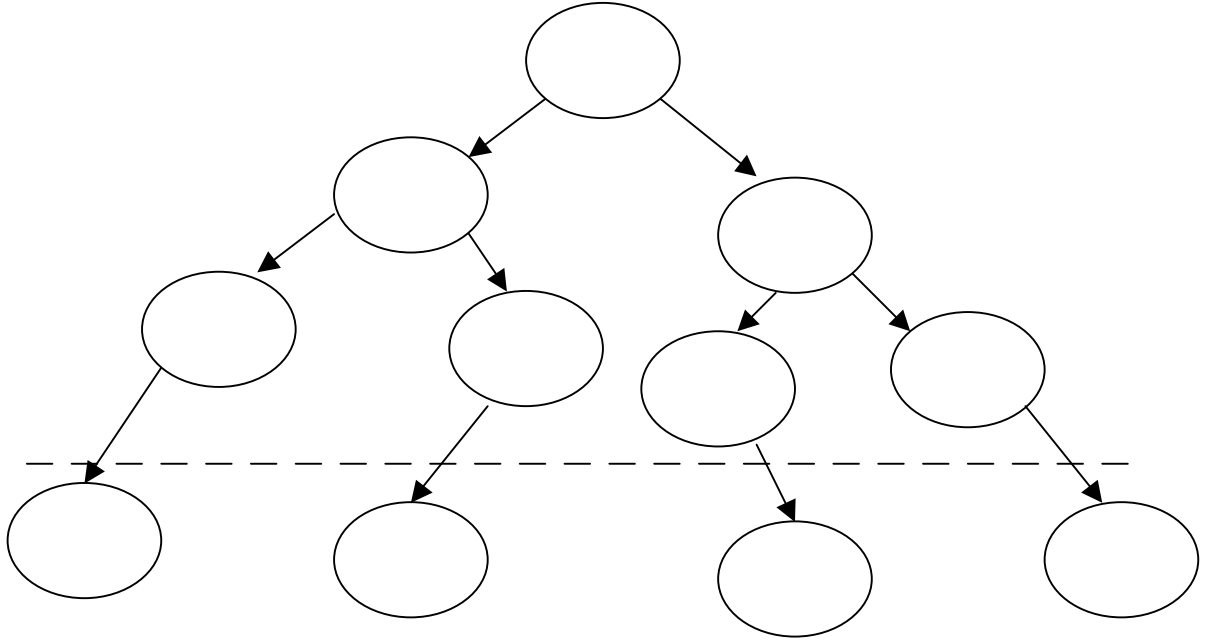
İkili ağaç olabilmesi için her düğümün kesinlikle iki alt düğümünün bulunması zorunlu değildir. Fakat her düğümün en fazla iki alt düğümü olabilir. Aşağıdaki ağaçta bir ikili ağaçtır.



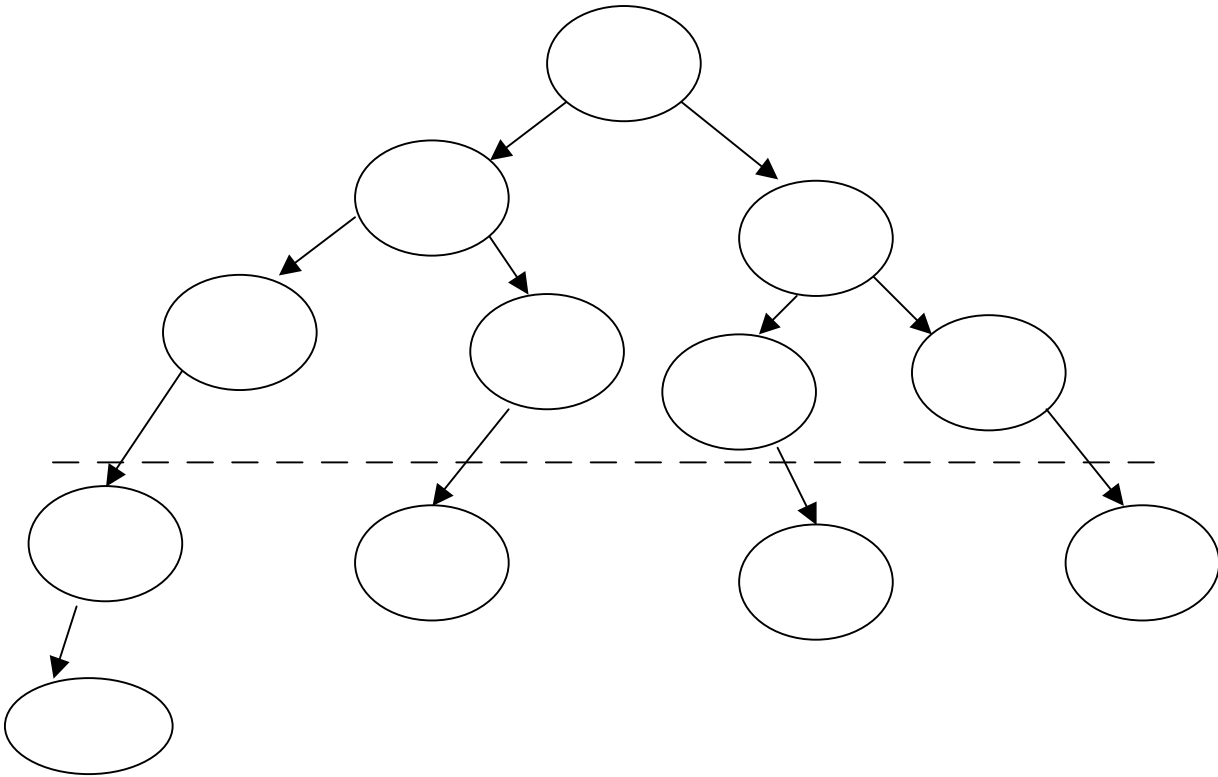
Bir ikili ağacın yüksekliği onun en uzun dalının düğüm sayısıdır. Örneğin aşağıdaki ağacın yüksekliği 4' tür.



Ağacın en alt kademe atıldığında tüm dallarındaki yüksekliği aynı ise bu tür ağaçlara dengelenmiş ağaçlar denilmektedir. Dengelenmiş ağaçlarda en alt düğümler ve onların üst düğümleri dışında bütün düğümlerin kesinlikle iki elemanı olmak zorundadır. Başka bir deyişle en alt kademedeki düğümler çıkartıldığında ağacın yüksekliği her koldan aynı değerde kalıyorsa böyle ağaçlara dengelenmiş ağaçlar denir.

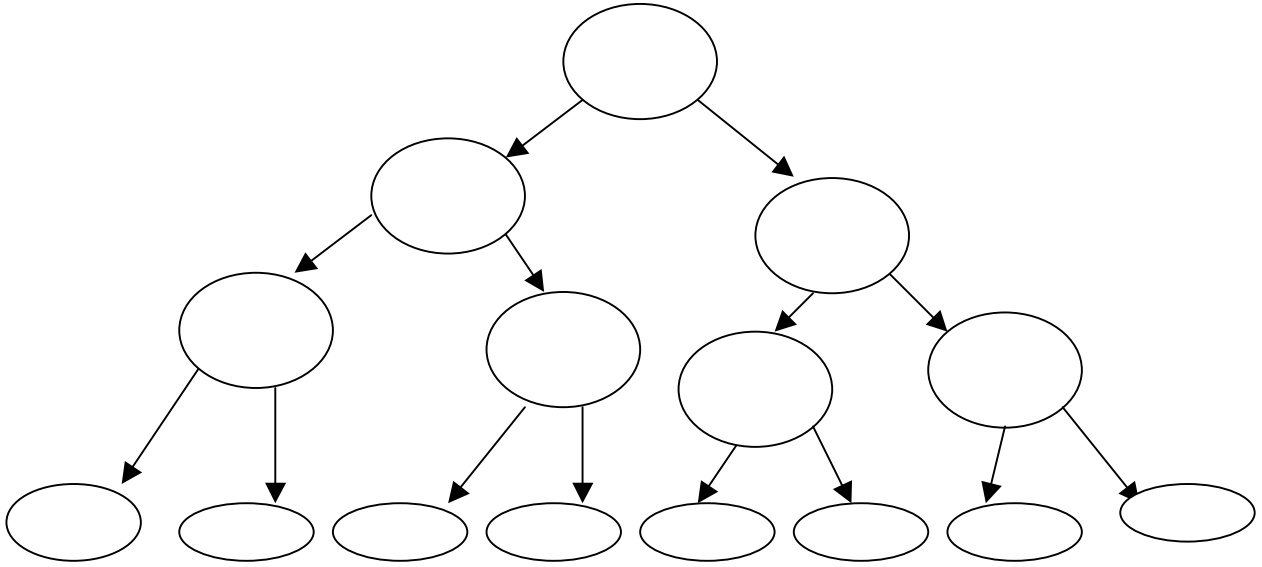


Aşağıdaki ağaç dengelenmiş ikili ağaç değildir.



Görüldüğü gibi en alt kademe dışındaki tüm dalların aynı yükseklikte olması en alt kademe ve onun bir yukarısındaki kademe dışındaki düğümlerin hepsinin tam olarak iki alt düğüme sahip olmasını gerektirmektedir.

Dengelenmiş bir ikili ağaçta en alt kademe düğümlerinde hepsi doluysa bu tür ağaçlara tam dengelenmiş ikili ağaçlar denir. Örneğin



Yüksekliği n olan tam dengelenmiş ağacın eleman sayısı $2^n - 1$ dir.

İkili ağacın bir düğümü C' de aşağıdaki gibi bir yapıyla temsil edilebilir.

```
struct NODE{
    DATATYPE val;
    struct NODE *pLeft;
    struct NODE *pRight;
};
```

İkili ağaç ve genel olarak ağaç yapıları bir elemanı bir anahtar bilgi eşliğinde algoritmik olarak yerleştirip hızlı bir biçimde geri almak için kullanılmaktadır. Yani ağaç yapılarının algoritmik anlamı arama konusuna ilişkindir. bu nedenle algoritmalar dünyasında ikili ağaç kavramı yerine ikili arama ağacı (binary search tree) kavramı tercih edilmektedir. Bilindiği gibi rasgele elemanlardan oluşan bir dizilimde arama işlemi için eleman bulunana kadar hepsine bakma yöntemi kullanılır. Bu işlemin ortalama karmaşıklığı $(2^n - 1)/2$ ' dir. Eğer elemanlar sıralıysa ve dizilim rasgele erişimli iteratorlere sahipse ikili arama denilen yöntem en hızlı yöntemdir. Karmaşıklığı $\log_2 n$ ' dir. İşte ikili arama ağacı elemanlar sıralı omadığı halde onları sıralıymış gibi tutan ve böylece ikili arama yönteminin uygulanmasına olanak sağlayan bir ağaç yöntemidir.

Aslında yukarıda şekilsel olarak gösterilen ağaç çizimlerinde her düğüm yalnızca alt düğümleri tutmaz. Her düğümün içinde *anahtar (key)* ve *değer (value)* biçiminde iki bilgi de bulunur. Bu durumda bir düğümün daha gerçekçi temsili şöyle olur:

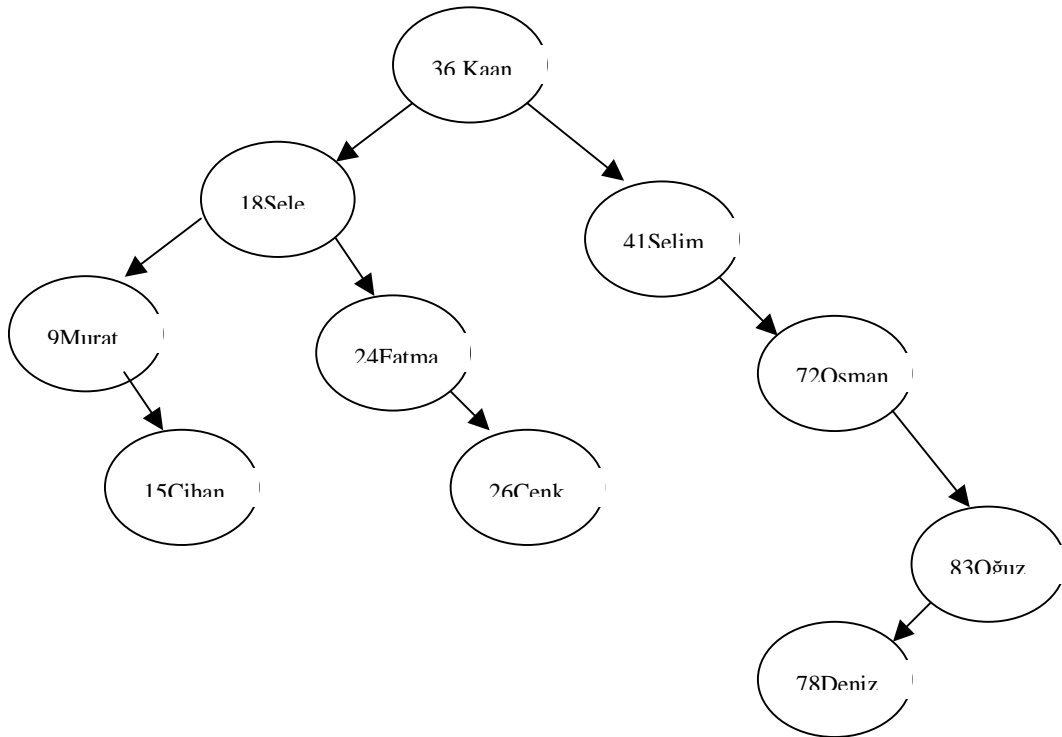
```
struct NODE{
    KEY key;
    VALUE value;
    struct NODE *pLeft;
    struct NODE *pRight;
};
```

Burada *KEY* ve *VALUE* basit ya da bileşik herhangi iki türü temsil etmektedir.

Dengelenmemiş ikili ağaçlarda eleman yerleştirme işlemi basit bir biçimde anahtara göre küçük olanlar sola, büyük olanlar sağa biçiminde yapılabilir. Örneğin, aşağıdaki 10 şahıs sırasıyla ağaca yerleştirilecek olsun.

No	İsim
36	Kaan
41	Selim
18	Selen
24	Fatma
72	Osman
83	Oğuz
26	Cenk
9	Murat
15	Cihan
78	Deniz

Burada anahtar değer isim ise yukarıdaki listedeki elemanlar ağaca şöyle yerleştirilir.



İkili ağaçta arama oldukça kolaydır. Ağacın kökü bir göstericide tutulur, aranacak değer büyükse sağa küçükse sola doğru ilerlenir. Yukarıdaki ağaç dengelenmiş değildir. Bir eleman eklendiğinde eğer denge bozuluyorsa dengeleme işlemini yapan klasik bir algoritmik kalıp vardır. Böylece elemanlar ağacın dengesi bozulmadan eklenebilmektedir. Dengelenmiş bir ikili ağaçta ağaçtaki tüm eleman sayısı n olmak üzere eleman araması için gereken işlem sayısı en olasılıkla kötü $\log_2 n$ ' dir. Dengelenmiş ikili ağaca eleman eklenmesi için önce elemanın yerinin bulunması gerekir. Ayrıca dengeleme işleminin ek bir takım işlemler gerektireceği açıktır. Eleman ekleme karmaşıklığı da logaritmiktir.

C++' ın standart kütüphanesindeki ilişkisel nesne tutan sınıflar tipik olarak dengelenmiş ikili ağaç yapısını kullanmaktadır.

Dengelenmiş İkili Ağaçlarla Bağlı Listelerin Karşılaştırılması

İlk bakışta dengelenmiş ikili ağaçların bağlı listelere karşı kesin bir üstünlüğü varmış gibi görünmektedir. Gerçekte ikili ağaçların asıl üstünlüğü arama konusundadır. Bağlı listelerde arama doğrusal yapılabilirken dengelenmiş ikili ağaçlarda logaritmik yapılmaktadır. Bu avantajına karşın ikili ağaca eleman ekleme algoritmik bir işlemdir. Yani elemanın yeri bulunduğundan sonra ekleme yapılır. Halbuki bağlı listelerin başına ya da sonuna elemanlar sabit zamanlı olarak eklenebilir. Bu durumda ekleme işleminin çok sık fakat arama işleminin seyrek yapıldığı durumlarda bağlı listeler, ekleme işleminin seyrek arama işleminin çok sık yapıldığı durumlarda dengelenmiş ikili ağaçlar tercih edilmelidir.

map Sınıfı

map sınıfı tipik bir ilişkisel nesne tutan sınıftır.

```
template <class Key, class T, class Compare = less<Key>,
class Allocator = allocator<pair<const Key, T> > >
class map{
    //...
};
```

Görüldüğü gibi sınıfın dört template parametresi vardır. Anahtar bilgiyi temsil eden tür, değer bilgisini temsil eden tür belirtilmek zorundadır. Örneğin;

```
map<int, string> a;
```

```
map< string, person> b;
```

Birinci nesne *int* bir bilgiye göre *string* aramaktadır. İkinci nesne muhtemelen kişinin isminden hareketle onun bilgilerini aramakta kullanılacaktır. Sınıfın üye fonksiyonları ağaca yerleştirme yaparken ve arama yaparken karşılaştırma yapmak zorundadır. İşte üçüncü template parametresinde anahtar değerlerin karşılaştırılmasında kullanılacak olan karşılaştırma fonksiyonu belirtilir. Görüldüğü gibi default olarak kullanılacak karşılaştırma fonksiyonu *less* fonksiyon nesnesidir. *less* fonksiyon nesnesi ise küçüktür karşılaştırmasını < operatorü kullanarak yapmaktadır. O halde default kullanılacak operator < operatorüdür.

Yazılacak karşılaştırma fonksiyonunun katı ve zayıf sıralama bağıntısı (strict weak ordering relation) oluşturması gerekir. *Katı (strict)*, sıralama bağıntısının yansıma özelliğine sahip olmayacağını belirtmektedir. Yani karşılaştırma fonksiyonu *Comp* ise *Comp(a,a)* false değerini vermelidir. Sıralama bağıntısı ters simetrik olması gereken bir bağıntıdır. Bu durumda *Comp(a, b)=true* ise *Comp(a, b)=false* değeri vermelidir. Burada sözü edilen bağıntı $B = \{(a,b) | Comp(a,b) = true\}$ koşulunu sağlayan sıralı ikililerinin oluşturduğu bağıntıdır. Görüldüğü gibi gerçek sayılar kümesinde *Comp* yerine < operatorü kullanılırsa oluşan bağıntı bu koşulları içermektedir. *Zayıf (weak)* kavramı burada kısmi sıralama bağıntısını anlatmak için kullanılmıştır. Yani her (a,b) ikilisinin bağıntıda bulunması gerekmemektedir.

map Sınıfında Ağaca Eleman Yerleştirme

Sınıfın [] operator fonksiyonu index değeri olarak anahtar değeri alır. Anahtara karşı gelen değer referansıyla geri döner. *map* aynı anahtara ilişkin birden fazla değeri tutmaz. Bu nedenle aynı anahtar değerine atama yapılırsa eski değer gider yerine yenisi yerleştirilir. Örneğin;

```
map<int, string> a;
```

```
a[100] = "ali";
```

```
a[50] = "veli";
```

"a[key] işleminde neler olmaktadır?" Operatör fonksiyonu önce key değerine ilişkin bir düğümün ağacda bulunup bulunmadığına bakar. Eğer anahtar ağacda bulunuyorsa o anahtara karşılık gelen değer referansı ile geri döner. Eğer anahtara ilişkin bir düğüm ağacda yoksa anahtarı bu değer olan yeni bir düğüm yaratır. O düğümü ağaca monte eder ve o düğümün değerine ilişkin referansla geri döner. anahtar ve değer herhangi bir türden olabilir. Örneğin;

```
map<string, int> a;
```

```
a["ali"] = 100;
```

```
a["veli"] = 50;
```

```
cout << a["ali"] << endl;
```

map Sınıfının İteratorleri

Aslında *map* sınıfı düğüm içerisinde elemanları key ve value değerlerinden oluşan bir *pair* nesnesi olarak saklar. *map* sınıfının iteratorleri çift yönde ilerleyebilen iteratorlerdir. *iter* bir *map* iteratoru olmak üzere **iter* bir *pair* nesnesi belirtir. Yani **iter* düğümdeki elemandır, fakat düğümdeki eleman bir *pair* çifti olarak bulunmaktadır. O halde *iter* bir *map* iteratoru olmak üzere bu iteratorun gösterdiği yerdeki bilginin anahtar değeri (**iter*).*first* ya da *iter->first* ile elde edilir. Anahtara karşı gelen değer ise (**iter*). *second* ya da *iter->second* ile elde edilir.

```
map<string, int> a;
```

```
a["ali"] = 100;
```

```
a["veli"] = 50;
```

```
a["osman"] = 300;
```

```
a["ahmet"] = 500;
```

```
a["jale"] = 72;
```

```
a["iskender"] = 87;
```

```
for (map<string, int>::iterator iter = a.begin(); iter != a.end(); ++iter)
    cout << iter->first << " " << iter->second << endl;
```

```
return 0;
```

Görüldüğü gibi *map* sınıfında iteratorlerle ilerlendiğinde küçükten büyüğe anahtara göre sıralı bir dizilim elde edilir. *map* sınıfını iteratorleri ile geriye doğru da gidilebilir. Bu sınıfın iteratorleri ters yönde ilerleyen iteratorleri destekler.

```
for(map<string, int>::reverse_iterator iter = a.rbegin(); iter!=a.rend(); iter++)
```

map Sınıfının insert Fonksiyonları

insert algoritmik olarak anahtara göre ağaçtaki yeri bulup ekleme işlemi yapar. *insert* işleminde insert edilecek anahtar ve değer bir hamlede pair çifti olarak verilir.

```
iterator insert(iterator it, const value_type& x);
```

Burada *value_type*, key ve value çiftinden oluşan *pair* nesnesini temsil etmektedir. Fonksiyon yeni elemanın iterator değerine geri döner. Bu *insert* fonksiyonu diğer ilişkisel nesne tutan sınıflarda da aynı biçimde vardır. Fonksiyonun birinci parametresindeki iterator değerinin aranmaya başlayacağı düğümün iteratorunu belirtmektedir. normal olarak bu parametre ilk elemanın iterator değeri olarak verilir. Fakat bu fonksiyon yerine daha basit bir insert fonksiyonu vardır.

```
pair<iterator, bool> insert(const value_type& x);
```

Burada parametre insert edilecek pair nesnesidir. Fonksiyonun geri dönüş değeri işlemin başarısı ve insert edilen elemanın iterator değerini bir *pair* çifti olarak verir. Eğer insert edilmek istenen elemanın anahtar değeri daha önce ağaçta varsa insert fonksiyonu birşey yapmaz ve başarısızlıkla geri döner. Bu *insert* fonksiyonu da diğer tüm ilişkisel nesne tutan sınıflarda vardır.

```
map<string, int> a;
```

```
pair<map<string, int>::iterator, bool> result = a.insert(make_pair("ali", 300));
```

Nihayet başka bir dizilimin iki iterator arası bir hamlede insert edilebilir. Burada dizilimin uygun pair nesnelerinden oluşması gerekmektedir.

Arama Fonksiyonları

Arama fonksiyonları da bütün ilişkisel nesne tutan sınıflarda aynı biçimdedir. Arama işlemindeki tipik fonksiyon *find* fonksiyonudur.

```
iterator find(const Key& key);
```

```
iterator find(const Key& key) const;
```

Fonksiyonun parametresi anahtar değeridir. Eğer arama işlemi $\log_2 n$ karmaşıklığında yapılır, değer bulunursa bulunulan değerın iterator değerine bulunamazsa *end* iterator değerine dönlür.

Diğer map fonksiyonları *multimap* sınıfı ile ele alınacaktır.

multimap Sınıfı

multimap sınıfı tamamen *map* sınıfının aynı değerde anahtarları destekleyen biçimidir. Yani map sınıfı için söz konusu olan fonksiyonların hemen hepsi *multimap* sınıfı için de geçerlidir. Bazı fonksiyonların davranış farklılığı vardır, bunlar ele alınacaktır.

multimap sınıfında `[]` operator fonksiyonu yoktur. Eleman ekleme işlemi *insert* fonksiyonlarıyla yapılabilir. *insert* fonksiyonları aynı anahtar verildiğinde başarısız olmaz. Örneğin;

```
multimap<int, string> c;
```

```
c.insert(pair<int, string> (100, "ali"));
```

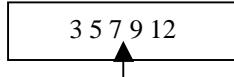
```
c.insert(pair<int, string> (100, "veli"));
```

```
c.insert(pair<int, string> (100, "selami"));
```

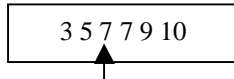
```
for (multimap<int, string>::iterator iter = c.begin(); iter != c.end(); ++iter)
    cout << iter->first << " " << iter->second << endl;
```

lower_bound ve upper_bound Fonksiyonları

Bu fonksiyonlar tüm ilişkisel nesne tutan sınıflarda olması zorunlu fonksiyonlardır. Yani bu fonksiyonlar *map*, *multimap*, *set*, *multiset* sınıflarında da vardır. *lower_bound* parametre olarak anahtar türünden bir değer alır. Dizilimde parametre olarak verilen anahtar değerden küçük olmayan ilk değere ilişkin iterator değeriyle geri döner. Örneğin, dizilimdeki anahtar değerler aşağıdaki gibi olsun ve biz 8 değerini parametre olarak verelim.



Bu fonksiyon *multimap* ya da *multiset* fonksiyonlarında kullanılırsa aynı olan değerlerin ilkinin ilişkili iterator elde edebilir. Örneğin, aşağıdaki dizilimde 7 değeri aranacak olsun:



upper_bound fonksiyonu ise parametre olarak verilen anahtar değerinden büyük olan ilk elemanın iterator değerini bulur. Örneğin aşağıdaki dizilimde parametre olarak 7 kullanılırsa 9'a ilişkin iterator değeri elde edilir. Örneğin bir *multimap* ya da *multiset* içerisinde *lower_bound* ve *upper_bound* fonksiyonuyla aynı anahtara sahip elemanların iterator aralığını elde edebiliriz. Zaten bu işlemi yapan *equal_range* isimli ayrı bir fonksiyon da vardır.

using namespace std;

```
int main(void)
{
    multimap<int, string> c;
    multimap<int, string>::iterator iter;

    c.insert(pair<int, string>(50, "selami"));
    c.insert(pair<int, string>(100, "ali"));
    c.insert(pair<int, string>(100, "veli"));
    c.insert(pair<int, string>(100, "selami"));
    c.insert(pair<int, string>(500, "selami"));

    iter = c.lower_bound(100);

    while (iter->first == 100) {
        cout << iter->second << endl;
        ++iter;
    }

    /*for (multimap<int, string>::iterator iter = c.begin(); iter != c.end(); ++iter)
        cout << iter->first << " " << iter->second << endl;
    */

    return 0;
}
```

set ve multiset Sınıfları

set ve *multiset* sınıfları da tıpkı *map* ve *multimap* sınıflarında olduğu gibi dengelenmiş ikili ağaç kurarlar. Fakat bu sınıflarda ağacın düğümlerindeki bilgiler bir pair biçiminde anahtar ve değer çiftlerinde oluşmazlar. Düğümlerdeki değerler tekil nesnelerden oluşmaktadır. Ağaca yerleştirme işlemi yine default olarak *less* fonksiyon nesnesi yani < operatorüyle yapılır. *set* ve *multiset* sınıflarının kullanımı, *map* ve *multimap* sınıflarının kullanımından daha kolaydır.

Örneğin biz anahtar bilgi string, değer bilgisi int olan bir ağaç kurmak isteyelim. Eğer bunu *map* sınıfı ile yaparsak anahtar ve değer türlerini template parametresi olarak belirtiriz. *map* sınıfı anahtar ve pair i bir map nesnesi olarak belirtir ve düğüme yerleştirir. *map* sınıfının iteratoru de birinci elemanı string olan, ikinci elemanı int olan bir pair nesnesine ilişkin olacaktır. Eğer biz bu örneği *set* kullanarak yapacak olursak önce string ve int elemana sahip olan bir sınıf oluşturmamız gerekir. *set* yapısına biz bu sınıf türünden nesneleri yerleştirmek üzere parametre olarak veririz. Tabi *set* düğümün ağaçtaki yerini bulabilmek için yarattığımız sınıfın < operatorünü kullanacaktır. O halde biz o sınıf için < operator fonksiyonunu stringleri karşılaştıracak biçimde yazmamız gerekir. Ayrıca *set*, aslında *map* sınıfından daha yeteneklidir. Örneğin biz yalnızca int bilgilerden *set* ile bir ağaç oluşturabiliriz, fakat *map* ile oluşturamayız. İşlevsel olarak *set* sınıfı adeta *map* sınıfını kapsıyor olsa da bazı durumlarda *map* çok daha kullanışlı olabilmektedir. Örneğin, string ve int çiftlerinden oluşan bir ağaç kurmak istiyorsak bunu *map* kullanarak çok kolay yapabiliriz. Tabi bu işlemi *set* kullanarak da yaparız fakat *set* için bir sınıf yaratmak ve < operator fonksiyonunu yazmak gerekir. O halde *set* ile *map* aynı amaçla kullanılmasına karşın duruma göre aralarında kullanım kolaylığı bakımından farklar olabilmektedir. *set* sınıfının template bildirimi şöyledir:

```
template <class Key, class Compare = less<Key>,
class Allocator = allocator<Key> >
class set {
...
}
```

Görüldüğü gibi yazılması zorunlu olan tek bir template parametresi vardır, o da düğüme saklanacak olan sınıf türüdür.

```
using namespace std;
```

```
int main(void)
{
    set<int> a;
    for (int i = 0; i < 100; ++i)
        a.insert(i);

    copy(a.begin(), a.end(), ostream_iterator<int>(cout, "\n"));

    return 0;
}
```

set sınıfının *insert* ve *erase* fonksiyonları *map* sınıfının gibidir. Anımsanacağı gibi biz *map* sınıfında *insert* fonksiyonunun parametresi olarak anahtar değeri pair çiftiyle birleştirip veriyorduk. Burada *insert* fonksiyonunun tek parametresi vardır, o parametre de hem anahtar hem de değeri içeren türdür.

SINIF ÇALIŞMASI:

Person isimli bir sınıf yazınız. Sınıfın bir elemanı *m_name* diğer elemanı *m_no* olsun. *m_name* string türünden, *m_no* ise int türündendir. Sınıfın başlangıç fonksiyonu isim ve numarayı alarak veri elemanlarına yerleştirecektir. Ayrıca sınıf için global bir ostream << fonksiyonu yazılacaktır. Yazılan bu sınıf türünden nesneleri isme göre sıralı olacak biçimde bir *set* sınıfı içerisinde saklayınız, sonra isime göre sıralı bir biçimde yazdırmak için *copy* algoritmasını *ostream_iterator* sınıfından faydalanarak kullanınız. Şüphesiz person sınıfı için bir < operator fonksiyonuna gereksinim vardır.

multiset sınıfı tamamen *set* sınıfı gibidir, fakat birden fazla aynı eleman sınıfa yerleştirilebilir.

Nesne Tutan Adaptor Sınıflar

Veri yapısı olarak başka bir sınıfı kullanan sınıflara *adaptor sınıflar* denilmektedir. Standart kütüphanedeki en önemli iki adaptör sınıf *stack* ve *queue* sınıflarıdır. Örneğin bir kuyruk sistemi *list* sınıfı kullanılarak da *deque* sınıfları kullanılarak da yazılabilir. Yani biz *list* ya da *deque* sınıflarını kullanarak bir *queue* sınıfı yazabiliriz.

using namespace std;

```
template <class T, class Container = deque<T> >
```

```
class Queue {
```

```
public:
```

```
    void PutQueue(const T &r)
```

```
    {
```

```
        m_c.push_back(r);
```

```
    }
```

```
    bool GetQueue(T &val)
```

```
    {
```

```
        if (m_c.size() == 0)
```

```
            return false;
```

```
        val = m_c.front();
```

```
        m_c.pop_front();
```

```
        return true;
```

```
    }
```

```
private:
```

```
    Container m_c;
```

```
};
```

```
int main()
```

```
{
```

```
    Queue<int> q;
```

```
    q.PutQueue(100);
```

```
    q.PutQueue(200);
```

```
    q.PutQueue(300);
```

```
    q.PutQueue(400);
```

```
    q.PutQueue(500);
```

```
    while (GetQueue)
```

```
    return 0;
}
```

queue Sınıfı

queue sınıfı default olarak *deque* sınıfını kullanan bir kuyruk sınıfıdır, bildirimi şöyledir:

```
template <class T, class Container = deque<T> >
class queue {
...
};
```

Birinci template parametresi kuyrukta saklanacak olan türü temsil eder. İkinci template parametresi ise kuyruk oluşturmada kullanılacak sınıfı belirtir, bu sınıf default olarak *deque* sınıfıdır. *queue* sınıfının şu fonksiyonları vardır:

empty fonksiyonu kuyruğun boş olup olmadığı bilgisine geri döner.

size fonksiyonu kuyruktaki eleman sayısını verir.

front fonksiyonu kuyruktaki sıradaki elemanı alır, fakat onu silmez.

pop fonksiyonu sıradaki elemanı silmek için kullanılır.

push fonksiyonu kuyruğa eleman yerleştirmek için sınıfın kullanılır.

Örneğin;

```
queue<int> q;

for(int i = 0; i < 100; ++i)
    q.push(i);
while(!q.empty()){
    cout << q.front() << endl;
    q.pop();
}
```

stack Sınıfı

Standart kütüphanede *stack* veri yapısı için *stack* isimli bir sınıf bulunmaktadır. Sınıfın bildirimi *<stack>* dosyası içindedir.

```
template <class T, class Container = deque<T> >
class stack {
...
};
```

Sınıfın fonksiyonları şunlardır:

empty fonksiyonu *stack*-in boş olup olmadığı bilgisine geri döner.

size fonksiyonu *stack*teki eleman sayısını verir.

top fonksiyonu *stack*teçekilecek elemanı verir, fakat onu silmez.

pop fonksiyonu *stack*te sıradaki elemanı silmek için kullanılır.

push fonksiyonu *stack*-e eleman yerleştirmek için kullanılır.

```
stack<int> s;
```

```
for(int i = 0; i < 100; ++i)
    s.push(i);

while(!s.empty()) {
    cout << s.top() << endl;
    s.pop();
}
```

Bağlayıcı Sınıflar ve Fonksiyonlar

Bağlayıcı sınıflar ve fonksiyonlar standart kütüphanenin en karmaşık öğelerindendir.

Anahtar Notlar:

Anımsanacağı gibi standart kütüphanede temel işlemleri yapan bir takım fonksiyon nesne sınıfları vardır. Bu fonksiyon nesne sınıfları unary_function ve binary_function sınıflarından türetilmişlerdir. unary_function ve binary_function sınıflar yalnızca okunabilirliği arttırmak için kullanılan typedef isimlerine sahip sınıflardır.

Bağlayıcı sınıflar tek parametrelili bir fonksiyon nesnesi görevi yaparlar. Yani X bir bağlayıcı sınıf türünden nesne ise X(a) gibi bir çağırma yapılabilir. Bağlayıcı sınıflar tek parametrelili bir fonksiyon gibi çalışırken bizim verdiğimiz iki parametrelili bir fonksiyonu da çağırırlar. İki önemli bağlayıcı sınıf vardır: *binder1st* ve *binder2nd*. *binder2nd*, *binder1st* sınıfından daha fazla kullanılmaktadır. *binder2nd* sınıfının bildirimi şöyledir.

```
template <class Operation>
class binder2nd
: public unary_function<typename Operation::first_argument_type,
typename Operation::result_type> {
protected:
    Operation op;
    typename Operation::second_argument_type value;
public:
    binder2nd(const Operation& x,
    const typename Operation::second_argument_type& y);
    typename Operation::result_type
    operator()(const typename Operation::first_argument_type& x) const;
};
```

Sınıfın başlangıç fonksiyonunun birinci parametresi iki parametrelili bir fonksiyon nesnesi, ikinci parametresi ise bu iki parametrelili fonksiyon nesnesi çağırılırken kullanılacak ikinci parametredir. Sınıfın template parametresi bu fonksiyon nesnesinin türüdür.

binder2nd sınıfı tek parametrelili bir fonksiyon nesnesi olarak kullanılır. Yani bu sınıfın tek parametrelili bir fonksiyon çağırma operator fonksiyonu vardır. Fakat bu operator fonksiyonu içerisinde bizim verdiğimiz iki parametrelili fonksiyon çağırılır. Başlangıç fonksiyonunda verdiğimiz ikinci parametre iki parametrelili fonksiyon çağırılırken onun ikinci parametresi yapılmaktadır, zaten second ismi buradan gelmektedir. Özetle, *binder2nd* tek parametrelili fonksiyon gibi çalışan fakat kendi içinde çift parametrelili fonksiyon işlemi yapan tuhaf bir sınıftır. Örneğin;

```
binder2nd<greater<int> > op(greater<int>(), 50);
    if(op(100)){
}

```

Bu işlemin eş değeri;

```
if((greater<int>(100, 50)){
    ...
}

```

Örneğin, bir vektördeki 100' den büyük olan değerleri 1000 ile değiştirelim.

```
replace_if(v.begin(), v.end(), binder2nd<greater<int> >(greater<int>(), 100), 1000);

```

Burada aslında *binder2nd<greater<int> >(greater<int>(), 100)* nesnesi tek parametrelili bir fonksiyon nesnesi görevindedir. Yani bu ifade bir sınıf nesnesidir ve *replace_if* fonksiyonu dizilimdeki her değeri bu fonksiyona parametre olarak geçirerek bu fonksiyonu çağırır. Tabi burada aslında *greater* fonksiyonu çağırılmış olacaktır. Buradaki 100 *greater* fonksiyonu her çağırıldığında onun ikinci parametresi yapılacaktır. Şüphesiz bu işlemin yapılabilmesi için *binder2nd* sınıfı *greater* fonksiyonunu ve 100 değerini veri elemanlarında saklamalıdır.

SINIF ÇALIŞMASI:

binder2nd sınıfını kullanarak bir vektördeki tek olan elemanları *remove_if* fonksiyonu ile tek hamlede siliniz. Anımsanacağı gibi *remove_if* fonksiyonu tek parametrelili bir predicate fonksiyon almaktadır.

```
using namespace std;

```

```
int main(void)
{
    vector<int> v;

    for (int i = 0; i < 100; ++i)
        v.push_back(i);

    vector<int>::iterator iter = remove_if(v.begin(), v.end(),
        binder2nd<modulus<int> >(modulus<int>(), 2));

    v.erase(iter, v.end());

    copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));

    return 0;
}

```

SINIF ÇALIŞMASI:

İstediğiniz türden bir dizilim alınız, dizilimin ortalamasını bulunuz. Sonra ortalama küçükle olanları dizilimden siliniz. Ortalama bulmak için aşağıdaki fonksiyon kullanılabilir.

```
template <class InputIterator, class T>

```

```
T accumulate(InputIterator first, InputIterator last, T init);

```



```
using namespace std;
```

```
int main(void)
{
    int a[] = {3, 3, 3, 3, 3, 4, 4, 4, 4, 4};
    vector<int> v(a, a + 10);

    int sum = accumulate(v.begin(), v.end(), 0);
    double avg = (double) sum / v.size();
    // double avg = accumulate(v.begin(), v.end(), 0.) / v.size();
    cout << avg << endl;

    vector<int>::iterator iter = remove_if(v.begin(), v.end(),
                                          binder2nd<less<double>>(<less<double>(), avg));

    v.erase(iter, v.end());

    copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));

    return 0;
}
```

bind2nd isimli fonksiyon *binder2nd* sınıfının kolay bir biçimde kullanılması için düşünülmüştür. Bu fonksiyonun geri dönüş değeri *binder2nd* türündendir. Fonksiyon şöyle yazılmıştır.

```
template <class Operation, class T>
binder2nd<Operation> bind2nd(const Operation& op, const T& x){
    return binder2nd<operation>(op, x);
}
```

Bu durumda örneğin,

```
bind2nd(greater<int>(), n);
```

ile

```
binder2nd<igreater<int>>(<greater<int>(), n);
```

eşdeğerdir.

Standart kütüphanede tamamen benzer biçimde çalışan *binder1st* isimli bir sınıf ile *bind1st* isimli bir fonksiyon da vardır. Bu sınıfın ve fonksiyonun *binder2nd* sınıfı ve *bind2nd* fonksiyonundan tek farkı programcının verdiği parametrenin ikinci değil birinci parametre yapılarak fonksiyonun çağırılmasıdır. Örneğin,

```
remove_if(v.begin(), v.end(), bind1st(greater<int>(), 100));
```

Görüldüğü gibi burada eğer 100 dizilimdeki elemandan büyükse silinme gerçekleşecektir. Başka bir deyişle 100' e eşit ya da 100' den küçük olanlar silinecektir. Bu işlemin eşdeğeri şöyledir:

```
remove_if(v.begin(), v.end(), bind2nd(less_equal<int>(), 100));
```

copy_if Fonksiyonu

Pekçok fonksiyonun predicate alan *_if* li biçimi olduğu halde maalesef *copy_if* biçiminde standart bir fonksiyon yoktur. Bu standart kütüphane tasarımındaki bir böcek olarak değerlendirilebilir. Eğer *copy_if* olsaydı aşağıdaki gibi faydalı bir işlemi yapabilirdik.

```
copy_if(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"), bind2nd(greater<int>(), 100));
```

copy_if fonksiyonu standart kütüphaneye ilk eklenecek fonksiyonlardan biri olarak görülmektedir. Zaten derleyicilerin bir bölümünde standart olmamasına karşın <algorithm> başlık dosyası içerisinde bu fonksiyon bulundurulmaktadır. *copy_if* fonksiyonu şöyle yazılabilir:

```
template <class IISource, class OIDest, class UnaryPredicate>
OIDest copy_if(IISource first, IISource last, OIDest dest, UnaryPredicate op)
{
    while (first != last) {
        if (op(*first))
            *dest++ = *first;
        ++first;
    }
    return dest;
}
```

```
using namespace std;
```

```
template <class IISource, class OIDest, class UnaryPredicate>
OIDest copy_if(IISource first, IISource last, OIDest dest, UnaryPredicate op)
{
    while (first != last) {
        if (op(*first))
            *dest++ = *first;
        ++first;
    }

    return dest;
}

int main(void)
{
    int a[] = {3, 3, 3, 3, 3, 4, 4, 4, 4, 4};
    vector<int> v(a, a + 10);

    copy_if(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"),
            bind2nd(greater<int>(), 3));

    return 0;
}
```

Üye Fonksiyon Göstericileri

Sınıfın üye fonksiyonunun başlangıç adresi bir üye fonksiyon göstericisine yerleştirilebilir. Üye fonksiyon göstericileri şöyle tanımlanır:

```
void (Sample::*p)();
int(sample::*f)(int, int);
```

Bu tanımlamalarda * atomunun yeri doğrudur. Eğer tanımlama aşağıdaki gibi yapılsaydı anlamsız olurdu.

```
void (*Sample::p)();
```

Burada p, Sample sınıfının faaliyet alanında aranır. Halbuki doğru tanımlamada anlatılmak istenen p' nin Sample sınıfının elemanı olmadığı, *p' nin yani p' nin gösterdiği fonksiyonun Sample sınıfının elemanı olduğudur.

Bilindiği gibi C' de bir fonksiyonun ismi zaten o fonksiyonun başlangıç adresini belirtmektedir. Halbuki C++' da fonksiyon isminin fonksiyonun bütününe belirttiği kabul edilmiştir. dolayısıyla C++' da Func bir fonksiyon olmak üzere bu fonksiyonun adresi Func biçiminde değil, &Func biçiminde alınmalıdır. Fakat C ile uyumu koruyabilmek için, C++' da global ve sınıfın static üye fonksiyondan fonksiyon adresine otomatik dönüşüm tanımlanmıştır. Yani Func global ya da sattaic üye fonksiyon olmak üzere C++' da bu fonksiyonun adres ifadesi için Func ya da &Func ifadeleri kullanılabilir. Fakat standartlara göre sınıfın static olmayan üye fonksiyonunun adresini elde edebilmek için kesinlikle & operatörü kullanılmalıdır. Func, Sample sınıfının satıc olmayan bir üye fonksiyonu olsun:

```
void (Sample::*pf)();
pf = &Sample::Func;
```

C++' da fonksiyon göstericisi olduğu gibi fonksiyon referansı kavramı da vardır. Fakat üye fonksiyon referansı kavramı yoktur. Üye fonksiyon göstericisi olabilir ama üye fonksiyon referansı olamaz, global fonksiyon referansı olabilir. Örneğin;

```
void (&r)() = Func;
r();
```

Görüldüğü gibi r bir fonksiyon referansı olmak üzere fonksiyonun çağırılma ifadesi r(); biçiminde yapılmaktadır.

Anahtar Notlar:

C++' da bir fonksiyonun ismi fonksiyonun adresi değil kendisi anlamına gelmektedir. Fonksiyonun kendisine atama yapılamamasına karşın bir sol taraf değeridir. O halde C++' da biz bir fonksiyon referansı tanımlarken fonksiyon isminin kendisini ilk değer olarak vermeliyiz. Fakat fonksiyon göstericisine atama yapılırken adres alma operatörünü kullanmalıyız. C ile uyumu korumak için global fonksiyonların adresleri için adres operatörünün kullanılması gerekliliği kaldırılmıştır. Örneğin;

```
void (&r)() = Func;
void (*p)() = &Func;
```

Üye fonksiyon türünden referans tanımlanamaz ama gösterici tanımlanabilir. Ayrıca bir üye fonksiyonun içerisinde başka bir üye fonksiyonun adresini almak için yine sınıf ismiyle kombine etmek zorunludur. Başka bir deyişle bir üye fonksiyonun adresi her zaman sınıf ismiyle kombine edilerek alınır.

```
class Sample{
public:
    void Func();
    void Sample()
{
```

```

    void (Sample::*p)();
    p = &Func; //geçersiz
    //...
}
};

```

↓
&Sample::Func

Şüphesiz sınıf bildirimi içerisinde üye fonksiyon göstericisi tanımlayabilmek için yine sınıf ismiyle kombine etmek gerekir.

```

class Sample{
private:
{
    void (*m_pf1)(); //global fonksiyon göstericisi
    void (Sample::*m_pf2)(); //üye fonksiyon göstericisi
    //...
}
};

```

Bir sınıfın static üye fonksiyonunun adresi üye fonksiyon göstericisinde değil, normal yani global fonksiyon göstericisine atanabilir. Örneğin, thread yaratan CreateThread gibi bir API fonksiyonu bizden bir fonksiyon adresi istesin. Biz bu fonksiyona global bir fonksiyonun ya da static bir üye fonksiyonun adresini geçebiliriz.

```

#include <windows.h>
#include <iostream>

```

```

class Thread {
public:
    Thread(int n);
    ~Thread();
    void Wait();
    static DWORD CALLBACK ThreadProc(LPVOID pParam);
private:
    HANDLE m_hThread;
    DWORD m_dwThreadId;
    int m_n;
};

```

```
using namespace std;
```

```

Thread::Thread(int n) : m_n(n)
{
    m_hThread = ::CreateThread(NULL, 0, &Thread::ThreadProc, this, 0,
    &m_dwThreadId);
}
void Thread::Wait()
{
    ::WaitForSingleObject(m_hThread, INFINITE);
}
DWORD Thread::ThreadProc(LPVOID pParam)
{
    Thread *pThread = reinterpret_cast<Thread *>(pParam);

```

```

        for (int i = 0; i < pThread->m_n; ++i) {
            cout << i << endl;
            ::Sleep(1000);
        }

        return 0;
    }
    Thread::~Thread()
    {
        ::ExitThread(0);
        ::CloseHandle(m_hThread);
    }
    int main(void)
    {
        Thread thread(10);
        thread.Wait();
        return 0;
    }
}

#include <windows.h>
#include <string>
#include <iostream>

class Thread {
public:
    Thread(int n, const char *pName);
    ~Thread();
    void Wait();
    static DWORD CALLBACK ThreadProc(LPVOID pParam);
private:
    HANDLE m_hThread;
    DWORD m_dwThreadId;
    std::string m_name;
    int m_n;
};

using namespace std;

Thread::Thread(int n, const char *pName) : m_n(n), m_name(pName)
{
    m_hThread = ::CreateThread(NULL, 0, &Thread::ThreadProc, this, 0,
    &m_dwThreadId);
}

void Thread::Wait()
{
    ::WaitForSingleObject(m_hThread, INFINITE);
}

```

```

DWORD Thread::ThreadProc(LPVOID pParam)
{
    Thread *pThread = reinterpret_cast<Thread *> (pParam);

    for (int i = 0; i < pThread->m_n; ++i) {
        cout << pThread->m_name << ":" << i << endl;
        ::Sleep(1000);
    }
    return 0;
}
Thread::~Thread()
{
    ::ExitThread(0);
    ::CloseHandle(m_hThread);
}
int main(void)
{
    Thread thread1(10, "first");
    Thread thread2(5, "second");
    thread1.Wait();
    thread2.Wait();

    return 0;
}

```

Üye Fonksiyon Göstericisi ile Üye Fonksiyonların Çağırılması

pf bir üye fonksiyon göstericisi olmak üzere pf göstericisinin içindeki adreste bulunan fonksiyon ancak bir sınıf nesnesi ile çağırılabilir. Çünkü çağırılacak fonksiyon bir this göstericisine gereksinim duymaktadır. Bir üye fonksiyon göstericisi ile üye fonksiyonu çağırabilmek için `".*"` biçiminde bir operatör kullanılır. Bu operatörle çağırma işlemi şöyle yapılmalıdır.

```
(nesne .* fonksiyon_göstericisi)(...);
```

`.*` operatörünün sol tarafındaki operand bir sınıf nesnesi sağ tarafındaki operand ise o sınıfın taban sınıfına ilişkin bir üye fonksiyon adresi olmalıdır. `.*` operatörünün önceliği düşük olduğu için paranteze alınmıştır. Örneğin;

```

void(A::*pf)();
pf = &A::Func();
A a;
(a.*pf)();

```

Ayrıca p bir sınıf türünden gösterici olmak üzere bu sınıfa ilişkin üye fonksiyonu fonksiyon göstericisiyle çağırabilmek için `"->*"` operatörü kullanılmaktadır.

```
(nesne_adresi -> *fonksiyon_göstericisi)(...);
```

Görüldüğü gibi `.*` operatörünün solunda nesnenin kendisi olduğu halde `->*` operatörünün solunda nesnenin adresi vardır.

```

void(A::*pf)();
pf = &A::Func();
A *pA = new A();

```

```

(a->*pf)();
using namespace std;

class A {

public:
    void Func()
    {
        cout << "test\n";
    }
};

int main(void)
{
    void (A::*pfA)() = &A::Func;
    A a;
    /*
        A *pA = new A();

        (pA ->*pfA)();
    */
    (a.*pfA)();

    return 0;
}

```

C++' da Fonksiyon Adresleri Arasında Yapılan Dönüştürmeler

C' de ve C++' da doğrudan ya da dolaylı bir biçimde yani hiçbir şekilde fonksiyon adresleri ile data adresleri arasında dönüştürme yapılamaz. void* türü data adresi türüdür bir fonksiyon adresi geçici olarak saklanacaksa bir fonksiyon göstericisinde saklanmalıdır. Yine C++' da üye fonksiyon adresleriyle global fonksiyon adresleri arasında doğrudan ya da tür dönüştürme operatörüyle bir dönüştürme tanımlanmamıştır.

C++' da üye fonksiyon adresleriyle ilgili iki dönüştürme tanımlanmıştır.

1. Doğrudan Dönüştürme:

C++' da yalnızca taban sınıf türünden bir üye fonksiyonun adresi türemiş sınıf türünden bir üye fonksiyon göstericisine doğrudan atanabilir, tabi parametrik yapılarının aynı olması gerekir. Bunun tersinin normal olacağı sanılsa da tersi anormal bu biçimi normal ve güvenlidir. Çünkü taban sınıf türünden üye fonksiyon adresini türemiş sınıf türünden bir fonksiyon göstericisine atarsak bu fonksiyon göstericisi ile türemiş sınıf nesnesi kullanılarak çağırma yapılabilir. Yani biz türemiş sınıf nesnesi yoluyla taban sınıf üye fonksiyonunu çağırılmış oluruz ki bu normaldir. Bunun tersi olsaydı o zaman taban sınıf nesnesi ile türemiş sınıf üye fonksiyonunu çağırır duruma gelirdik.

2. reinterpret_cast ya da C Tarzı Tür Dönüştürme Operatörü İle

Herhangi bir sınıfın herhangi bir parametrik yapıya sahip üye fonksiyonunun adresi yine herhangi bir sınıfın herhangi bir parametre yapısına sahip üye fonksiyon adresine dönüştürülebilir.

```
void (A::*pfA)();
```

```
int (B::*pfB)(int, int);
```

```
...
```

```
pfA = reinterpret_cast <void(A::*)()> (pfB);
```

Bir olay gerçekleştiğinde bir sınıfın bir üye fonksiyonunun çağırılmasını isteyelim. Bu tür işlemler özellikle windows gibi mesaj tabanlı sistemlerdeki kütüphaneler tarafından kullanılmaktadır. Bunun için bizim üye fonksiyon adresini de üye fonksiyonun çağırılmasında kullanılacak nesnenin adresini de bir biçimde saklamamız gerekir.

Adres İşlemi Yapan Bağlayıcı Sınıflar

Bu konu standart kütüphanenin en karmaşık konularındandır. Elimizde bir dizilim (container) olduğunu düşünelim. Bu dizilim bir sınıf türünden nesneleri içeriyor olsun. Örneğin,

```
vector<A> v;
```

Tek bir fonksiyon ile biz bu dizilimin her elemanı ile sınıfın bir üye fonksiyonunu çağırarak isteyelim. İşte standart kütüphanede adres işlemi yapan bağlayıcı sınıflar bu yüzden kullanılmaktadır. Burada bağlayıcı sınıf tek parametrelili bir fonksiyon görevindedir. Yani *unary_function* sınıfından türetilmiştir. Sınıf bir üye fonksiyonun adresini saklar, sınıfın tek parametrelili bir fonksiyon çağırma operatör fonksiyonu vardır. Bu fonksiyonun parametresi üye fonksiyonun ilişkin olduğu sınıf türünden bir referanstır. Bu tek parametrelili fonksiyon çağırma operatör fonksiyonu parametresiyle verilen nesne ile ilgili üye fonksiyonu çağırır.

```
vector<A> v;
```

```
...
```

```
for_each(v.begin(), v.end(), mem_fun_ref_t<A, void>(&A::Func));
```

Burada *mem_fun_ref_t* bağlayıcı sınıfın ismidir.

1. Bağlayıcı sınıf bir template sınıftır.
2. Template sınıfın template parametresi üye fonksiyon adresini belirleyen parametrelerdir.
3. Bu template sınıf bir üye fonksiyon göstericisi veri elemanına sahiptir.
4. Bağlayıcı sınıfın başlangıç fonksiyonu üye fonksiyonun adresini alarak sınıfın veri elemanında saklar.
5. Sınıfın tek parametrelili fonksiyon çağırma operatör fonksiyonu ilgili sınıf türünden bir referans parametresi almaktadır ve bu fonksiyon bu sınıf nesnesi ile üye fonksiyonu çağırır.
6. *mem_fun_ref_t* bağlayıcı sınıfının çağıracağı üye fonksiyonu parametresiz olmak zorundadır.

mem_fun_ref_t sınıfının bildirimi şöyle olmalıdır:

```
template <class S, class T> class mem_fun_ref_t
```

```
: public unary_function<T, S> {
```

```
public:
```

```
explicit mem_fun_ref_t(S (T::*p)());
```

```
S operator()(T& p) const;
```

```
};
```


Görüldüğü gibi fonksiyon çağırma operator fonksiyonunun geri dönüş değeri çağırılan üye fonksiyonun geri dönüş değeridir. *mem_fun_ref_t* türünden nesnenin oluşturulmasını kolaylaştırmak için *mem_fun* isimli bir bağlayıcı fonksiyon düşünülmüştür.

```
template<class S, class T> mem_fun_ref_t<S,T>
mem_fun(S (T::*f)());
```

Görüldüğü gibi fonksiyonun geri dönüş değeri *mem_fun_ref_t* türünden bir nesnedir. Bu durumda

```
for_each(v.begin(), v.end(), mem_fun_ref_t<A, void>(&A::Func));
```

ile

```
for_each(v.begin(), v.end(), mem_fun_ref (&A::Func));
```

eşdeğerdir.

SINIF ÇALIŞMASI:

Stringlerden oluşan bir vektör alınız, tek hamlede *remove_if* fonksiyonu ile karakter sayısı 5'ten fazla olan stringleri siliniz.

```
using namespace std;
```

```
template <class IISource, class OI Dest, class Predicate>
OI Dest copy_if(IISource first, IISource last, OI Dest dest, Predicate op)
{
    while (first != last) {
        if (op(*first))
            *dest++ = *first;
        ++first;
    }
    return dest;
}
```

```
int main(void)
{
    vector<string> v;
    v.push_back("kaan");
    v.push_back("ali");
    v.push_back("veli");
    v.push_back("selami");
    /*copy_if(v.begin(), v.end(), ostream_iterator<string>(cout, "\n"),
        mem_fun_ref(&string::size));
    */
    copy(v.begin(), v.end(), ostream_iterator<string>(cout, "\n"));
    return 0;
}
```

Örneğin biz *copy* fonksiyonu ile string türünden bir vektörün içeriğini basit bir biçimde yazdırabiliriz.

```
using namespace std;
```

```
template <class IISource, class OI Dest, class Predicate>
```

```

OIDest copy_if(IISource first, IISource last, OIDest dest, Predicate op)
{
    while (first != last) {
        if (op(*first))
            *dest++ = *first;
        ++first;
    }
    return dest;
}

```

```

int main(void)
{
    vector<string> v;

    v.push_back("kaan");
    v.push_back("ali");
    v.push_back("veli");
    v.push_back("selami");

    transform(v.begin(), v.end(), ostream_iterator<const char *>(cout, "\n"),
               mem_fun_ref(&string::c_str));
    /*copy(v.begin(), v.end(), ostream_iterator<string>(cout, "\n"));
    */
    return 0;
}

```

Yada örneğin string türünden bir dizilimin tüm elemanlarının karakter uzunluğunu bir hamlede aşağıdaki gibi yazdırılabilir.

```

int main(void)
{
    vector<string> v;

    v.push_back("kaan");
    v.push_back("ali");
    v.push_back("veli");
    v.push_back("selami");

    transform(v.begin(), v.end(), ostream_iterator<const char *>(cout, "\n"),
               mem_fun_ref(&string::size));

    return 0;
}

```

mem_fun_ref_t sınıfı ve *mem_fun_ref* fonksiyonu bir dizilimdeki sınıf nesneleriyle sınıfın parametresiz üye fonksiyonlarını çağırarak kullanılır. Fakat buna seçenek olarak sınıfın bir parametrelili fonksiyonunu çağırabilecek *mem_fun1_ref_t* sınıfı da vardır. bu sınıf için ayrı bir bağlayıcı fonksiyon ismi türetilmemiştir. Farklı parametre yapılarına ilişkin aynı isimli fonksiyonlar olabildiği için bu sınıfın bağlayıcı fonksiyonu yine *mem_fun_ref* fonksiyonudur.

```

template <class S, class T, class A> class mem_fun1_ref_t
: public binary_function<T, A, S> {
public:

```

```
explicit mem_fun_ref_t(S (T::*p)(A));
S operator()(T& p, A x) const;
};
```

```
template<class S, class T, class A> mem_fun_ref_t<S,T,A>
mem_fun_ref(S (T::*f)(A));
```

Görüldüğü gibi *mem_fun_ref_t* sınıfı iki parametrelili fonksiyon görevi yapmaktadır. Fonksiyonun birinci parametresi dizilimin elemanı, ikinci parametresi ise dışarıdan tespit edilecek değerdir. Bu nedenle bu sınıfı *binder* fonksiyonlarıyla kullanmak daha uygundur. Örneğin;

```
int main(void)
{
    vector<Number> v;

    for (int i = 0; i < 100; ++i)
        v.push_back(i);
    transform(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"),
        bind2nd(mem_fun_ref(&Number::Func), 5));

    return 0;
}
```

mem_fun_ref ön ekli sınıf ve fonksiyonların *mem_fun* li biçimleri de vardır. *mem_fun_ref_t* gibi *mem_fun_t* sınıfı *mem_fun_ref* fonksiyonu gibi *mem_fun* fonksiyonu da vardır. Bu fonksiyonlar ve sınıflar arasındaki ilişki basittir. *mem_fun_ref_t* dizilimde sınıf nesneleri varsa ve bu nesnelerin her biriyle bir üye fonksiyon çağırmak için kullanılır. Halbuki *mem_fun_t* sınıfı dizilimde nesnelerin adresleri varsa dizilimdeki her nesne adresi ile belirtilen üye fonksiyonu çağırılmaktadır. Örneğin, bir vektörün string nesnelerinin başlangıç adreslerini sakladığını düşünelim bu durumda vektörün her elemanı *string** türündendir. Bu elemanlarla *string* sınıfının üye fonksiyonunu çağırmak için *mem_fun_ref* fonksiyonunu değil *mem_fun* fonksiyonunu kullanmalıyız.

```
int main(void)
{
    vector<string*> v;
    v.push_back(new string("kaan"));
    v.push_back(new string("ali"));
    v.push_back(new string("veli"));
    v.push_back(new string("selami"));

    transform(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"),
        mem_fun(&string::size));
    return 0;
}
```

mem_fun_t sınıfının *mem_fun1_t* vardır.

not1 ve not2 Bağlayıcı Fonksiyonları

not1 fonksiyonu *unary_negate* sınıfı türünden bir nesne ile geri döner. *unary_negate* sınıfı tek parametrelili bir fonksiyon sınıfıdır.

```
template <class Predicate>
class unary_negate
: public unary_function<typename Predicate::argument_type, bool> {
public:
explicit unary_negate(const Predicate& pred);
bool operator()(const typename Predicate::argument_type& x) const;
};
```

Görüldüğü gibi bu sınıf tek parametrelili bir predicate fonksiyonu almakta ve bu fonksiyonu çağırarak bunun tersiyle geri dönmektedir. Yani örneğin,

```
bool Proc(int);
unary_negate<bool (*) (int)>
a(Proc);
```

Burada `a(10)` ile `!Proc(10)` aynı anlamdadır. `unary_negate` fonksiyonu özellikle tek parametrelili bir predicate fonksiyonun ters davranmasını sağlamak için kullanılır. Bu sınıfa bağlama yapan `not1` fonksiyonu şöyledir:

```
template <class Predicate>
unary_negate<Predicate> not1(const Predicate& pred);
```

Kısa bir anlatımla `pred` tek parametrelili bir predicate fonksiyon olmak üzere, `!pred(x)` işlemi ile `not1(pred)(x)` aynı anlamdadır. `not1` fonksiyonu uygulamada sık kullanılmaktadır.

`binary_negate` fonksiyon sınıfı iki parametrelili bir predicate fonksiyonun tersini almak için kullanılmaktadır.

```
template <class Predicate>
class binary_negate
: public binary_function<typename Predicate::first_argument_type,
typename Predicate::second_argument_type, bool> {
public:
explicit binary_negate(const Predicate& pred);
bool operator()(const typename Predicate::first_argument_type& x,
const typename Predicate::second_argument_type& y) const;
};
```

Bu sınıfta `not2` isimli bağlayıcı bir fonksiyonu vardır.

```
template <class Predicate>
binary_negate<Predicate> not2(const Predicate& pred);
```

Yani `Pred` iki parametrelili predicate fonksiyon olmak üzere, `!Pred(x,y)` ile `not2(Pred)(x,y)` aynı anlamdadır.

`not1` ve `not2` fonksiyonlarının tipik kullanımına ilişkin örnekler verilebilir. Örneğin;

```
remove_if(v.begin(), v.end(), not1(bind2nd(greater<int>(), 5)));
```

Burada 5' ten büyük olmayanlar silinmiştir. Şüphesiz buradaki işlem `not1` kullanmak yerine `less_equal` ile yapılabilir. Şimdi `not1(bind2nd(greater<int>(), 5))` işlemini çözümlemeye çalışalım.

1. `greater<int>` fonksiyonunu `bind2nd` çağıracaktır. `bind2nd` tek parametrelili bir fonksiyon gibi davranan çift parametrelili bir fonksiyondur.

2. `greater<int>` fonksiyonunun geri dönüş değeri `bind2nd` fonksiyonunun geri dönüş değeri olarak elde edilmektedir, `not1` bunun tersini almaktadır.

copy_if fonksiyonunun olmaması bir böcek olarak değerlendirilebilir. *copy_if* fonksiyonunun işlevi performans düşümü olmadan mevcut fonksiyonlarla sağlanamamaktadır. Fakat *copy_if* fonksiyonunun işlevini yerine getiren bir kaç kalıp kullanılabilir. bu kalıplardan biri *copy_if* yerine *remove_copy_if* fonksiyonunu kullanmaktır. *remove_copy_if* fonksiyonu iyi isimlendirilmemiştir. Bir dizilimde belli bir koşulu sağlamayanları başka bir dizilime kopyalar. Yani adeta *copy_if* fonksiyonunun tersini yapmaktadır.

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
    OutputIterator result, Predicate pred);
```

O halde örneğin, *copy_if(first, last, dest, pred);* ile *remove_copy_if(first, last, dest, not1(pred));* aynı anlamdadır. Yani örneğin,

```
copy_if(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"),
    bind2nd(greater<int>(), 5));
```

işleminin eş değeri,

```
remove_copy_if(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"),
    not1(bind2nd(greater<int>(), 5)));
```

Bu ikinci ifade de *less_equal*'da kullanılabilir.

auto_ptr Sınıfı

auto_ptr sınıfı bir akıllı gösterici (smart pointer) sınıfıdır. Bu sınıf memory başlık dosyası içerisinde. Akıllı gösterici sınıfı gösterici gibi davranan sınıf nesnesidir, iterator kavramıyla bir ilgisi yoktur. iterator kavramı veri yapısını dolaşmak için kullanılan bir kavramdır. Akıllı gösterici sınıfında iterator kavramında olduğu gibi arttırmak ve eksilmek teması yoktur. X bir akıllı gösterici sınıfı olsun, bu sınıfı Y türünden bir göstericiyi temsil ediyor olsun. X sınıfı türünden bir nesnenin gösterici gibi davranabilmesi için * ve -> operator fonksiyonlarının tanımlanmış olması gerekir.

Göstericiyi doğrudan kullanmak yerine bir sınıfa bağlayıp akıllı gösterici biçimine dönüştürerek kullanmanın faydası nedir? İşte bazı durumlarda göstericinin bir sınıf nesnesi biçiminde olması avantaj sağlamaktadır. Çünkü akıllı gösterici sınıfının bitiş fonksiyonu delete işlemi yaparak tahsis edilmiş olan alanı silebilir. Bir exception oluştuğunda o zamana kadar yaratılan yerel sınıf nesneleri için bitiş fonksiyonu çağırılır ama dinamik tahsis edilmiş olan sınıf nesneleri için çağırılmaz, bir boşaltma yapılmaz. Eğer biz göstericiyi sınıf nesnesi ile temsil edersek otomatik bir geri bırakma gerçekleşebilir. Benzer biçimde MIL sentaksı ile ilk değer verme sırasında bir exception oluştuğunda da aynı türden problem oluşabilir. Özetle akıllı gösterici sınıfı nesnenin faaliyet alanı bittiğinde otomatik boşaltımı sağlayabilmektedir. Ayrıca C++'da akıllı göstericiler referans sayacı oluşturma (reference counting) konusunda da faydalı olmaktadır.

auto_ptr sınıfı template bir sınıftır, template parametresi nesnenin hangi türden gösterici gibi davranacağını belirtir. Örneğin:

```
auto_ptr<X> a(new X()); //a, X sınıfı türünden gösterici gibi davranır.
```

auto_ptr sınıfı atama işlemi sırasında içerik kaopyalaması yapan bir sınıf değildir. Bunun yerine sahipliği atanan tarafa devreden bir yapıya sahiptir. Örneğin;

```
{
    auto_ptr<X> a(new X());
    auto_ptr<X> b;
    b = a;
```

```
...
}
```

Burada blok sonunda hem a hem de b için bitiş fonksiyonu çağırılırsa blok iki kez boşaltılma durumuna gelir. Bu da gösterici hatasına yol açar. Sahipliği devretme o alan için boşaltma yapılmayacağı anlamına gelir. Bu atama işlemi ile a sahipliği b' ye devreder yani yaratılmış olan dinamik alana artık a' nın bitiş fonksiyonu değil, b' nin bitiş fonksiyonu silecektir. Sahipliği devretme sırasında atamanın sağ tarafındaki nesne üzerinde değişiklik yapılması gerekmektedir. Bu durum klasik atama kavramıyla bağdaşmamakla birlikte faydalı bir durum oluşturur. *auto_ptr* sınıfının anhtar özellikleri şunlardır:

- Atama operator fonksiyonu va kopya başlangıç fonksiyonu atanılan nesnenin sahipliğini bırakır.
- Tipik olarak sınıfın private iki veri elemanı vardır. Bir tanesi template parametresi türünden gösterici, diğeri ise sahipliğin nesnede olup olmadığını belirten bool türde bir elemandır.
- Sınıfın tek parametrelili fakat default NULL değeri alan bir başlangıç fonksiyonu vardır. Programcı, new operatoruyla tahsisat yaparak akıllı göstericiyi oluşturur. Yani alanın tahsis edilmesi programcı tarafından, boşaltılması sınıf tarafından yapılmaktadır.

```
template<class X> class auto_ptr {
...
};
```

Önemli üye fonksiyonları şunlardır:

```
X& operator*() const throw();
X* operator->() const throw();
X* get() const throw();
X* release() throw();
void reset(X* p =0) throw();
```

get fonksiyonu saklanan adresi geri almak için, *release* sahipliği bırakmak için ve *reset* ise tamamen başlangıç durumuna dönmek için kullanılır.

```
using namespace std;
```

```
class Sample {
public:
    Sample(int a) : m_a(a)
    {}
    ~Sample()
    {
        cout << "destructor\n";
    }
    void Disp() const
    {
        cout << m_a << endl;
    }
private:
    int m_a;
};
```

```
int main(void)
```

```

{
    {
        auto_ptr<Sample> p(new Sample(10));
        auto_ptr<Sample> a;

        a = p;
        a->Disp();
    }

    return 0;
}

```

Özellikle sınıfların veri elemanlarında *auto_ptr* nesneleri kullanılmaktadır. Örneğin;

```

class Sample {
public:
    Sample(int a) : m_p(new int), m_something(params)
    {
        ...
        if (...)
            throw Exception(...);
    }
    // ...
private:
    auto_ptr<int> m_p;
    Something m_something;
};

```

Burada *Something* sınıfının başlangıç fonksiyonunda ya da *Sample* sınıfının başlangıç fonksiyonunda throw işlemi olursa, tahsis edilen alan otomatik olarak boşaltılır.

Allocator Kavramı

Standart kütüphanede nesne tutan sınıflar kendi ilerinde pek çok durumda dinamik tahsisat yapmaktadır. Örneğin, *List* sınıfı düğümleri tahsis ederken heap üzerinde dinamik olarak tahsis etmektedir. Sistemlerin heap alanları değişik biçimlerde organize edilmiş olabilir. Örneğin, Win32 sistemlerinde birden fazla heap alanı yaratılabilmektedir. Peki bu nesne tutan sınıflar tahsisatı nasıl yaparlar? İşte *Allocator* nesne tutan sınıfların kullandığı tahsisat nesnesidir. Yani nesne tutan sınıflar programcıdan bir *allocator* nesnesi isterler ve o *allocator* nesnesini kullanarak dinamik tahsisatları yaparlar. Örneğin, Unix/Linux sistemlerinde paylaşılan bir bellek alanı oluşturduğumuzu düşünelim. Bağlı listenin bu alan üzerinde tahsisatlarını yapmasını isteyelim. Bu durumda biz bir *allocator* sınıfı yazmalıyız ve tahsisat için o sınıf türünden bir *allocator* nesnesini vermeliyiz. Yazacağımız *allocator* sınıfı tahsisatları bu paylaşılan bellek alanından yapmalıdır. Tabi normal olarak bu tür gereksinimleri fazlaca rastlanmaktadır. Bu nedenle kütüphanede default bir *allocator* sınıfı bulundurulmaktadır. Bu default *allocator* sınıfı tahsisatları *new* operatörünü kullanarak yapar.

Bir *allocator* sınıfının hangi özelliklere sahip olması gerektiği standartlarda tablo 32 de açıklanmıştır. Yani programcı *allocator* işlemi yapan bir sınıf yazacaksa bu tabloda belirtilen elemanları içermelidir. Kütüphanedeki standart *allocator* sınıfı bütün bu özellikleri desteklediğine göre onun incelenmesi yoluyla bu özelliklerde anlaşılabilir.

expression	return type	assertion/note pre/post-condition
<code>X::pointer</code>	Pointer to T.	
<code>X::const_pointer</code>	Pointer to const T.	
<code>X::reference</code>	T&	
<code>X::const_reference</code>	T const&	
<code>X::value_type</code>	Identical to T	
<code>X::size_type</code>	unsigned integral type	a type that can represent the size of the largest object in the allocation model.
<code>X::difference_type</code>	signed integral type	a type that can represent the difference between any two pointers in the allocation model.
typename <code>X::template rebind<U>::other</code>	Y	For all U (including T), <code>Y::template rebind<T>::other</code> is X.
<code>a.address(r)</code>	<code>X::pointer</code>	
<code>a.address(s)</code>	<code>X::const_pointer</code>	
<code>a.allocate(n)</code> <code>a.allocate(n,u)</code>	<code>X::pointer</code>	Memory is allocated for n objects of type T but objects are not constructed. <code>allocate</code> may raise an appropriate exception. The result is a random access iterator. ²¹⁴⁾ [Note: If <code>n == 0</code> , the return value is unspecified.]
<code>a.deallocate(p, n)</code>	(not used)	All n T objects in the area pointed by p shall be destroyed prior to this call. n shall match the value passed to <code>allocate</code> to obtain this memory. Does not throw exceptions. [Note: p shall not be null.]
<code>a.max_size()</code>	<code>X::size_type</code>	the largest value that can meaningfully be passed to <code>X::allocate()</code> .
<code>a1 == a2</code>	bool	returns true iff storage allocated from each can be deallocated via the other.
<code>a1 != a2</code>	bool	same as <code>!(a1 == a2)</code>
<code>X()</code>		creates a default instance. Note: a destructor is assumed.
<code>X a(b);</code>		post: <code>Y(a) == b</code>
<code>a.construct(p,t)</code>	(not used)	Effect: <code>new((void*)p) T(t)</code>
<code>a.destroy(p)</code>	(not used)	Effect: <code>((T*)p) -> ~T()</code>

Bir *allocator* sınıfının kabaca *allocate* ve *deallocate* biçiminde adeta *malloc* ve *free* fonksiyonları gibi fonksiyonları vardır. Bir *allocator* nesnesi belirli bir tür için tahsisat yapan nesnedir. Dolayısıyla *allocate* fonksiyonu da byte cinsinden değil, eleman sayısı olarak tahsisatını yapar. Örneğin, *allocator* nesnemiz *int* türden tahsisat yapmak için tasarlanmış olsun. biz de *allocate* fonksiyonunu 5 parametresiyle çağırmış olalım. Bu durumda fonksiyon 5 byte değil, 5 *int*'lik bir alan tahsis eder.

Default allocate Sınıfı

Nesne tutan sınıflar kullanılacak *allocator* sınıfını template parametresi olarak alırlar ve o türden private bir *allocator* nesnesi tutarak tahsisatları onunla yaparlar. Nesne tutan sınıfların bu *allocator* template parametresi default değer almaktadır. Eğer, biz bu parametreyi belirtmezsek default *allocator* sınıfı temöplate parametresi olarak kullanılır. Örneğin, *deque* sınıfının template bildirimi şöyledir:

```
template <class T, class Allocator = allocator<T> >
class deque {
    //...
private:
    Allocator a;
    //...
};
```

Görüldüğü gibi burada sınıfın birinci template parametresi belirtilmek zorundadır. Fakat ikinci parametre belirtilmezse default *allocator* sınıfı kullanılacaktır. Örneğin, biz *deque<int> x;* biçiminde bir nesne tanımlarsak, aslında *deque<int, allocator<int> > x;* biçiminde nesne tanımlamakla aynı işi yapmış oluruz. Ayrıca nesne tutan sınıfların *allocator* parametrelili bir başlangıç fonksiyonları da vardır. Burada amaç istenilen türün yanısıra daha önce istenildiği gibi yaratılmış olan bir *allocator* nesnesini sınıfın kullanmasını sağlamaktır. Örneğin, *deque* sınıfının bir başlangıç fonksiyonu şöyledir.

```
explicit deque(const Allocator& = Allocator());
```

allocator sınıfının en önemli fonksiyonu *allocate* fonksiyonudur.

```
pointer allocate(size_type n, allocator<void>::const_pointer hint=0);
```

Fonksiyonun birinci parametresi ilgili türden elemandan kaç parça tahsis edileceğini belirtir, ikinci parametre aslında kullanılmamaktadır. Fakat bazı algoritmaların dışarıdan ekstra bir bilgiye gereksinim duyacağı düşünüldüğünden dahil edilmiştir. Fonksiyonun geri dönüş değeri tahsis edilen alanın başlangıç adresidir. Default *allocator* sınıfı bu fonksiyonda global *operator new* fonksiyonunu kullanmaktadır. Fakat bu fonksiyon eğer tahsisat türü bir sınıf türündense başlangıç fonksiyonunu çağırılmaz. Başlangıç fonksiyonu ayrıca sınıfın *construct* isimli üye fonksiyonuyla çağırılmaktadır.

Tahsis edilen alan sınıfın *deallocate* fonksiyonuyla serbest bırakılır.

```
void deallocate(pointer p, size_type n);
```

Fonksiyonun birinci parametresi boşaltılacak alanın başlangıç adresi, ikinci parametresi ise kaç eleman boşaltılacağıdır. Görüldüğü gibi fonksiyon klasik *free* fonksiyonu gibi değildir. Yani tüm alanı değil, alanın bir bölümünü de boşaltabilmektedir.

Sınıfın *construct* fonksiyonu *placement new* opratorunu kullanarak *allocate* fonksiyonuyla tahsis edilen alan için başlangıç fonksiyonunu çağırır.

```
void construct(pointer p, const_reference val);
```

Fonksiyonun birinci parametresi elemanın adresi, ikinci parametresi başlangıç fonksiyonuna parametre olarak kullanılacak değerdir. *destroy* fonksiyonu da bitiş fonksiyonunu çağırarakta kullanılır.

```
void destroy(pointer p);
```

Fonksiyonun parametresi bitiş fonksiyonu çağırılacak elemanın adresidir.

Nesne Tutan Sınıfların Allocator Nesnelerini Kullanması

Bilindiği gibi bir allocator nesnesi yalnızca belli türden tahsisatları yapabilir. Peki nesne tutan sınıflar acaba tüm tahsisatlarında bu nesneyi kullanabilirler mi? Hayır. Örneğin bir bağlı liste nesnesi yaratalım.

```
list<int> a;
```

bu tanımlama aşağıdakiyle eşdeğerdir.

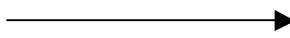
```
list<int, allocator<int> > a;
```

Görüldüğü gibi bu nesne yalnızca *int* türünden tahsisatlarda bu allocator nesnesini doğrudan kullanabilir. Halbuki bağlı listenin elemanları *node* gibi bir yapı türündendir, *int* bunun içerisinde. İşte nesne tutan sınıflar ne zaman *template* türünün dışındaki bir türden tahsisat yapacak olsalar *allocator* sınıfı içerisindeki *template rebind* sınıfının *other* elemanını kullanırlar. *Template rebind* sınıfının *other* elemanı *allocator* belirten bir tür ismidir. Default *allocator* sınıfının *rebind* sınıfı ve *other* tür ismi aşağıdaki gibi oluşturulmuştur.

```
template <class T>
class allocator{
    //...
    template<class U>
    struct rebind{
        typedef allocator<U> other;
    };
    //...
};
```

Görüldüğü gibi *other*, *allocator* sınıfı yine *allocator* sınıfına bağlanmıştır. Yani *list<int, allocator<int> > a;* gibi bir nesnede *Node* türünden tahsisatlar *allocator<Node>* türünden bir *allocator* nesnesiyle yapılacaktır. Şimdi *list* *template* sınıfı içerisinde bu tahsisatın nasıl yapılmış olabileceğini düşünelim:

```
template <class T, Allocator = allocator<T> >
class list{
    //...
};
```



```
Allocator::rebind<Node>::other nodeAlloc;
Node<T> *pNode;
pNode = nodeAlloc.allocate(1);
//...
```

Ayrı Bir Allocator Sınıfının Yazılması ve Kullanılması

Allocator konusu tahsisatları programcının isteği doğrultusunda nesne tutan sınıflara yaptırmak amacıyla düşünülmüştür. Yani *allocator* kavramından amaç programcının kendi *allocator* nesnelerini yazmasına olanak sağlamaktır. Tabi böyle bir gereksinimle çok seyrek karşılaşilmektedir.

Yazılacak *allocator* sınıfının standartlarda belirtilen "*Allocator Requirements*" belirlemelerini karşılaması gerekir. Yani yazılacak sınıfın bazı *typedef* isimleri ve üye fonksiyonları olmak zorundadır. Fakat bu sıkıcı bir işlemdir, bunun yerine kütüphanedeki *allocator* sınıfından türetme yapılabilir. Böylece bazı elemanları yazmaya gerek olmayabilir. Yazılacak *allocator* sınıfının *template* olması zorunlu değildir. Fakat bu sınıfın içerisinde *template rebind* yapısının bulunması gerekir. Programcının bu *rebind* sınıfının *other* türünü kendisine bağlaması gerekir, yoksa başka türden tahsisatlar için *allocator* sınıfı devreye girer. Tabi programcının kendi *allocator* sınıfını da *template* olarak yazması daha esneklik sağlar.

Örnek bir *template allocator* sınıfı aşağıdaki gibi oluşturulabilir.

```
template<T>
class MyIntAllocator<T>: public allocator<T>{
public:
    pointer allocate(size_type n, allocator<void>::const_pointer hint=0)
    {
        //...
    }
    void deallocate(pointer p, size_type n)
    {
        //...
    }
    template<class U>
    struct rebind{
        typedef MyIntAllocator<U> other;
    };
};
```

```
using namespace std;
```

```
template<class T>
class CustomAllocator : public allocator<T> {
public:
    pointer allocate(size_type n, const allocator<void>::const_pointer hint = 0)
    {
        cout << "allocate called: " << n << endl;
        return allocator<T>::allocate(n, hint);
    }
    void deallocate(pointer p, size_type n)
    {
        cout << "deallocate called: " << n << endl;
        return allocator<T>::deallocate(p, n);
    }
    template<class U>
    struct rebind {
        typedef CustomAllocator<U> other;
    };
};

int main(void)
{
    vector<int, CustomAllocator<int> > v;
```

```

        v.push_back(100);
        return 0;
}

```

Örnek;

```
using namespace std;
```

```

template <class T>
class CustomAllocator : public allocator<T> {
public:
    CustomAllocator()
    {}
    CustomAllocator(HANDLE hHeap);
    pointer allocate(size_type n, const allocator<void>::const_pointer hint = 0);
    void deallocate(pointer p, size_type n);
    template <class U>
    struct rebind {
        typedef CustomAllocator<U> other;
    };
private:
    HANDLE m_hHeap;
};

template <class T>
CustomAllocator<T>::CustomAllocator(HANDLE hHeap) : m_hHeap(hHeap)
{}

template <class T>
CustomAllocator<T>::pointer CustomAllocator<T>::allocate(size_type n, const
allocator<void>::const_pointer hint = 0)
{
    return (T *) ::HeapAlloc(m_hHeap, 0, n * sizeof(value_type));
}

template <class T>
void CustomAllocator<T>::deallocate(pointer p, size_type n)
{
    ::HeapFree(m_hHeap, 0, p);
}

int main(void)
{
    HANDLE hHeap;
    if ((hHeap = HeapCreate(0, 1000000, 1000000)) == NULL) {
        fprintf(stderr, "Cannot create heap!..\n");
        exit(EXIT_FAILURE);
    }
    CustomAllocator<int> customAllocator(hHeap);
    vector<int, CustomAllocator<int> > x(customAllocator);
    for (int i = 0; i < 10; ++i)
        x.push_back(i);
}

```

complex Sayı Sınıfı

complex sınıfı <complex> başlık dosyasında bulunan template bir sınıftır. Sayının gerçek ve sanal kısımlarının hangi türden olacağı template parametresiyle belirtilmektedir.

```
template<class T>
class complex{
    //..
private:
    T m_real;
    T m_imag;
};
```

Sınıfın başlangıç fonksiyonu şöyledir:

```
template<class T>
complex(const T& re = T(), const T& im = T());
```

Görüldüğü gibi bu başlangıç fonksiyonu default başlangıç fonksiyonu olarak da kullanılabilir. Bu durumda gerçek ve sanal kısımlara 0 atanacaktır.

Sayının gerçek ve sanal kısımlarını tutan veri elemanları sınıfın private bölümündedir. Bunları elde etmek için aşağıdaki gibi iki get fonksiyonu vardır.

```
T real() const;
T imag() const;
```

Aynı türden iki karmaşık sayıyı toplayan, çıkartan, çarpan ve bölen operator fonksiyonları vardır. Ayrıca bir karmaşık sayıyı yazdıran ve okuyan ostream ve istream operator fonksiyonları yazılmıştır. Örnek;

```
using namespace std;
```

```
int main(void)
{
    complex<int> a(3, 2);
    complex<int> b(3, 2);
    complex<int> c;

    c = a + b;

    cout << c << endl;

    return 0;
}
```

Ayrıca sınıfın bir karmaşık sayıyla bir gerçek sayıyı toplayan, çıkartan, çarpan, bölen fonksiyonları da vardır. Sınıfın diğer ayrıntıları standartlardan izlenebilir.

C++'ın Standart Exception Sınıfları

Bilindiği gibi programlardaki hatalar karşısında doğal türler yerine bir sınıf ile throw işlemi yapmak daha iyi bir yöntemdir. C++'daki yazılmış pek çok kütüphanede exception işlemlerine yönelik bir türetme şeması vardır. Örneğin MFC sınıf sisteminde *CException* isimli sınıf bu işlem için tasarlanmıştır. Standart kütüphanede de exception işlemleri için *exception* isimli sınıf ve bu sınıftan türetilen sınıflar kullanılmaktadır.

exception sınıflarının taban sınıfı *exception* sınıfıdır. Bu sınıf <exception> dosyası içindedir.

```

namespace std {
    class exception {
    public:
        exception() throw();
        exception(const exception&) throw();
        exception& operator=(const exception&) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();
    };
}

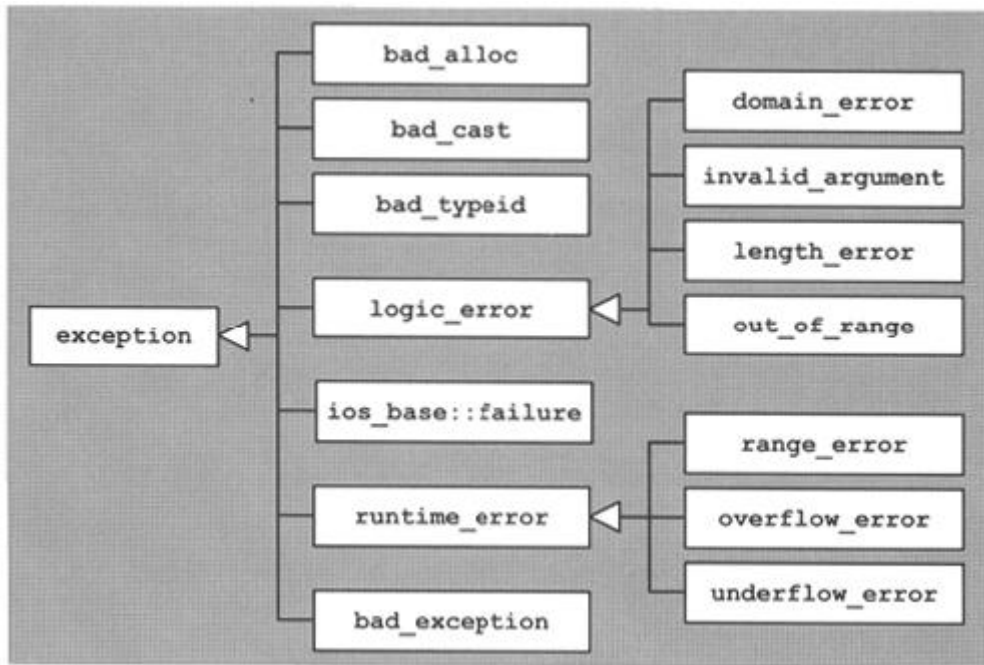
```

Buradaki en önemli fonksiyon *what* fonksiyonudur. *what* sanal bir fonksiyondur ve exception nedenine ilişkin bir yazının adresiyle geri döner. Standart kütüphanenin kendi içerisinde exception nesnesinin kendisiyle throw edip catch parametresi olarak referans kullanma yönetimi izlenmiştir. Programcı da bu modeli benimseyebilir.

```

1. throw X();
   catch(X a){}
2. throw X();
   catch(X &r){}
3. throw X();
   catch(X *p){
       ....
       delete p;
   }

```



Kütüphanede bulunan exception sınıflarının türetme şeması şöyledir.

Buradaki birçok sınıfın başlangıç fonksiyonu *const string&* türündendir. Bu sınıflar aldıkları bu yazıları sınıfın veri elemanında saklayarak *what* fonksiyonunda bu yazıyı verirler. En önemli iki ana kol *logic_error* ve *runtime_error* sınıflarıdır. *runtime_error* programın çalışma zamanı sırasında çıkabilecek hataları temsil etmektedir. *logic_error* ise daha çok programın çalışma zamanıyla ilgili olmayan uygunsuz kullanımlara yönelik düşünülmüştür.

Exception sınıfları zatan kütüphane içerisinde kullanılmaktadır. Programcı da bu sınıflardan faydalanabilir. Bunu iki biçimde yapabilir:

1. Hiç sınıf türetmez doğrudan bu exception sınıflarıyla throw eder, hangi sınıf uygunsa onu seçer. Yani *logic_error* ya da *runtime_error* sınıflarını kullanabileceği gibi bunlardan türetilmiş *invalid_argument*, *overflow_error* gibi sınıfları da kullanabilir.

```
try{
    if(test())
        throw runtime_error("test failure")
}
catch(exception &e){
    cout<<e.what()<<endl;
}
```

2. Programcı standart exception sınıflarından türetme yaparak bu sınıflar için *what* sanal fonksiyonunu yazabilir. Yine hangi sınıf uygunsa o sınıftan türetme yapar. Örneğin,

```
class something_exception:runtime_error{
public:
    something_exception(const string &reason):m_reason(reason)
    {}
    virtual const char *what const throw()
    {
        return m_reason.c_str();
    }
private:
    string m_reason;
}
```

Tabi programcı projesine uygun bir biçimde bir exception sınıf ağacı oluşturabilir.

Standart exception sınıfları birkaç başlık dosyasına yayılmıştır. Taban sınıf olan exception <exception> başlık dosyasındadır. Diğerlerinin çoğu (*runtime_error*, *logic_error* ve bunlardan türemiş olanlar) <stdexcept> başlık dosyasındadır.

Çokbiçimliliğe ilişkin Uygulamalar

Çokbiçimlilik nesne yönelimli programlama tekniğinin en önemli kavramlarından biridir. Çokbiçimliliğin sağladığı en önemli fayda programın bazı bölümlerinin türden bağımsız bir biçimde yazılmasına olanak sağlamaktır. Örneğin kodun bir bölümünde Shape isimli taban sınıf türünden pShape isimli gösterici ya da sanal fonksiyonlarla bir takım işlemler yapılmış olabilir. Bu kod Shape sınıfından türetilmiş herhangi bir şekil sınıfı için anlamlı ve geçerlidir. Yani kodun ilgili bölümü şekil ne olursa olsun geçerli ve değişmez bir biçimde kalabilecektir. Çok biçimlilik sayesinde programa yeni bir özellik eklenmesi kolaylaşır. Programcı ekleyeceği yeni özellik için programın çok büyük kısmını değiştirmek zorunda kalmaz. Kodun büyük bölümünün değiştirilmemesi böcek oluşumunu da proje büyüdükçe artmayacaktır.

Çok biçimlilik için şüphesiz bir türetme şeması gerekmektedir. Programcı proje içerisindeki gerçek nesneleri ve kavramları sınıflarla temsil eder. Bu sınıflarda ortak özellikler varsa taban sınıfları oluşturur ve sanal fonksiyonlar yardımıyla çok biçimli yapıyı kurar.

Undo Özelliği Olan Satır Tabanlı Editör Uygulaması

Satır tabanlı editörler (Line Editors) ekran işlemleri yoğun olmayan belli bir anda tek bir satır üzerinde işlemlerin yapılabildiği basit editörlerdir. Böyle bir uygulamada öncelikle çeşitli nesneler ve kavramlar sınıflarla temsil edilmeli ve program bu sınıfların etkileşimi ile oluşturulmalı. Örneğin satır tabanlı editör bir komut satırı içerir, komut satırı işlemleri Shell gibi bir sınıfla temsil edilebilir. Programın kendisi LineEditor isimli bir sınıfla temsil edilebilir. Programda dosya işlemleri yapmak gerekecektir. Bunun için standart fstream sınıfı kullanılabilir. Undo mekanizması, Undo isimli ayrı bir sınıf tarafından sağlanabilir. Programda bazı ekran işlemleri gerekebilir. Eğer gerekirse bu işlemlerde Screen gibi bir sınıfla temsil edilebilir. Shell sınıfı ile LineEditor sınıfı arasındaki ilişki türetme ilişkisi ya da içerme ilişkisi olabilir. Tasarım mümkün olduğu kadar ileride yapılacak benzer projelerde birikim oluşturacak biçimde yapılmalıdır. Örneğin buradaki Shell sınıfını programcı başka projelerde de kullanabilmek için genel tasarlayabilir. LineEditor bir çeşit Shell olmadığına göre LineEditor, Shell kavramını kullandığına göre içerme ilişkisi daha anlamlıdır.

Programın Undo işlemini yapan kısmı çokbiçimli tasarlanabilir. Yani kabaca bir soyut Undo sınıfı olur, sınıfın Undo işlemini yapan saf sanal bir MakeUndo gibi bir fonksiyonu bulunur. Undo gerektiren tüm durumlar bu Undo sınıfından türetilmiş olan sınıflarla temsil edilir. Bu durumda Undo için kullanılacak stack sistemi aşağıdaki gibi olabilir:

```
stack<Undo x> undostack;
```

Görüldüğü gibi undostack stak'in de adresler tutulmaktadır. Fakat bu adresler Undo sınıfından türetilmiş heterojen nesnelere ilişkindir. O halde undo işlemi için tek yapılacak şey stack' ten pop işlemi ile nesnenin adresini çekmek bu nesne adresi yardımıyla MakeUndo sanal fonksiyonunu çağırmasıdır. Undo sınıfından türetilmiş olan her sınıf kendi undo işlemini yapacaktır.

shell.h

```
#ifndef _SHELL_H_
#define _SHELL_H_
```

```
#include <string>
```

```
class CommandProcessor {
```

```
public:
```

```
    bool ProcessCommand(const std::string &cmd, const std::string &params)
```

```
    {
```

```
        return cmd != "quit";
```

```
    }
```

```
};
```

```
class Shell {
```

```
public:
```

```
    Shell(CommandProcessor *pCmdProc)
```

```
        : m_pCmdProc(pCmdProc), m_prompt("cmd"),
```

```
    m_seperator('>')
```

```
    {}
```

```
    Shell(CommandProcessor *pCmdProc, const char *pPrompt, char seperator = '>')
```

```
        : m_pCmdProc(pCmdProc), m_prompt(pPrompt),
```

```
    m_seperator(seperator)
```



```

    {}
    void Run();
private:
    bool parseCommandProc(const std::string &cmdLine);
private:
    CommandProcessor *m_pCmdProc;
    std::string m_prompt;
    char m_seperator;
};

```

```
#endif
```

shell.cpp

```
#include <iostream>
```

```
#include "shell.h"
```

```
using namespace std;
```

```

bool Shell::parseCommandProc(const string &cmdLine)
{
    const char *pDelims = "\\t";
    string::size_type pos1, pos2;
    string cmd, params;

    pos1 = cmdLine.find_first_not_of(pDelims, 0);
    if (pos1 == string::npos)
        goto EXIT;
    pos2 = cmdLine.find_first_of(pDelims, pos1);

    cmd = cmdLine.substr(pos1, pos2 - pos1);

    pos1 = cmdLine.find_first_not_of(pDelims, pos2);
    if (pos1 == string::npos)
        goto EXIT;

    params = cmdLine.substr(pos1);
EXIT:
    return m_pCmdProc->ProcessCommand(cmd, params);
}

```

```

void Shell::Run()
{
    string cmd;

    for (;;) {
        cout << m_prompt << m_seperator;
        getline(cin, cmd);
        if (!parseCommandProc(cmd))
            break;
    }
}

```

```
#if 1

class LineEditor {

};

class LineEditorCommandProcessor : public CommandProcessor {
public:
    LineEditorCommandProcessor(LineEditor *pLineEditor)
        : m_pLineEditor(pLineEditor)
    {

    }
    virtual bool ProcessCommand(const string &cmd, const string &params)
    {
        return true;
    }
private:
    LineEditor *m_pLineEditor;
};

int main(void)
{
    CommandProcessor cmdProc;
    Shell shell(&cmdProc);

    shell.Run();

    return 0;
}

#endif
```

calculator.h

```
#include "calculator.h"
#include <cstdlib>

using namespace std;

Calculator::Calculator()
{
}

Calculator::~Calculator()
{
}

void Calculator::Run()
{
}
```

```

        CalculatorCommandProcessor calcCommandProc(this);
        Shell shell(&calcCommandProc);

        shell.Run();
    }

    bool CalculatorCommandProcessor::ProcessCommand(const std::string cmd, const
std::string params)
    {
        if(cmd == "add") {
        }
    }

    int main(void)
    {
        return 0;
    }

```

Shell sınıfı yalnızca LineEditor uygulamasında değil, genel kullanım için tasarlanmalıdır. Bu sınıftan beklentimiz bir komut satırı çıkarması, komutları alarak parse etmesi ve komutu işletmek için kontrolü başka bir koda devretmesidir. Kontrolün başka bir koda devredilmesi için sanallık mekanizması kullanılabilir. Yani Shell sınıfı CommandProcessor gibi bir sınıf nesnesinin adresini alır, her parse ettiği komut ile CommandProcessor sınıfının ProcessCommand gibi sanal bir fonksiyonunu çağırabilir. Böylece Shell sınıfını kullanacak kişi CommandProcessor sınıfından türetme yaparak bu sanal fonksiyonu yazar.

Shell2.h

```

#ifndef _SHELL_H_
#define _SHELL_H_

#include <vector>
#include <string>

class CommandProcessor {
public:
    virtual bool ProcessCommand(const std::vector<std::string> &params)
    {
        return params[0] != "quit";
    }
};

class Shell {
public:
    Shell(CommandProcessor *pCmdProc)
        : m_pCmdProc(pCmdProc), m_prompt("cmd"),
m_seperator('>')
    {}
    Shell(CommandProcessor *pCmdProc, const char *pPrompt, char seperator = '>')
        : m_pCmdProc(pCmdProc), m_prompt(pPrompt),
m_seperator(seperator)
    {}

```

```

        void Run();
private:
        bool parseCommandProc(const std::string &cmdLine);
private:
        CommandProcessor *m_pCmdProc;
        std::string m_prompt;
        char m_seperator;
};

#endif

```

Shell2.cpp

```

#include <iostream>
#include "shell.h"

using namespace std;

bool Shell::parseCommandProc(const string &cmdLine)
{
    const char *pDelims = "\t";
    string::size_type pos1, pos2 = 0;
    vector<string> params;

    for (;;) {
        pos1 = cmdLine.find_first_not_of(pDelims, pos2);
        if (pos1 == string::npos)
            break;
        pos2 = cmdLine.find_first_of(pDelims, pos1);
        params.push_back(cmdLine.substr(pos1, pos2 - pos1));
    }
    return m_pCmdProc->ProcessCommand(params);
}

void Shell::Run()
{
    string cmd;

    for (;;) {
        cout << m_prompt << m_seperator;
        getline(cin, cmd);
        if (!parseCommandProc(cmd))
            break;
    }
}

#if 0

class LineEditor {

};

```

```

class LineEditorCommandProcessor : public CommandProcessor {
public:
    LineEditorCommandProcessor(LineEditor *pLineEditor)
        : m_pLineEditor(pLineEditor)
    {

    }
    bool ProcessCommand(const std::vector<std::string> &params)
    {
        copy(params.begin(), params.end(), ostream_iterator<string>(cout, "\n"));

        return params[0] != "quit";
    }
private:
    LineEditor *m_pLineEditor;
};

int main(void)
{
    LineEditorCommandProcessor cmdProc(new LineEditor());
    Shell shell(&cmdProc);

    shell.Run();

    return 0;
}

#endif
ledit.h
#ifndef _LEDIT_H_
#define _LEDIT_H_

#include <deque>
#include "shell.h"

class LineEditor;

class LineEditorCommandProcessor : public CommandProcessor {
public:
    LineEditorCommandProcessor(LineEditor *pLineEditor)
        : m_pLineEditor(pLineEditor)
    {}
    bool ProcessCommand(const std::vector<std::string> &params);
private:
    LineEditor *m_pLineEditor;
};

class LineEditor {
public:

```

```

        bool ProcessCommand(const std::vector<std::string> &params);
private:
    struct Command {
        const char *pCommandText;
        void (LineEditor::*pCommandProc)();
    };
private:
    void addLine();
    void typeLines();
    void deleteLine();
private:
    const std::vector<std::string> *m_pParams;
    std::deque<std::string> m_lines;
    static Command ms_commands[];

};

#endif
ledit.cpp
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include <cstdlib>
#include "ledit.h"

using namespace std;

LineEditor::Command LineEditor::ms_commands[] = {
    {"add", addLine},
    {"type", typeLines},
    {"delete", deleteLine},
    {0, 0}
};

bool LineEditorCommandProcessor::ProcessCommand(const std::vector<std::string>
&params)
{
    return m_pLineEditor->ProcessCommand(params);
}

bool LineEditor::ProcessCommand(const std::vector<std::string> &params)
{
    if (params[0] == "quit")
        return false;

    m_pParams = &params;

    for (int i = 0; ms_commands[i].pCommandText != 0; ++i) {

```

```

        if (ms_commands[i].pCommandText == params[0]){
            (this->*ms_commands[i].pCommandProc)();
            break;
        }
    }
    if (ms_commands[i].pCommandText == 0)
        cout << "Bad command or file name" << endl;

    return true;
}

void LineEditor::addLine()
{
    string str;

    cout << "Enter Line: ";
    getline(cin, str);
    m_lines.push_back(str);
}

void LineEditor::typeLines()
{
    copy(m_lines.begin(), m_lines.end(), ostream_iterator<string>(cout, "\n"));
}

void LineEditor::deleteLine()
{
    if (m_pParams->size() == 1) {
        cout << "line number missing!\n";
        return;
    }
    if (m_pParams->size() > 2) {
        cout << "too many parameters!\n";
        return;
    }
    vector<string>::size_type lineNumber = atoi((*m_pParams)[1].c_str());
    if (lineNumber > m_lines.size()){
        cout << "incorrect line number!\n";
        return;
    }
}

int main(void)
{
    LineEditorCommandProcessor cmdProc(new LineEditor());
    Shell shell(&cmdProc);

    shell.Run();

    return 0;
}

```

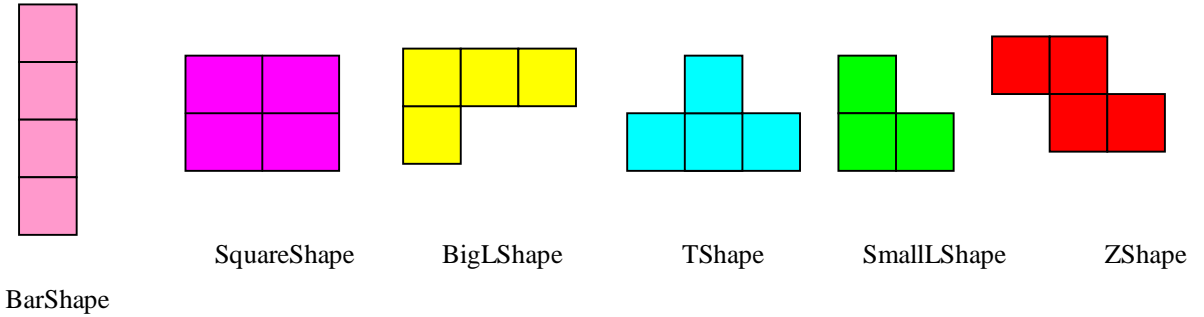
}

Anahtar Notlar:

Pekçok biçimli uygulamalarda genelde taban sınıfların bitiş fonksiyonları sanal yapılmalıdır. C++'da saf sanal bitiş fonksiyonu mümkün olsa da uygulamada bitiş fonksiyonun tanımlanmasının yin de yapılması gerekmektedir. Çünkü maalesef türemiş sınıfların bitiş fonksiyonu her zaman taban sınıfın bitiş fonksiyonunu çağırır. Eğer biz taban sınıfın bitiş fonksiyonunu saf sanal yapıp tanımlamasını bulundurmazsak link aşamasında taban sınıfın bitiş fonksiyonunun bulunmamasından problem ortaya çıkar. Genel olarak C++'da saf sanal fonksiyonun ayrıca tanımlanması yapılabilir. Fakat tanımlamanın sınıf bildiriminin dışında yapılması gerekir. Yani "=0" bildiriminde sonra bir daha blok açılmaz.

Tetris Oyun Programı Uygulaması

Tetris oyun tarihinin belki de en popüler programıdır. Oyunda bir takım şekiller düşmekte bu şekillerden satırlar tamamlanmaktadır. Tetrisekte belli başlı şekiller şunlardır:



Tetris oyununun kendisi bir sınıfla temsil edilebilir. Örneğin oyun sınıfın *Run* fonksiyonuyla başlatılabilir.

```
class Tetris{
public:
    //...
    void Run();
};
int main()
{
    Tetris tetris;
    Tetris.Run();
    //...
}
```

Tetris sınıfı için bir takım görüntü işlemleri gerekebilir. Bunun için *Tetris* sınıfı örneğin *Scr* isimli bir sınıftan faydalananıy olabilir. *Scr* türünden sınıf nesnesi *Tetris* sınıfının veri elemanı olarak kullanılabilir.

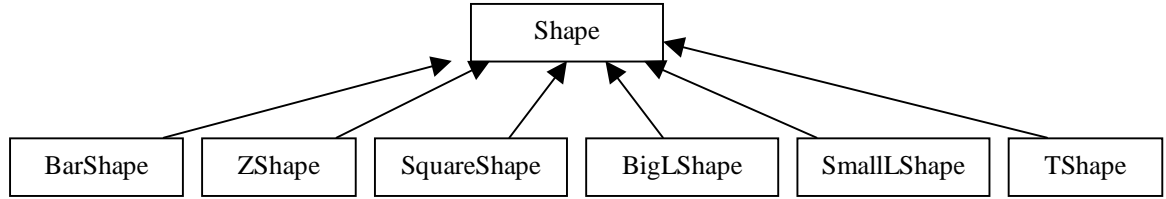
```
class Tetris{
public:
    //...
    void Run();
private:
    Scr m_scr;
```



```
};
```

Oyun için bir takım başlangıç işlemleri *Tetris* sınıfının başlangıç fonksiyonu içinde yapılabilir.

Tetris oyununda bütün şekiller *Shape* isimli bir sınıftan türetilmiş sınıflarla temsil edilebilir.



Shape sınıfının şekilleri hareket ettirmekte kullanılan bir grup sanal fonksiyonu olabilir. Her şekil kendi hareketlerini kendisi yapmaktadır. Böylece programcı şeklin hangi şekil olduğunu bilmeden genel kodlar yazabilir.

```
class Shape{
```

```
public:
```

```
    virtual void MoveDown() = 0;
```

```
    virtual void TurnRight() = 0;
```

```
    virtual void TurnLeft() = 0;
```

```
    virtual void MoveDownFast() = 0;
```

```
    virtual void MoveLeft() = 0;
```

```
    virtual void MoveRight() = 0;
```

```
    virtual ~Shape() = 0;
```

```
};
```

Run fonksiyonunda kabaca rasgele bir şekil alınmaktadır. Klavyeden basılan tuşa bakılarak şekil hareket ettirilmektedir. *Run* fonksiyonunun çatısı aşağıdaki gibi olabilir:

```
void Tetris:: Run(){
```

```
    for(;;){
```

```
        Shape *pShape = createNewShape();
```

```
    for(;;){
```

```
        pShape -> MoveDown();
```

```
        sleep(PERIOD);
```

```
        if(KeyboardPressed()){
```

```
            switch(getKey()){
```

```
                case LEFT:
```

```
                    pShape -> MoveLeft();
```

```
            case RIGHT:
```

```
                pShape -> MoveRight();
```

```
                break;
```

```
        }
```

```
    }
```

```
}
```

```
//...
```

```
    CheckLine();
```

```
}
```

```
}
```

Görüldüğü gibi *Run* fonksiyonu türden bağımsızdır. Oyuna yeni bir şekil eklemek için tek yapılacak şey yeni bir sınıf türetip sanal fonksiyonları yazmak ve *createNewShape*

fonksiyonunu güncellemektir. Şüphesiz şekillerin aşağıda bir sıra oluşturup oluşturmadığına her şekil düştükten sonra bakmak gerekir. Buna bakma işlemi tamamen görüntüsel ya da matematiksel olarak yapılabilir. Yani programcı şekillerin renklerinden hareketle bir satırın tamamlanıp tamamlanmadığını anlayabilir. Fakat şüphesiz aşağıda biriken şekillerin bir biçimde tutulması gerekebilir. Aşağıda biriken şekiller bir nesne tutan sınıfta toplanabilir ve puanlama yapılabilir.

Anahtar Notlar:

Karmaşık projelerde sembolik sabitleri #define önişlemci komutu yerine enum sabiti biçiminde bildirmek daha iyi bir tekniktir. Çünkü:

1. *enum sabitlerinin bir faaliyet alanı söz konusudur.*
2. *enum sabitleri derleme modülüne ilişkin olduğu için daha güvenilirdir.*
3. *Derleyici sistemlerinin debuggerları enum sabitlerini tanıyabilirler.*

Paint Brush gibi Bir Çizim Programının Tasarımı

Paint Brush benzeri bir çizim programı orta karmaşıklıkta bir proje konusu olabilir. Şüphesiz böyle bir program ağırlıklı bir biçimde çizim işlemleriyle ilgilidir. Çizim işlemleri ise kullanılan grafik sisteme göre değişen biçimlerde yapılabilmektedir. Burada bu çizim işlemlerinden ziyade şekiller için oluşturulacak çok biçimli yapı üzerinde durulacaktır.

Bu tür çizim programlarının en önemli özelliği çizilen şekillerin bir biçimde saklanması ve onların üzerine tıklanarak o nesnelerin seçilmesidir. Seçilen şekiller üzerinde çeşitli işlemler yapılabilmektedir. Kullanılan şekillerin hepsinin kendine özgü özellikleri olmakla birlikte çok biçimliliği oluşturacak olan ortak özellikleri de vardır. Örneğin, bir noktanın o şeklin içinde olup olmadığının belirlenmesi tipik bir çok biçimli özelliktir. Çünkü böyle bir özellik vardır, fakat her sınıfın bunu gerçekleştirmesi farklı biçimde olmaktadır. Gerçektende her şeklin bir noktanın kendi içinde olup olmadığını tespit etmek için ayrı bir algoritması vardır. Benzer biçimde yine tüm şekillerin seçilmesi ortak bir özelliktir. Fakat bu ortak özellik yine her şekil tarafından farklı bir biçimde gerçekleştirilmektedir. O halde şeklin seçilmesi de çok biçimli bir özelliktir. Yine her şeklin taşınması sırasındaki hareketi onlara özgüdür ve farklıdır.

Şüphesiz böyle bir çizim programında heterojen bütün şekillerin nesne tutan bir sınıfta saklanması gerekir. Veri yapısı olarak bağlı liste uygun gözükmemektedir.

Yine böyle bir programın tasarımı için şekiller *Shape* gibi bir sınıftan türetilmiş sınıflarla temsil edilebilir. *Shape* sınıfı aşağıdaki gibi soyut bir sınıf olabilir.

```
class Shape{
public:
    virtual bool IsInside(int x, int y) const = 0;
    virtual void Select() = 0;
    virtual void Move(int x, int y);
    //...
};
```

Tüm şekillerin tutulacağı veri yapısı şöyle olabilir:

```
list <Shape *> m_shapes;
```

Bu durumda fareyle tıklandığında tıklanan yerde bir şeklin olup olmadığı eğer bir şekil varsa onun seçilmesi işlemi şöyle yapılabilir:

```
for (list<Shape*>::iterator iter = m_shapes.begin(); iter != m_shapes.end(); ++iter){
    if ((*iter) -> IsInside(x, y)){
        (*iter) -> Select();
        break;
    }
}
```

Görüldüğü gibi tüm şekiller *IsInside* fonksiyonu ile kontrol edilmiş eğer tıklama işlemi bir şekil üzerine yapılmışsa o şekil *Select* fonksiyonuyla seçilmiştir. Seçilen şeklin silinmesi gibi bir işlem çok kolaydır. Şekil bağlı listeden çıkartılmalı ve görsel olarak da yok edilmelidir. Yine programın kendisi bir sınıf ile temsil edilebilir ve o sınıfın üye fonksiyonları programın temel işlemlerini yerine getirebilir.

Stringler ve char_traits İşlemleri

Bilindiği gibi *string* sınıfı *basic_string* sınıfının *char* açılımından başka bir şey değildir. *basic_string* sınıfının template parametreleri şöyledir:

```
template<class charT, class traits = char_traits<charT>,
class Allocator = allocator<charT> >
class basic_string;
```

Görüldüğü gibi sınıfın üç template parametresi vardır ve biz en az bir parametresini yazmak zorundayız. *string* aşağıdaki gibi bir typedef ismidir.

```
typedef basic_string<char> string;
```

Bu durumda

```
string s;
```

demekle

```
basic_string< char, class_traits <char>, allocator<char> >
```

aynı anlamdadır.

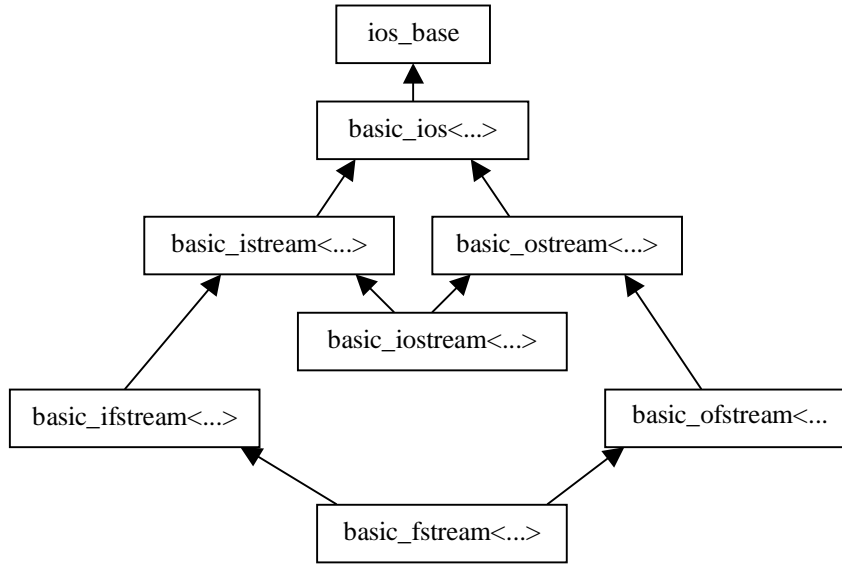
char_traits static fonksiyonlardan oluşan basit bir sınıftır. *string* sınıfı işlemlerini bu fonksiyonları çağırarak yapmaktadır. Örneğin, *string* sınıfının *find* fonksiyonu bir yazı içerisinde başka bir yazıyı bulmak için kullanılmaktadır. Fakat yazı içerisindeki karakterle aranan karakteri doğrudan karşılaştırmamakta *char_traits* sınıfının *eq* fonksiyonunu kullanmaktadır. Böylece eğer biz *basic_string* sınıfının davranışını değiştirmek istiyorsak *char_traits* sınıfına benzer bir sınıf yazmalıyız ve o sınıfı *basic_string* sınıfına template parametresi olarak geçirmeliyiz. Tabi bu işlem için *char_traits* özelliği gösterecek sınıfı sil baştan yazmak yerine o sınıftan türetme yaparak yalnızca istediğimiz fonksiyonları yazabiliriz.

char_traits sınıfına ilişkin en popüler örnek *basic_string* sınıfından faydalanarak büyük harf-küçük harf duyarlılığı olmayan bir *string* sınıfı yaratmaktır. Bunu sağlamak için *char_traits* sınıfından sınıf türetilir ve default büyük harf-küçük harf duyarlılığı olan fonksiyonların büyük harf-küçük harf duyarlılığı olmayan biçimleri yazılır. Sonra *basic_string* sınıfında bu sınıf template parametresi olarak verilir.

iostream Sınıf Sisteminin Ayrıntılı İncelenmesi

iostream sınıf sistemi her türlü okuma yazma işlemlerinde kullanılan ayrıntılı bir sınıf sistemidir.

iostream sınıf sisteminin temel yapısı aşağıdaki gibidir:



Buradaki template sınıflar genel olarak iki template parametresi alırlar. birinci template parametresi karakter olarak kullanılacak türü, ikinci template parametresi *trait* denilen işlem değiştirmek için faydanılacak türü belirtir.

```
template<class CharT, class traits = char_traits<CharT> >
class xxx{
    //...
};
```

iostream sınıf sistemindeki temel birim karakterdir ama bu karakterin hangi türle temsil edileceği template parametresiyle belirtilmektedir. tabi uygulamada karakter belirten tür ya ASCII karakterleri için kullandığımız `char` ya da UNICODE karakterler için kullandığımız `wchar_t` türüdür.

Anahtar Notlar:

Karakterin 1 byte ile ifade edildiği sisteme genellikle “narrow character” denilmektedir. “narrow character” tipik olarak C’ de `char` türüyle temsil edilmektedir. Her karakterin 2 byte ile temsil edildiği sisteme (örneğin UNICODE sistemi) “wide character” denir ve `wchar_t` türüyle temsil edilmektedir. `wchar_t`, C’ de bir typedef olmasına karşın C++’ da ayrı bir anahtar sözcük ve türdür. Bunların dışında bir de bazı karakterlerin 1 byte ile bazı karakterlerin ise 1’ den fazla byte ile temsil edildiği “multibyte character” kavramı vardır. Fakat bu sistem UNICODE sisteminden sonra terkedilmektedir.

iostream sınıf sistemi içerisindeki template sınıfların `char` ve `wchar_t` türleri için typedef edilmiş isimleri vardır.

```
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
typedef basic_iostream<char> iostream;
```

```
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;
```

```
typedef basic_istream<wchar_t> wistream;  
typedef basic_ostream<wchar_t> wostream;  
typedef basic_iostream<wchar_t> wiostream;
```

```
typedef basic_ifstream<wchar_t> wifstream;  
typedef basic_ofstream<wchar_t> wofstream;  
typedef basic_fstream<wchar_t> wfstream;
```

Yani örneğin bizim

```
basic_ifstream <char> is;
```

yazmamız ile

```
ifstream is;
```

aynı anlama gelmektedir.

iostream sınıf sisteminin fonksiyonları genel olarak dört tür işlem ile işlevlerini gerçekleştirir.

1. Parse etme
2. Tamponlama
3. Dönüştürme
4. Transfer

Tamponlama işlemi *basic_streambuf<...>* denilen bir sınıfla yapılmaktadır.

iostream sınıf sisteminin en önemli kavramlarından biri stream'in (stream sözcüğü dosya kavramı etrafında toplanabilecek nesneleri ifade eden genel bir terimdir.) durumunu belirten durum bilgisidir. Stream'in durumu için *ios_base* sınıfının public kısmında aşağıdaki bayraklar bulunmaktadır.

```
static const iostate badbit;  
static const iostate eofbit;  
static const iostate failbit;  
static const iostate goodbit;
```

Burada sözü edilen *iostate* seçimi derleyiciye bırakılmış bir türdür. Buradaki değerler tüm bitleri 0, yalnızca bir biti 1 olan sabitlerdir. Stream'in içinde bulunduğu durum *ios_base* sınıfının private veri elemanlarında saklanmaktadır.

goodbit Kavramı:

Eğer streamin durumu tamamen normale son işlemde hiçbir problem oluşmamışsa *ios_base* sınıfının *goodbit* bilgisini tutan elemanı set edilir.

eofbit Kavramı:

Son yapılan işlem dosya sonuna gelmekten dolayı başarısız olmuşsa *eofbit* set edilir.

failbit Kavramı:

Eğer son yapılan işlem ciddi bir *io* hatası dışındaki nedenlerden dolayı başarısız olmuşsa bu bir set edilmektedir. Bu bitin set edilmesi tipik olarak okuma işlemleriyle olur.

```
int a;
```

```
cin >> a;
```

Burada klavyeden sayısal değil alfabetik bir bilgi girdiğimizi düşünelim. Okuma işlemi başarısız olacaktır. Fakat bir *io* hatası söz konusu değildir, *failbit* set edilir. Dosya sonuna gelmekten dolayı okuma işlemi başarısız olursa hem *failbit* hem de *eofbit* set edilir.

badbit Kavramı

Bu bit telafi edilemez derecede *io* hataları oluştuğunda set edilir. Yani bu bit set edilmişse yapılacak bir şey yoktur.

Streamin durum bilgileri *basic_ios* sınıfının çeşitli fonksiyonlarıyla alınıp set edilebilir. *basic_ios* sınıfının bu bilgileri veren temel dört üye fonksiyonu şunlardır:

```
bool good() const;
bool fail() const;
bool bad() const;
bool eof() const;
```

fail fonksiyonu yalnızca *failbit* değil, *failbit* ya da *badbit* set edilmişse true değerini verir.

basic_ios sınıfının *setstate* fonksiyonu tümünden bütün bayrakları set etmekte kullanılabilir.

```
void setstate(iostate stat);
```

Bu fonksiyon ekleme yapma fikriyle çalışır. Örneğin, *eofbit* set edilmiş olsun. Bizde fonksiyonu şöyle çağıralım:

```
setstate(ios_base::failbit);
```

Şimdi hem *eofbit* set durumda olacaktır, hem de *failbit* set durumda olacaktır. *rdstate* isimli fonksiyon ise bitset olarak tüm durumu alır.

```
iostate rdstate() const;
```

clear fonksiyonu ise bir bütün olarak tüm durumu değiştirmektedir.

```
void clear(iostate stat = goodbit);
```

clear fonksiyonu ile *setstate* fonksiyonu arasındaki fark *clear* fonksiyonunun önce tüm bayrakları reset edip yalnızca belirtileni set etmesi, *setstate* fonksiyonunun ise diğer bayraklara dokunmadan yalnızca belirtileni set etmesidir. Bu durumda:

```
clear();
```

çağırması bütün error bayraklarını reset edip *goodbit* bayrağını set etmektedir.

Ayrıca *basic_ios* sınıfının konuyla ilgili iki operator fonksiyonu vardır.

```
operator void *() const;
bool operator !() const;
```

*void ** türüne dönüştürme yapan tür dönüştürme operator fonksiyonu *fail* fonksiyonunu çağırır. Eğer bu fonksiyon *true* verirse null adrese, *false* verirse 0 dışı herhangi bir adrese geri döner. *!* operator fonksiyonu ise *fail* fonksiyonunu geri dönüş değerine geri dönmektedir.

Streamin Durum Bilgilerinin İşlemlerde Kullanılması

Programcı herhangi bir *iostream* işleminden sonra stream durumuna bakarak yorum yapabilir. Eğer *goodbit* set edilmişse yaptığı işlemde hiçbir problem yoktur, *badbit* set edilmişse problem büyüktür yapılabilecek bir şey yoktur. *failbit* set edilmişse istenilen sonuç elde

edilememiştir, fakat bunun nedeni dosya sonuna gelmek olabileceği gibi bilginin yanlış bir formatta olması gibi bir neden de olabilir.

Örneğin, *basic_istream* sınıfının >> operator fonksiyonlarında format uyumsuzluğu nedeniyle bir hata oluştuğunda tipik olarak failbit set edilmektedir. Biz okuma işleminden sonra streamin durumuna bakıp güvenliği artırmaya çalışabiliriz.

Aşağıdaki kullanım en sık raslanan durumlardan biridir:

```
while(f >> a){
    ...
}
```

Burada >> operator fonksiyonunun ürettiği değer f nesnesinin kendisidir. O halde derleyici *basic_istream* nesnesini *bool* türüne dönüştürmek isteyecektir. Bu işi yapabilecek tek fonksiyon *basic_ios* sınıfının *void** türüne dönüştürme yapan operator fonksiyonudur, o fonksiyon çağırılacaktır. Yani yukarıdaki işlemin eşdeğeri şudur:

```
while((f >> a).operator void *()){
    ...
}
```

Bu işleminde eşdeğeri şudur:

```
while(!(f >> a).fail()){
    ...
}
```

O halde:

```
while(f >> a){
    ...
}
```

Bu durumda bu döngüden *failbit* ya da *badbit* set edildiğinde çıkarız. Dosya sonuna gelinmesi de *failbit*in set edilmesine yol açmaktadır. Tabi programcı döngüden çıktıktan sonra eğercek sebebi anlamak isteyebilir. O halde tipik bir dosya sonuna gelene kadar okuma işlemi bu biçimde yapılabilir.

```
while(cin >> a){
    cout << a;
}
```

>> operator fonksiyonları char okuması dışında önce boşlukları atmakta sonra boşluksuz karakter kümesini alıp yorumlamaktadır. Fakat char okumasında boşlukları da okumaktadır.

```
#include <iostream>
#include <string>
#include <cctype>
#include <fstream>
```

```
using namespace std;
```

```
int main()
{
    ifstream f("data");
    int n;

    while (f >> n) {
```

```

        cout << n << endl;
    }
    if (!f.eof()) {
        cerr << "Read fail!...\n";
        exit(EXIT_FAILURE);
    }

    return 0;
}

```

iostream Sınıfında Dosya İşlemleri

iostream sınıf sistemindeki dosya açış modları *ios_base* sınıfında aşağıdaki sabitlerle belirtilmiştir.

```

static const openmode app;
static const openmode ate;
static const openmode binary;
static const openmode in;
static const openmode out;
static const openmode trunc;

```

openmode derleyiciyi yazanlara bırakılmış olan bir tamsayı türünü temsil eder. *in* modu olan dosyayı açmak için kullanılır. *in* moduyla açılan dosyalardan yalnızca okuma yapılabilir. Dosyayı *in* moduyla açmaya çalışırsak fakat dosya yoksa *failbit* set edilir. *out* dosya yoksa yaratır, varsa olanı açar. *out* ile açtığımız dosyalara yalnızca yazma yapabiliriz. *trunc*, *out* ile birlikte kullanılabilir, tek başına bir anlam ifade etmez. Dosya olsa bile içeriği sıfırla anlamına gelir. *app* dosyaya ekleme yapmak için kullanılır, dosya *app* modunda açılırsa tüm yazılanlar atomik bir biçimde sona yazılır. *ate* diğer açış modları ile birlikte kullanılırsa dosyayı açtıktan sonra dosya göstericisini dosyanın sonuna çeker. *binary*, dosyayı binary modda açmak için kullanılır default text moddur. Bütün açış kombinasyonları geçerli değildir. Bir arada kullanılabilecek geçerli kombinasyonlar şunlardır:

in

out

out|trunc

out|app

in|out

in|out|trunc

Yukarıdakilerin hepsine *ate* ve *binary* kombine edilebilir. Bunların dışındaki diğer tüm seçenekler geçersiz kabul edilmektedir. Örneğin *in|trunc* biçiminde bir kombinasyon yoktur, fakat *in|ate* ya da *out|binary* kombinasyonları kullanılabilir.

basic_ifstream (yani *ifstream*) sınıfının default başlangıç fonksiyonunun yanısıra aşağıdaki gibi bir başlangıç fonksiyonu da vardır.

```
explicit basic_ifstream(const char* s, ios_base::openmode mode = ios_base::in);
```

Fonksiyonun birinci parametresi dosyanın ismi ikinci parametresi dosyanın açış modudur ve default olarak *ios_base::* biçiminde verilmiştir. Yani biz dosyayı şöyle açabiliriz:

```
ifstream f("test");
if(!f){
```



```
...
}
```

Dosya açılmazsa *basic_ios::failbit* set edilir. *basic_ifstream* sınıfında dosya açmak için başka bir yöntemde default başlangıç fonksiyonuyla nesneyi yaratıp daha sonra *open* ile dosyayı açmaktır. Örneğin;

```
ifstream f;
f.open("test");
if(!f){
    ...
}

ifstream f("test.cpp");
char ch;

//f >> noskipws;

if (!f) {
    cerr << "cannot open file!...\n";
    return 1;
}

while (f >> ch)
    cout << ch;

if (!f.eof()) {
    cerr << "read fail!...\n";
    return 1;
}
return 0;
```

Benzer biçiminde *ofstream* sınıfında default başlangıç fonksiyonunun yanısıra aşağıdaki gibi bir başlangıç fonksiyonu vardır.

```
explicit basic_ofstream(const char* s, ios_base::openmode mode= ios_base::out);
```

Görüldüğü gibi fonksiyonun ikinci parametresi *ios_base::out* default değerini almıştır. Dosya *ofstream* sınıfının başlangıç fonksiyonuyla nesne yaratıp, sonra *open* fonksiyonuyla açabilir. Örneğin;

```
ofstream f("test");
if(!f){
    ...
}

yada
ofstream f;
f.open("test");
if(!f){
    ...
}
```

Görüldüğü gibi *basic_ifstream* okuma amacıyla, *basic_ofstream* yazmak amacıyla kullanılmaktadır. Eğer hem okuma hemde yazma yapmak istiyorsak bu her iki sınıftan çoklu türetilen *basic_fstream* sınıfını kullanmalıyız.

basic_fstream sınıfının default başlangıç fonksiyonunun yanısıra aşağıdaki başlangıç fonksiyonu da vardır.

```
explicit basic_fstream(const char* s, ios_base::openmode mode = ios_base::in|ios_base::out);
```

Görüldüğü gibi bu modda dosya default olarak hem okuma hemde yazma modunda açılmaktadır. Tabi benzer biçimde dosya nesne default başlangıç fonksiyonu yaratıldıktan sonra *open* fonksiyonuyla da açılabilir.

```
fstream f("test");
```

```
if(!f){
    ...
}
yada
fstream f;
f.open("test");
if(!f){
    ...
}
```

Error Bayraklarının Set Edilmesi Durumunda *iostream* Sınıfının Davranışları

iostream sınıf sisteminde *eofbit*, *failbit* ya da *badbit* bayraklarının herhangi birisi set edilmişse sınıfların okuma ve yazma yapan üye fonksiyonları hemen başarısızlıkla geri dönmektedir. Örneğin, dosyanın sonuna kadar okuya okuya gidelim, dosyanın sonuna geldiğimizde *eofbit* set edilecektir. Şimdi bu biti *clear* yapmadan *seek* işlemi bile yapamayız. O halde bu bayraklardan biri set edildiğinde işleme devam etmek için öncelikle *f.clear()*; çağırması yaparak error bayraklarını set etmeliyiz.

```
int main(void)
{
    cout.clear(ios::eofbit);
    cout.clear();

    cout << 100;
    return 0;
}
```

```
int main(void)
{
    int a;
    cin.clear(ios::eofbit);
    cout.clear();

    cin >> a;
    cout << a << endl;
    return 0;
}
```

<< ve >> Operator Fonksiyonlarının Tipik Davranışları

basic_istream sınıfının >> operator fonksiyonları ve *basic_ostream* sınıfının << operator fonksiyonları her zaman karakter temelinde işlem yaparlar. Yani dosyanın binary modda

açılmasıyla bunun bir ilgisi yoktur. Bu fonksiyonlar C’ deki *fprintf*, *fscanf* fonksiyonuna benzetilebilir.

```
int main(void)
{
    ofstream f("data");
    if (!f) {
        cerr << "cannot open file!...\n";
        exit(EXIT_FAILURE);
    }
    for (int i = 1; i <= 100; ++i)
        f << i << endl;

    return 0;
}
```

```
int main(void)
{
    ifstream f("data");

    if (!f) {
        cerr << "cannot open file!...\n";
        exit(EXIT_FAILURE);
    }

    int i;

    while (f >> i)
        cout << i << endl;

    if (!f.eof()) {
        cerr << "read error\n";
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

basic_istream sınıfının >> operator fonksiyonları default olarak baştaki boşlukları (leading space) atar. İlk boşluksuz olan yazı kümesi içerisinde okuma türüne uygun olan karakterleri okur. Okuma türüne uygun olmayan ilk karakterde durur. Eğer hiçbir şey okuyamazsa *failbit* set eder.

basic_istream sınıfının char& parametrelili >> operator fonksiyonu da boşluk karakterlerini atmaktadır. Halbuki *fscanf* fonksiyonunda %c ile boşluk karakterleri atılmamaktadır.

```
int main(void)
{
    ifstream f("data");

    if (!f) {
        cerr << "cannot open file!...\n";
        exit(EXIT_FAILURE);
    }
}
```

```

    }

    char i;

    while (f >> i)
        cout << i;

    if (!f.eof()) {
        cerr << "read error\n";
        exit(EXIT_FAILURE);
    }
    return 0;
}

```

Fakat istersek bu >> operator fonksiyonlarının baştaki boşluk karakterlerini atmasını engelleyebiliriz. Bunun için *ios_base* sınıfında *unsetf* fonksiyonuyla *skipws* bayrağı reset edilmelidir. Örneğin;

```

int main(void)
{
    ifstream f("data");

    if (!f) {
        cerr << "cannot open file!...\n";
        exit(EXIT_FAILURE);
    }

    f.unsetf(ios_base::skipws);

    char i;

    while (f >> i)
        cout << i;

    if (!f.eof()) {
        cerr << "read error\n";
        exit(EXIT_FAILURE);
    }

    return 0;
}

```

Bu bayrağın reset edilmesi *noskipws* manipulatorüyle de aşağıdaki gibi yapılabilir:

```
f >> noskipws;
```

basic_istream sınıfının *string&* parametrelili >> operator fonksiyonu önce boşlukları atar (default durumda). Sonra ilk boşluksuz yazı kümesini *string*’e yerleştirir.

```

int main(void)
{
    ifstream f("data");

    if (!f) {

```

```

        cerr << "cannot open file!...\n";
        exit(EXIT_FAILURE);
    }

    string s;

    f >> s;

    cout << s;

    return 0;
}

```

Sınıfın `char*` parametrelili `>>` operator fonksiyonu ilk boşluksuz yazıyı dizinin içerisine yerleştirir, tabi burada dizinin yeterince geniş açılmış olması gerekir.

```

int main(void)
{
    ifstream f("data");

    if (!f) {
        cerr << "cannot open file!...\n";
        exit(EXIT_FAILURE);
    }

    string name, surname;
    int no;

    while (f >> name >> surname >> no)
        cout << name << " " << surname << " " << no << endl;

    return 0;
}

```

Dosya Göstericisinin Konumlandırılması

basic_istream sınıfının dosya göstericisi ile *basic_ostream* sınıfının dosya göstericisi farklıdır, bir tanesi *seekg* diğeri *seekp* ile konumlandırılır. Fakat *basic_iostream* sınıfı ile yani hem yazma hem de okuma işlemlerini yapmak için dosya göstericisini konumlandırıcaksak bu işlemi *seekg* ya da *seekp* fonksiyonlarından herhangi biriyle yapabiliriz. hangisi ile yaptığımızın bir önemi yoktur, zaten bu sınıfta bu iki fonksiyon çakışıktır. Yani birisiyle dosya göstericisini konumlandırıdığımızda diğeri de bu işlemde etkilenir. Özetle, dosyayı *ifstream* ile açmışsak konumlandırmayı *seekg* fonksiyonuyla, *ofstream* ile açmışsak *seekp* fonksiyonuyla *fstream* ile açmışsak *seekg* ya da *seekp* fonksiyonuyla yapmalıyız. Konumlandırma fonksiyonların tek parametrelili biçimleri baştan itibaren konumlandırmada, iki parametrelili biçimleri tıpkı *fseek* fonksiyonundaki gibi orjinli konumlandırma da kullanılır. *seekp* fonksiyonu da aynı biçimdedir. Benzer biçimde bu sınıfların *tellg* ve *tellp* fonksiyonları dosya göstericisinin konumlarını almakta kullanılır.

```

int main(void)
{
    ifstream f("data");

    if (!f) {

```

```

        cerr << "cannot open file!...\n";
        exit(EXIT_FAILURE);
    }

    string name, surname;
    int no;

    while (f >> name >> surname >> no)
        cout << name << " " << surname << " " << no << endl;

    f.clear();
    f.seekg(0);

    while (f >> name >> surname >> no)
        cout << name << " " << surname << " " << no << endl;

    return 0;
}

```

getline Fonksiyonları

getline fonksiyonları text işlemi yaparken oldukça faydalı fonksiyonlardır. Belirli bir karakter görene kadar okuma yaparlar. Bu karakter default argümanla belirtilmiştir ve default olarak ‘\n’ karakteridir. *getline* fonksiyonları *basic_istream* sınıfının ve std isim alanının global fonksiyonu olarak bulunmaktadır. En çok kullanılan *getline* fonksiyonu string parametrelili *getline* fonksiyonudur. Bu fonksiyon üye fonksiyon olarak değil global fonksiyon olarak yazılmıştır. Bu fonksiyonun birinci parametresi *basic_istream* sınıfı türünden bir sınıf referansı, ikinci parametresi bir string referansıdır. Fonksiyonun birkaç biçimi vardır. Bu fonksiyonlar *basic_istream* nesnesine geri döndüklerinden dolayı error bayrakları testine sokulabilirler. Örneğin bir dosyayı satır satır okuma şöyle yapılabilir:

```

int main(void)
{
    fstream f("data");

    if (!f) {
        cerr << "cannot open file!...\n";
        exit(EXIT_FAILURE);
    }

    string s;

    while (getline(f, s))
        cout << s;

    return 0;
}

```

Satırın sonundaki ‘\n’ karakteri yerleştirilmemektedir.

Bu biçimdeki okuma da şöyle yapılabilir:

```

int main(void)
{

```

```

    fstream f("data");

    if (!f) {
        cerr << "cannot open file!...\n";
        exit(EXIT_FAILURE);
    }

#define BUFSIZE          100

    char s[BUFSIZE];
    while (f.getline(s, BUFSIZE))
        cout << s << endl;

    return 0;
}

```

getline fonksiyonu dosya sonuna gelene kadar ya da BUFSIZE-1 karakteri okuyana kadar ya da ‘\n’ karakterini okuyana kadar (‘\n’ okunur ama yerleştirilmez) işlemini devamm ettirir.

Binary İşlem Yapan Fonksiyonlar

Belirli bir adresten itibaren n kadar byte’ ı okuyup yazan fonksiyonlardır. *basic_istream* sınıfının *read* isimli fonksiyonu dosya göstericisinin gösterdiği yerden n byte okumak için kullanılabilir.

```
basic_istream<charT,traits>& read(char_type* s, streamsize n);
```

Görüldüğü gibi fonksiyon void* değil char* parametre almaktadır. Benzer biçimde *basic_ostream* sınıfının da *write* isimli bir üye fonksiyonu vardır.

```
basic_ostream& write(const char_type* s, streamsize n);
```

Bunların dışında *basic_istream* sınıfının benzer *get* isimli fonksiyonları *basic_ostream* sınıfının ise *put* isimli fonksiyonları vardır.

Görüldüğü gibi *read* ve *write* fonksiyonları ne kadar bilginin okunup yazıldığı bilgisinin vermemektedir. Son formatsız okuma ya da yazma miktarı sınıfların *gcount* ve *pcount* fonksiyonlarıyla elde edilebilir.

```

bool CopyFile(const char *source, const char *dest)
{
    const int BUFSIZE = 1024;

    ifstream fi(source, ios_base::in|ios_base::binary);

    if (!fi)
        return false;

    ofstream fo(dest, ios_base::out|ios_base::binary);

    if (!fo)
        return false;

    char buf[BUFSIZE];

```

```

    unsigned n;

    while (fi.read(buf, BUFSIZE), (n = fi.gcount()) > 0)
        if (!fo.write(buf, n))
            return false;
    if (!fi.eof())
        return false;

    return true;
}

int main(void)
{
    if (!CopyFile("test.cpp", "testcpp.bak")) {
        cerr << "cannot copy file!...\n";
        exit(EXIT_FAILURE);
    }

    cout << "ok\n";

    return 0;
}

```

ignore Fonksiyonu

Tampondan belirli karakterleri atmak için C’ de maalesef standart bir fonksiyon yoktur. Bu durum özellikle klavye okumaları sırasında problem olmaktadır. Fakat C++’ ın *ostream* kütüphanesinde belirli bir karakteri görene kadar tampon boşaltılabilmektedir.

basic_istream<charT,traits>& ignore (streamsize n = 1, int_type delim = traits::eof());

Fonksiyonun birinci parametresi maximum atılacak karakter sayısı, ikinci parametresi hangi karakter görülene kadar atma işleminin yapılacağıdır. Birinci parametreye çok büyük değer geçebiliriz. Örneğin:

cin.ignore(numeric_limits<streamsize>::max(), '\n');

Burada fonksiyonun birinci parametresi *streamsize* hangi türdense onun en büyük değeri olarak girilmiştir. İkinci parametre *lf* karakteridir yani enter tuşuna basılana kadar tüm karakterler atılmaktadır.

```

#include <iostream>
#include <limits>

using namespace std;

int main()
{
    char ch1, ch2;

    cin >> noskipws;

    cin >> ch1;
    cout << ch1 << endl;
}

```



```

        cin.ignore(numeric_limits<streamsize>::max(), '\n');

        cin >> ch2;
        cout << ch2 << endl;

        return 0;
    }
}

```

Örneğin bu fonksiyon sayesinde bir dosyadaki satır sayısı şöyle bulunabilir:

```

int main()
{
    ifstream fi("sample2.cpp");

    if (!fi) {
        cerr << "Cannot open file!..\n";
        exit(EXIT_FAILURE);
    }

    int i;
    for (i = 0; fi.ignore(numeric_limits<streamsize>::max(),
'\n'); ++i)
        ;

    cout << i << endl;

    return 0;
}

```

Database örneği:

```

#include <iostream>
#include <string>
#include <limits>
#include <fstream>
#include <cstdlib>
#include <exception>
#include <io.h>

using namespace std;

struct Person {
    char name[30];
    int no;
};

enum {ADDREC = 1, DISPREC, EXIT};

int GetMenu();
void AddRec();
void DispRec();

fstream g_f;

```

```
int GetMenu()  
{  
    int option;  
  
    cout << "1) Kayit ekle" << endl;  
    cout << "2) Listele" << endl;  
    cout << "3) Cikis" << endl;  
    cout << "Secim: ";  
    cin >> option;  
  
    if (cin.fail())  
        throw runtime_error("Invalid option");  
  
    cin.ignore(numeric_limits<streamsize>::max(), '\n');  
  
    return option;  
}  
  
int main()  
{  
    try  
    {  
        if (_access("data", 0) == 0)  
            g_f.open("data");  
        else  
  
            g_f.open("data", ios_base::in|ios_base::out|ios_base::trunc  
);  
  
        if (!g_f) {  
            cerr << "Cannot open file!...\n";  
            exit(EXIT_FAILURE);  
        }  
  
        for (;;) {  
            int option;  
  
            option = GetMenu();  
            switch (option) {  
                case ADDREC:  
                    AddRec();  
                    break;  
                case DISPREC:  
                    DispRec();  
                    break;  
                case EXIT:  
                    return 0;  
            }  
        }  
        return 0;  
    }  
}
```

```

catch (exception &e)
{
    cerr << "Fatal error: " << e.what() << endl;
    exit(EXIT_FAILURE);
}

void AddRec()
{
    Person per;

    cout << "Adi Soyadi: ";
    cin.getline(per.name, 30, '\n');
    cout << "No: ";
    cin >> per.no;

    g_f.seekp(0, ios_base::end);

    g_f.write(reinterpret_cast<char *>(&per), sizeof(Person));
}

void DispRec()
{
    Person per;

    g_f.seekg(0, ios_base::beg);

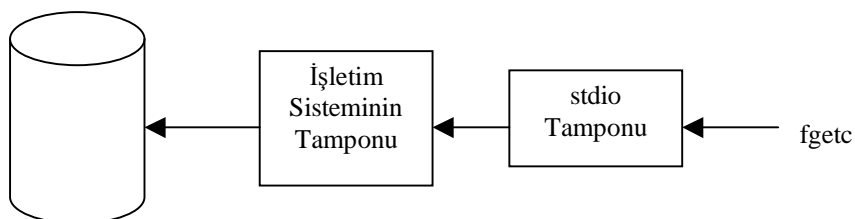
    while (g_f.read(reinterpret_cast<char *>(&per),
sizeof(Person)), g_f.gcount() > 0)
        cout << per.name << "    " << per.no << endl;

    if (g_f.bad())
        throw runtime_error("Read error");
}

```

iostream Kütüphanesi ve Tamponlama İşlemleri

Bilindiği gibi kullanıcı düzeyindeki dosya kütüphanelerinin çoğu bir tamponlama mekanizması içermektedir. Bir dosyayla ilgili işlem yapılırken dosyanın bir bölümünün bir tampona çekilmesi ve küçük okuma ve yazmalarda o tamponun kullanılması performansı arttırmaktadır. *stdio* ve *iostream* kütüphaneleri aslında gerçek okuma yazmalar için işletim sisteminin API fonksiyonlarını kullanır. Modern işletim sistemlerinin çoğu diskin belirli bir bölümünü RAM’de tutarak zaten aşağı seviyeli bir cash sistemi oluşturmaktadır. Fakat işletim sisteminin API fonksiyonlarını çağırmanın da bir maliyeti vardır. İşte tamponlama işlemi bu maliyeti azaltmak için kullanılır.

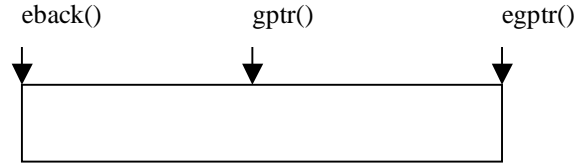


iostream sınıf kütüphanesinde tamponlama işlemi *basic_streambuf* isimli bir sınıfa bırakılmıştır. Yani bu sınıf tamponlama işlemi yapan sınıftır ve *iostream* kütüphanesindeki tamponlamalarda bu sınıf kullanılmaktadır. Ayrıca *iostream* sınıf sistemi kullanılan tamponlama mekanizmasının programcı tarafından değiştirilmesine olanak sağlamaktadır. *basic_istream* ve *basic_ostream* sınıflarının başlangıç fonksiyonları *basic_streambuf* sınıfı nesnesini almakta ve tamponlama da bu nesneyi kullanmaktadır. Bu durumda programcı *basic_streambuf* sınıfından bir sınıf türeterek kendi tamponlama mekanizmasını oluşturabilir ve bu mekanizmanın kullanılmasını sağlayabilir.

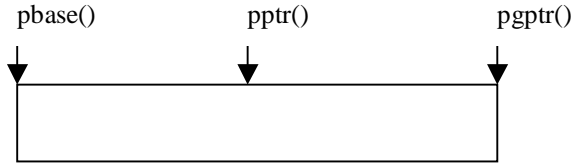
basic_streambuf Sınıfının Çalışması

Bu sınıf okuma ve yazma için ayrı tamponlar oluşturulabilecek biçimde tasarlanmıştır. okuma ve yazma tamponlarının başı, sonu ve aktif noktası birer gösterici ile belirtilmiştir. Bu göstericilerin değerlerini veren üye fonksiyonlar vardır.

OKUMA TAMPONU



YAZMA TAMPONU



Bu göstericilere değerlerini atayan iki fonksiyon vardır:

```
void setg(char_type* gbeg, char_type* gnext, char_type* gend);
```

```
void setp(char_type* pbeg, char_type* pend);
```

Sınıfın tampondan karakteri alıp bunu tampondan silen *sbumpc* isimli fonksiyonu vardır. *Sbumpc*, eğer *gptr* değeri null ise ya da *gptr* \geq *egptr* ise *uflow* isimli bir sanal fonksiyonu çağırır. *uflow* isimli sanal fonksiyon ise *underflow* isimli bir sanal fonksiyonu çağırır. Eğer *gptr* tampon içerisinde kalıyorsa bu göstericinin gösterdiği yerdeki bilgiyi alır ve *gptr* yi 1 artırır. *uflow* fonksiyonu default olarak *underflow* fonksiyonunu çağırdıktan sonra *gptr* yi 1 artırmaktadır. Yani programcı eğer *underflow* fonksiyonunu override edecekse tamponu tazelemeli fakat *gptr* göstericisini artırmamalıdır. *uflow* fonksiyonunu yazacaksa tamponu tazeledikten sonra *gptr* göstericisini artırmalıdır.

sbumpc *uflow* *underflow*

Karakter almakta kullanılan diğer bir fonksiyon *sgetc* fonksiyonudur. Bu fonksiyon karakteri alır fakat tampondan atmaz yani *gptr* göstericisini ilerletmez. Eğer *gptr* tampon içerisinde değilse ya da null ise doğrudan *underflow* fonksiyonunu çağırır.

sgetc *underflow*

```
int_type sgetc();
```

underflow fonksiyonu şöyledir:

```
int_type underflow();
```

Bu fonksiyonu override eden kişi şunları yapmalıdır:

1. Tamponu aygıt okuması yaparak tazelemeli *eback()* , *gptr* ve *egptr* göstericilerine yeni değerleri atamalıdır.
2. Tazelenen tampondaki ilk karakterle geri dönmelidir.
3. Eğer aygıttan okuma yapamıyorsa *traits::eof* değeriyle geri dönmelidir.

Aslında programcının *uflow* fonksiyonunu yazmasına gerek yoktur. Zaten default *uflow* kendi içerisinde *underflow* fonksiyonunu çağırır.

Sınıfın tampona yazma yapmakta kullanılan en önemli fonksiyonu *sputc* fonksiyonudur.

```
int_type sputc(char_type c);
```

Bu fonksiyon yazmıya tamponunda *pptr* göstericisinin gösterdiği yere yazmaya çalışır. Eğer *pptr()* \geq *epptr()* olmuş ise yani yazma tamponu dolmuş ise fonksiyon *overflow* sanal fonksiyonunu çağırır.

```
int_type overflow(int_type c = traits::eof());
```

overflow fonksiyonunu override eden programcının şunları yapması gerekir:

1. Tampon dolduğuna göre tampondaki tüm bilgileri aygıtı aktarıp tazeleme yapmalıdır.
2. *pbase()*, *pptr()* ve *epptr()* göstericilerine tampona göre değerlerini vermelidir.
3. *overflow* fonksiyonuna geçirilen parametre tampona yazılacak değerdir. Dolayısıyla programcının yazma tamponunu tazeledikten sonra bu karakteri tampona yazıp *pptr* göstericisini ilerletmesi gerekir. Eğer yazılacak karakter *traits::eof* ise herhangi bir tampona yazma yapılmamalıdır.

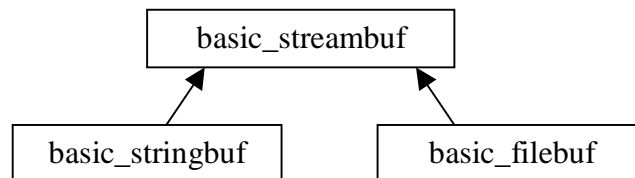
basic_streambuf Sınıfının Diğer Önemli Fonksiyonları

Bu fonksiyonların dışında sınıfın *sgetn* ve *sputn* isimli tampona n karakter yazan ve tampondan n karakter okuyan iki fonksiyonu vardır. Bu fonksiyonlar da tampon koşulu sağlanmamışsa tazeleme amaçlı *underflow* ve *overflow* sanal fonksiyonlarını çağırırlar. Bu fonksiyonlarında dışında tampona karakteri geri bırakan fonksiyonlar vardır.

```
int_type sputbackc(char_type c);
```

basic_streambuf Sınıfından Türetilen Tamponlama Sınıfları

basic_streambuf sınıfı tamponlama mekanizmasını belirleyen bir taban sınıftır. *iostream* sınıfları tamponlama için bu sınıftan türetilmiş olan *basic_filebuf* ve *basic_stringbuf* sınıflarını kullanmaktadır.



Dosya işlemlerinde *basic_filebuf* sınıfı türünden bir nesne, string tabanlı stream işlemlerinde *basic_stringbuf* türünden nesne kullanılmaktadır. Örneğin *ifstream*, *ofstream* sınıflarının başlangıç fonksiyonları *basic_filebuf* türünden bir nesneyi oluşturmakta ve tamponlamayı bu

nesneyi kullanarak yapmaktadır. Biz de aslında dosya işlemlerinde bu sınıfın yaptığı tamponlamadan faydalanmış oluruz.

Bilindiği gibi okuma yazma işlemini yapan fonksiyonlar aslında *basic_ostream* ve *basic_istream* sınıfının fonksiyonlarıdır. O halde tampona başvuracak olan fonksiyonlar bu sınıfın fonksiyonlarıdır. Gerçekten de dosya sınıfları *basic_filebuf* nesnesini oluşturduktan sonra sanki *basic_streambuf* nesnesiymiş gibi onu *basic_ostream* ve *basic_istream* sınıflarının kullanımlarına sunarlar. Yani *basic_ostream* ve *basic_istream* sınıfları kullandıkları tamponlama nesnesinin *basic_filebuf* olduğunu bilmezler. Fakat çok biçimli işlemler sonucunda *basic_filebuf* sınıfının fonksiyonları çağırılır.

Tamponlama sınıflarının böyle karışık tasarlanmasının nedeni programcının kendi tamponlama mekanizmasını devreye sokmasına olanak sağlamak içindir. Programcı kendi tamponlama sınıfını yazmayacak olduktan sonra bu sınıfların detaylı çalışmasını bilmek zorunda değildir.

Tamponlama Sınıflarının Temel iostream Sınıfları Tarafından Kullanılması

Tamponlama nesnesi *basic_ifstream*, *basic_ofstream*, *basic_fstream* gibi sınıflarda veri elemanı olarak tutulmaktadır. Fakat bu sınıfların başlangıç fonksiyonları sırasında tamponlama nesnesinin adresi *basic_ios* sınıfındaki *private* bir göstericinin içerisine yerleştirilir. Yani tamponlama nesnesi bu sınıfların veri elemanı biçimindedir. Fakat aynı zamanda bu nesnenin adresi *basic_ios* sınıfında da tutulmaktadır. Programcı isterse kullanılan tamponlama nesnesinin adresini bu sınıfın *rdbuf* üye fonksiyonu ile elde edebilir, hatta bunu değiştirebilir.

```
basic_streambuf<charT,traits>* rdbuf() const;
basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);
```

Birinci fonksiyon *basic_streambuf* nesnesinin adresini vermekte ikincisi ise bunu set etmekte kullanılır. *basic_ios* sınıfının *protected init* fonksiyonu da türemiş sınıfların bu göstericiyi set etmesi için kullanılır.

```
void init(basic_streambuf<charT,traits>* sb);
```

basic_streambuf sınıfı tamponda biriktirilmiş olan bilgilerin transferini yapan bir sınıf değildir. Tampondaki bilgilerin transfer işlemi *basic_streambuf* sınıfından türetilmiş olan sınıflarda yapılmaktadır. Örneğin *basic_filebuf*, *basic_streambuf* sınıfından türetilmiş tamponlama sınıfıdır, aynı zamanda transferin kendisini de yapmaktadır. *basic_ifstream*, *basic_ofstream* ve *basic_fstream* sınıfları *basic_filebuf* türünden bir tamponlama nesnesi bulundurur. Bu nesne hem tamponlama hem de transfer amacıyla kullanılmaktadır. Biz *basic_ifstream* türünden bir nesne tanımlayıp dosya açmış olalım. Biz şimdi bu nesne ile *basic_ios* sınıfının *rdbuf* fonksiyonunu çağırırsak ve tampon nesnesinin adresini alsak, bu adres bize sanki *basic_streambuf* nesnesinin adresiymiş gibi verilecektir. Halbuki bu adresteki nesne çok biçimli olarak *basic_filebuf* nesnesidir.

Bu anlatımlardan çıkan önemli sonuç şudur: Biz bir *iostream* nesnesine ilişkin *rdbuf* fonksiyonuyla onun tampon nesnesinin adresini alırsak bu tamponlama nesnesi yoluyla doğrudan dosya işlemlerini bile yapabiliriz. Örneğin,

```
#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;
```

```

int main(void)
{
    ifstream fi("buffer.cpp");

    if (!fi) {
        cerr << "Cannot open file!...\n";
        exit(EXIT_FAILURE);
    }

    streambuf *pbuf = fi.rdbuf();

    char s[100];

    pbuf->sgetn(s, 99);
    s[99] = '\0'
    cout << s << endl;
}

```

Biz bir sınıfın *basic_streambuf* nesne adresini diğer sınıfa vererek bir yönlendirme yapabiliriz. Örneğin;

```

#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

int main()
{
    ofstream fo("text.txt");

    if (!fo) {
        cerr << "Cannot open file!...\n";
        exit(EXIT_FAILURE);
    }

    cout.rdbuf(fo.rdbuf());
    cout << "This is a test\n" << endl;

    return 0;
}

```

Görüldüğü gibi *cout* ile yazılanlar artık dosyaya yazılacaktır. *basic_ostream* sınıfının << operator fonksiyonları ve *basic_istream* sınıfının >> operator fonksiyonları aslında yalnızca parse işlemini yapmaktadır. Tamponlama ve transfer işlemi tamponlama sınıf nesneleri tarafından yapılmaktadır. Daha açık bir biçimde söylenirse örneğin biz *basic_ostream* sınıfının << operator fonksiyonuyla bir şeyler yazdırmış olalım:

```
os << ".....";
```

<< operator fonksiyonunun yaptığı tek şey yazılacakları parse ederek belirlemek ve tamponlama nesnesini kullanarak tampona yazmaktır. Aygıtı aktarım tamamen bu tamponlama nesnesinin içerisinde gerçekleşmektedir. *istream* sınıflarında aslında işin çok büyük kısmı daha açık bir deyişle parsing işlemi dışındaki kısmı tamponlama sınıfları

tarafından yapılmaktadır. Yani elimizde tamponlama nesnesinin adresi olsa biz dosyalama işlemlerini yine yaparız, fakat zor yaparız. *basic_ostream* ve *basic_istream* sınıflarının fonksiyonları tamponlama nesnesini kullanan birer ara birim fonksiyonlar gibidir. Aslında *basic_ifstream*, *basic_ofstream* ve *basic_fstream* sınıfları da birer yardımcı sınıftır. İşletim sisteminin dosya sistemine ilişkin kullandığı açılan dosyanın handle değeri bu sınıflarda değil *basic_filebuf* sınıfında tutulur. Hatta dosyanın açılması bile aslında *basic_filebuf* sınıfı tarafından yapılmaktadır. Biz *basic_ifstream* sınıfının *open* fonksiyonunu çağırdığımızda bu fonksiyon da aslında tamponlama nesnesi yoluyla *basic_filebuf* sınıfının *open* fonksiyonunu kullanarak açılım yapmaktadır.

En ilginç noktalardan biri de *basic_ostream* sınıfının *basic_streambuf** parametrelili bir << operator fonksiyonunun olmasıdır. Bu operator fonksiyonu parametresiyle aldığı *basic_streambuf* göstericisini kullanarak dosya sonuna gelene kadar okuma yapıp bunu hedef aygıta aktarmaktadır. Böylece biz örneğin tek hamlede bir dosyayı aşağıdaki gibi ekrana yazdırabiliriz.

```
#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

int main()
{
    ifstream fi("text.txt");

    if (!fi) {
        cerr << "Cannot open file!...\n";
        exit(EXIT_FAILURE);
    }

    cout << fi.rdbuf() << endl;

    return 0;
}
```

Ya da biz aşağıdaki gibi iki dosyayı tek hamlede kopyalayabiliriz:

```
#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

int main()
{
    ifstream fi("text.txt");

    if (!fi) {
        cerr << "Cannot open file!...\n";
        exit(EXIT_FAILURE);
    }
}
```



```

ofstream fo("x.txt");
if (!fo) {
    cerr << "Cannot open file!..\n";
    exit(EXIT_FAILURE);
}

fo << fi.rdbuf() << endl;

return 0;
}

```

iostream Sınıf Sistemiyle stdio Fonksiyonları Arasındaki İşbirliği

Bilindiği gibi stdio kütüphanesinin farklı bir tamponlama biçimi vardır. *stdio* tamponlama biçimi *setvbuf* fonksiyonu ile *tam tamponlamalı (fullbuffered)*, *satır tamponlamalı (line buffered)* ya da *sıfır tamponlamalı (no buffered)* moda geçirilebilmektedir. C standartlarında normal bir dosyanın açıldığında default tamponlama biçimi hakkında bir şey söylenmemiştir. Tabi derleyiciler bunları default olarak tam tamponlamalı (full buffered) modda açmaktadır. Yine C standartlarında *stdin* ve *stdout* dosyalarının işin başında eğer ekran ve klavye gibi karşılıklı etkileşimli bir aygıta yönlendirilmişse, tam tamponlamalı olamayacağı fakat satır tamponlamalı ya da sıfır tamponlamalı olabileceği belirtilmiştir. Bu durum derleyicileri yazarların isteğine bırakılmıştır. *stderr* için ise hangi aygıta yönlendirilmiş olursa olsun, işin başında tam tamponlamalı mod yasaklanmıştır.

C’ de *stdin* dosyasından okuma yapılırken *stdout* dosyasının flush edilmesi gibi bir zorunluluk yoktur. Örneğin,

```

printf("ali");
getchar();

```

Eğer satır tabanlı tamponlama kullanılıyorsa bu yazının çıkması garanti değildir.

C++’ da *basic_ios* sınıfının *tie* isimli üye fonksiyonu ile her tür okuma işlemi öncesinde belirlenen bir *basic_ostream* nesnesinin flush edilmesi sağlanabilmektedir. *cin* nesnesinin başlangıçta default olarak *tie* fonksiyonu ile *cout* nesnesine bağlandığı kabul edilmektedir.

```

basic_ostream<charT,traits>* tie(basic_ostream<charT,traits>* tiestr);

```

Özetle bu durumda biz *cin* üzerinde işlem yapmadan önce her zaman *cout* otomatik olarak flush edilmektedir.

C++’ da *cin* nesnesinin C’ deki *stdin* dosyasını kullandığı, *cout* nesnesinin C’ deki *stdout* dosyasını kullandığı ve *cerr* nesnesinin C’ deki *stderr* dosyasını kullandığı açıkça söylenmiştir. Bunlarda bir senkronizasyon problemi çıkmayacağı da belirtilmiştir. Bu durumda derleyiciler bu senkronizasyonu tipik olarak şöyle sağlayabilirler: *cout* nesnesi tamponlama sınıf nesnesini kullanarak yazma işlemini yapar. Tamponlama nesnesi de transfer işlemine C’ nin *stdout* dosyasının tamponuna yazarak yapmaktadır. Örneğin;

```

printf("ali");
cout << "veli";
fflush(stdout);

```

Burada kesinlikle önce *ali* sonra *veli* yazısı çıkacaktır. Görüldüğü gibi senkronizasyon sağlanmakla birlikte işlem biraz yavaşlatılmıştır. Muhtemelen iki tampon kullanılmaktadır. *cout* yazılacakları kendi tamponlama nesnesinde biriktirmekte, yazılacaklar aygıta aktarılacakken aktarılacaklar aygıt yerine *stdout* tamponuna aktarılmaktadır. Fakat programcı

isterse standart C fonksiyonlarıyla yapılan bu senkronizasyonu kaldırabilir. Bunu yaptığında belki daha hızlı bir çalışma sağlayacaktır ama standart C fonksiyonlarıyla oluşan senkronizasyon bozulacaktır.

```
bool sync_with_stdio(bool sync = true);
```

Bu fonksiyon “false” parametresiyle çağırılırsa C’ deki *stdout* ile senkronizasyon kaldırılır.

Anahtar Notlar:

Buradaki senkronizasyon probleminin kaynağı şudur:

cout nesnesinin stdout dosyasına yazdığı, cin nesnesinin stdin dosyasından okuduğu belirtilmiştir. Fakat stdout ve stdin dosyaları C’ de de vardır. Bu iki dosya aynı dosya mıdır? İşte C’ deki stdout dosyasının C++’ daki stdout dosyasıyla aynı olması mecburan bir senkronizasyon oluşturacaktır. Fakat bunlar farklı ise (ikisi de ekranı temsil ediyor olabilir.) bunların tamponlama sistemleri de farklı olabilir. Aktarım aynı aygıtı aypıldığı halde senkronizasyon bozulabilir.

Manipülâtörler

Manipülâtörler *basic_ostream* ve *basic_istream* sınıflarının operator fonksiyonlarında kullanılan ve formatlama işlerine yarayan elemanlardır. Manipülâtörler parametresiz ve parametrelili olmak üzere ikiye ayrılmaktadır. Parametresiz manipülâtörler aslında birer fonksiyon ismidir. Manipülâtör fonksiyonlarının parametresi duruma göre *basic_ostream*, *ios_base* ya da *basic_istream* türünden referans alır. Örneğin *hex* manipülâtörü aşağıdaki gibi bir fonksiyondur.

```
ios_base& hex (ios_base& str);
```

Bu fonksiyon örneğin şöyle yazılmış olabilir:

```
inline ios_base& hex(ios_base& base)
{
    base.setf(ios_base::hex, ios_base::basefield);
    return base;
}
```

Biz “*cout << hex;*” dediğimizde aşağıdaki operator fonksiyonu çağırılır.

```
basic_ostream<charT,traits>& operator<<
(basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&))
```

Bu fonksiyon içerisinde şu basit işlemler yapılmıştır:

```
pf(*this);
return *this;
```

Özetle, biz “*cout << hex;*” yaptığımızda sırasıyla şunlar olmaktadır.

1. *<<* operator fonksiyonu çağırılmakta bu fonksiyon da *cout*’ u parametre yaparak *hex* fonksiyonunu çağırmaktadır.
2. *hex* fonksiyonu da parametresiyle belirtilen *ios_base* alt nesnesinin *hex* bayrağını set etmektedir.

Parametresiz manipülâtörler standartlarda birer fonksiyon olarak dökümanate edilmiştir. Örneğin, *hex* isminde bir fonksiyonun olduğu ve bunun yukarıda belirtilen işlemi yaptığı söylenmiştir. Fakat başka bir yerde de parametresi fonksiyon göstericisi olan bir *<<* operator

fonksiyonu tanımlanmıştır. Yani *hex*'in bir manipülatör olarak kullanılacağı dolaylı bir sonuç olarak standartlarda dökümanite edilmiştir.

Önemli Parametresiz Manipülatörler

Yukarıda da ele alındığı gibi parametresiz manipülatörler aslında bir parametrelili fonksiyon isimleridir.

- *boolalpha* manipülatörü ve *noboolalpha* manipülatörü *ios_base::boolalpha* flagini set ve reset etmektedir. Bu manipülatör bool değerlerin yazdırılırken true ve false olarak yazdırılmasını sağlar. Default durum *noboolalpha* durumudur. Bu bayrak bir daha set edilene kadar etkisini devam ettirir.
- *showbase* ve *noshowbase* manipülatörleri: 16'lık ve 8'lik sistemdeki sayıları *0x* ya da *0* önekiyle yazdırmak için kullanılırlar. Default durum *noshowbase* biçimindedir.

```
cout << showbase << hex << x << endl;
```

- *showpoint* ve *noshowpoint* manipülatörleri: Normal olarak gerçek sayılar noktadan sonra sıfır görüldüğünde yazılmaları durdurulur, 6.0 biçimindeki sayı da 6 biçiminde yazdırılır. *showpoint* bunun printf fonksiyonunda olduğu gibi bunu 6 basamaklı noktalı bir biçimde yazdırmaktadır.

- *skipws* ve *noskipws* manipülatörleri: Bu manipülatör okuma işleminde kullanılır. *basic_istream* sınıfının >> operatör fonksiyonları default olarak başlangıçtaki boşluk karakterlerini atar, char türden okumada bile bu fonksiyonlar böyle davranmaktadırlar. *noskipws* bu karakterlerin atılmamasını sağlar. Default durum *skipws* biçimindedir. Örneğin;

```
int a;
char s[10];

cin >> noskipws >> a;
if(!cin){
    cerr << "failed\n";
    cin.clear();
}
cin.get(s, 10);

cout << ":" << s;

return 0;
```

- *uppercase* ve *nouppercase* manipülatörleri: Bu manipülatörler hexadecimal yazdırma yaparken harflerin büyük gösterilmesini sağlamaktadır.
- *left*, *right* manipülatörleri: Geniş bir alana yazma yapıldığında hizalama durumunu belirtir, default sağa hizalama yapılmaktadır.
- *dec*, *hex*, *oct* manipülatörleri: Bu manipülatörler tamsayıların hangi tabanda yazdırılacağını ve okunacağını belirtir. Default durum *dec* biçimindedir ve bir daha set işlemi yapılana kadar belirleme geçerli kalır.
- *fixed* ve *scientific* manipülatörleri: Bunlar gerçek sayıların nasıl görüntüleneceğini belirtir. Default durum noktalı yani *fixed* biçimindedir, *scientific* üstel görüntüleme sağlar.

Parametrelili Manipulatorler

Parametrelili manipulatorlerde aslında bir fonksiyondur. Örneğin;

```
cout << setbase(16);
```

Burada *setbase* isimli fonksiyon gerçekten çağırılacaktır, fakat *ios_base* içindeki *hex* bayrağı nasıl set edilecektir? İşte bu fonksiyonun geri dönüş değeri bir sınıf türünden olsa ve kütüphane de o sınıf referans parametrelili bir << operatör fonksiyonu bulursa bu işlem mümkün olabilir. Standartlarda parametrelili manipulatorlerin birer fonksiyon olduğu belirtilmiştir, bunların geri dönüş değerine ilişkin sınıf *smanip* ismiyle set edilmiştir. Yani standartlarda bu manipulatorlere ilişkin sınıfların nasıl olacağı derleyiciyi yazanlara bırakılmıştır, ama *basic_ostream* ve *basic_istream* sınıflarına ilişkin operatör fonksiyonlarının olması gerektiği belirtilmiştir. Tüm parametrelili manipulatorler <iomanip> dosyası içindedirler. Örneğin gcc derleyicilerinde *setbase* manipulatorü için şu işlemler yapılmıştır:

```
struct _Setbase { int M_base; };
inline Setbase
setbase(int base)
{
    Setbase x;
    x.M_base = base;
    return x;
}
ostream& operator<<(ostream &os, Setbase f)
{
    os.setf(f.M_base == 8 ? ios_base::oct :
           f.M_base == 10 ? ios_base::dec :
           f.M_base == 16 ? ios_base::hex :
           ios_base::fmtflags(0), ios_base::basefield);
    return os;
}
```

Görüldüğü gibi *setbase* parametrelili manipulatorü *Setbase* isimli bir yapıya geri dönmekte bu yapıyla işlem yapan bir de << operatör fonksiyonu bulunmaktadır. *setbase* manipulatorü parametreyi yapının içine aktarmakta << operatör fonksiyonu da bunu kullanmaktadır. İşte standartlarda *setbase* gibi manipulatorlerin nasıl bir yapıya ya da sınıfa döneceği açıkça belirtilmemiştir. Bu fonksiyonların bir yapı ya da sınıfla geri dönmesi gerektiği ve bu yapı ya da sınıfla ilişkin bir << operatör fonksiyonunu olması gerektiğinden bahsedilmiştir. Bütün bu bildirimler *iomanip* dosyası içinde yer almalıdır.

Önemli Parametrelili Manipulatorler

- *setbase* tam sayılarda taban belirlemede kullanılır.
- *setprecision* toplam görüntülenecek basamak sayısını belirlemede kullanılır.
- *setw* yazdırılacak bilgi için ne kadar yer ayrılması gerektiğini belirtir. Örneğin;

```
cout << setw(20) << left << x << endl;
```

setw ile yapılan belirleme yalnızca sonraki yazdırmayı etkiler. Örneğin;

```
cout << left << setw(20) << a << setw(20) << b << setw(20) << c
```

- *setfill* manipülatörü bir karakter parametresi alır ve boş alanları bu karakterle doldurur. Default olarak boşluk karakteridir.

Parametrelili ve Parametresiz Manipülatörlerin Yazılması

Programcı isterse kendi sınıfları için çeşitli parametrelili veya parametresiz manipülatörler yazabilir. Parametresiz manipülatörler doğrudan bir fonksiyon olabilir. Bu fonksiyonun parametresi örneğin programcının belirlediği sınıf türünden olabilir. Böylece << operatör fonksiyonunu yazarak onun içerisinde manipülatör fonksiyonu çağırılabilir. Parametrelili manipülatörler için bir ara sınıf kullanılabilir.

C++ ile C++' dan Türetilmiş Yüksek Seviyeli Nesne Yönelimli Programlama Dillerinin Karşılaştırılması

C++ oldukça ayrıntılı profesyonel fakat zor bir programlama dilidir. Son 10 yıldır basit ve kolay öğrenilebilen, kolay ve çabuk ürün geliştirilebilen programlama dilleri ve araçlarının geliştirilmesi yönünde çaba harcanmaktadır. Bu basit dillerden en popüler olanları Java ve C# tır.

C++'ın basitleştirilmesi sırasında bir takım zor öğrenilen konular tamamen atılmış, bazı konular ise basitleştirilmiştir. Örneğin Java dilindeki basitleştirme şöyledir:

- Global fonksiyon kavramı yoktur, tüm fonksiyonlar sınıfın normal ya da static üye fonksiyonu biçiminde olmak zorundadır.
- Gösterici kavramı tamamen kaldırılmıştır.
- Operatör fonksiyonları yoktur.
- Sınıf nesneleri yalnızca heapte new operatörü ile yaratılabilir. Yerel sınıf nesneleri yaratma olanağı yoktur.
- const kavramı kısıtlı ölçüdedir.
- Çoklu türetme yoktur.
- Türetme biçimi kavramı yoktur, Java' daki türetme biçimi C++'ın public türetmesine benzemektedir.
- virtual anahtar sözcüğü yoktur, bütün fonksiyonlar default sanaldır.
- Tür tanımlaması yani typedef kavramı yoktur.
- Ön işlemci kavramı yoktur.
- Exception handling konusu vardır ama C++' a göre daraltılmıştır.
- Tek sınıftan türetilmiş bir şema kullanılmaktadır. Bir sınıf hiçbir sınıftan türetilmemiş olsa bile default *object* sınıfından türetildiği varsayılmaktadır.
- Yapı kavramı yoktur.
- Dil ile bütünleştirilmiş geniş bir sınıf kütüphanesi vardır. Adeta programcının yeni bir sınıf yazmak yerine zaten yazılmış standart sınıfları kullanması istenmektedir.
- Çöp toplayıcı denilen heapte kullanılmayan nesneleri otomatik silen bir kavram vardır.
- Fonksiyon parametrelerinin default değeri alması kavramı yoktur.
- Ön bildirim kavramı yoktur.

C# programlama dili Microsoft tarafından büyük ölçüde Java’ dan esinlenerek tasarlanmıştır. C#, programlama dili olarak Java’nın biraz daha iyileştirilmiş ve C++’ a yaklaştırılmış bir biçimdir. Java’ dan belirgin fazlalıkları şunlardır:

- Operatör fonksiyonları kısıtlı ölçüde de olsa C#’ da vardır.
- Sınıfın üye fonksiyonları default sanal değildir. Sanal yapmak için virtual anahtar sözcüğü kullanılır.
- Yapı kavramı vardır.
- Bileşen tabanlılığı artırmak için derleyici derlediği koda kod hakkında bilgi veren çeşitli meta datalar eklemektedir.
- C#’ da int, long gibi doğal türler de *object* sınıfından türemiş birer yapı olarak kullanılmaktadır.

Java ve C#’ da Gösterici Kavramının Kaldırılış Biçimi

Java ve C#’ da gösterici yoktur ama bir sınıf türünden nesne tanımlandığında C++ dilinde gösterici tanımlanmış gibidir.

Java ve C#	C++
<pre> x a; x = new A(); x.Func(); A y; y = x; </pre>	<pre> A *x; x = new A(); x -> Func(); A *y; y = x; </pre>

Java ve C#’ da stackte nesne tanımlamak diye bir şey yoktur. Sınıf nesneleri yalnızca new ile heapte yaratılabilir.

```

class Sample
{
    static void Main(string[] args)
    {
        A a = new A(10, 20);
        a.Disp();
    }
}

class A
{
    private int m_x;
    private int m_y;

    public A(int x, int y)
    {
        m_x = x;
        m_y = y;
    }

    public void Disp()
    {

```

```

        Console.WriteLine(m_x.ToString() + " " + m_y);
    }
}

```

Çöp Toplayıcı Sistemi

Java ve C# gibi dillerde tüm diziler ve nesneler new operatörü ile heap alanında yaratılmaktadır. Bu nesnelerin silinmesi çöp toplayıcı (garbage collector) denilen bir mekanizma tarafından otomatik yapılmaktadır. Çöp toplayıcı her nesneyi kaç referansın gösterdiğini izler, referans sayacı 0'a düştüğünde otomatik olarak nesneyi siler. Çöp toplayıcı sistemi bu programlama dillerini kolaylaştıran en önemli faktördür. Bu sayede geliştirici binlerce sınıf nesnelerini ya da dizilerini hiç boşaltma kaygısı duymadan yaratabilir. Bu nedenle bu dillerde delete biçiminde bir operatör yoktur.

Arakod Üretimi

Son 10 yıldır çalışabilen kodun taşınabilirliği hakkında çalışılmaktadır. Bu akvrama göre programlama dillerinin derleyicileri doğrudan o sistemdeki micro işlemciye ilişkin bir kod değil de hiçbir işlemciye ilişkin olmayan ve ismine arakod denilen bir makina kodu üretir. Arakod standarttır, doğrudan çalıştırılmayacağına göre bir yorumlayıcı eşliğinde çalıştırılır. Java'da bu yorumlayıcıya Java sanal makinası (Java Virtual Machine), .NET ortamında ise CLR (Common Language Runtime) denilmektedir. Java tarafından üretilen arakoda "Java Byte Code", .NET dillerinin ürettiği arakoda "Microsoft Intermediate Language" denilmektedir. Microsoft'un resmi rakamlarına göre arakodun yorumlanarak çalıştırılması zamanındaki performans kaybı %20 civarındadır. Yorumlayıcılar arakodu fonksiyon fonksiyon olarak gerçek koda dönüştürmekte ve yorumlanmış kodları biriktirmektedir. Böylece bu dillerde derlenmiş olan programlar başka sistemlerde hiçbir şey yapmadan eğer o sistemler için yazılmış bir yorumlayıcı varsa çalıştırılabilir.

Diller Arasındaki Çalışma Uyumluluğu

Microsoft .NET sistemi ile Java'daki fikri geliştirmiştir. .NET sistemi için yazılmış olan derleyicilerin hepsi ortak bir kod üretmektedir. Böylece bir dilde yazılmış olan sınıf hiçbir şey yapmadan başka bir programlama dili içinde çalışabilir. .NET ortamında çalışacak dillerin CLI denen satırdarda uymak zorundadır. Microsoftun bu ortamı için kendisinin derleyicilerini yazdığı 4 dil vardır. Bunlar Managed C++, C#, VB.NET ve J# dilleridir. Microsoft Visual Studio .NET sisteminde Managed ve Unmanaged olmak üzere iki tane C+ derleyicisi vardır. Unmanaged C++ normal C++ tır, Managed C++ ise C++'ın .NET ortamına uyum sağlaması için değiştirilmiş biçimidir.

Bileşen Kavramı ve Bileşen Tabanlı Diller

Bir projede daha önce yazılmış kodların kullanılması ve her defasında aynı kodların yeniden yazılmaması önemlidir. Nesne yönelimli programlama tekniğinde benzer biçimde daha önceden yazılmış sınıfların kullanılması esastır. Yazılımlarda ve projelerde karmaşıklık gitgide artmaktadır. İşte bir projeyi oluşturan sınıflar ya da sınıf sistemleri tamamen bir makina parçası gibi başka bir biçimde temin edilip birleştirilmesi fikri son yıllarda kendini göstermektedir. İşte dilden bağımsız olarak oluşturulmuş olan sınıflara bileşen (component) denilmektedir. Bir programlama dilinde yazılmış olan sınıfın başak bir programlama dilinden kullanılması kolay değildir. Bunu sağlamak için zaman içerisinde çeşitli çözümler önerilmiştir. Bunların en bilineni Microsoft'un COM tekniğidir. COM belirlemelerine uygun

olarak yazılmış olan bir sınıf herhangi bir programlama dilinden kullanılabilir. COM teknolojisinin en önemli dezavantajı kullanım zorluğu ve yazım zorluğudur. Microsoft .NET sistemi ile bu hedefe başka bir biçimde yönelmiştir. .NET ortamında diller ortak arakod ürettiğine göre kaynak kod arakoda dönüştürüldüğünde artık programlama dilinden bağımsız hale gelmektedir.

Geniş Sınıf Kütüphaneleri

Yeni basit nesne yönelimli programlama dillerinin en önemli bileşenlerinden biri geniş bir sınıf kütüphanesine sahip olmasıdır. Sınıf kütüphaneleri çok özel işlemlerin yapılabilmesine olanak sağlayan geniş kütüphanelerdir. Örneğin .NET sınıf sisteminde toplamda yüzlerce sınıf vardır. Her konu için o konuda çalışanların kullanabileceği sınıflar oluşturulmuştur. Bu ortamlarda çalışan geliştiriciler genellikle yeni bir sınıf yazmak zorunda kalmazlar. Java'nın böyle özel bir kütüphanesi vardır. .NET ortamında tüm programlama dilleri aynı kütüphane sistemini kullanmaktadır. Bu ortamlarda kütüphaneleri kullanma becerisi programlama dilini kullanma becerisinden daha önemlidir.

Java ve C#' da Tek Sınıftan Türeme Sistemi

Java' da ve C#' da biz bir sınıftan türetme yapmamış olsak bile bu sınıfların default olarak *object* sınıfından türetildiği varsayılmaktadır. Böylece her sınıf referansı *object* sınıfına dönüştürülebilir.

Bu dillerdeki nesne tutan sınıflar hep *object* referanslarını tutacak biçimdedir. Örneğin biz *Person* nesnelerini bir vektörde tutmaya çalışsak onu sanki *object* referanslarıymış gibi tutarız. Örneğin, bu dillerde nesne tutan sınıfa bir eleman ekleyen fonksiyonunun yapısı aşağıdaki gibidir.

```
void Add(object o);
```

Örneğin *Person* türünden nesne tutan sınıflar şöyle yerleştirilebilir:

```
Person a = new Person("ali");
Person b = new Person("veli");
v.Add(a);
v.Add(b);
```

Görüldüğü gibi buradaki nesne tutan sınıf C++' a göre aslında taban sınıf olan *object* sınıfı türünden göstericileri tutmaktadır. Böyle bir sistemde belirli bir indexteki elemanı alan bir fonksiyonunun yapısı da şöyle olmalıdır:

```
object Get(int index);
```

Görüldüğü gibi nesneler nesne tutan sınıfa *object* türündenmiş gibi yerleştirilmekte ve *object* türündenmiş gibi geri alınmaktadır. Elemanın alınması şöyle yapılmaktadır:

```
Person per;
per = (Person) v.Get(index);
```

C# ve Java' da Doğal Türler

Java' da int, long gibi türler doğal türlerdir ve stackte tutulmaktadır. Bunlar bir sınıf temsil etmediği için bu türden nesnelerin nesne tutan sınıflarda saklanması için ilk harfi büyük harf olan sarma sınıflar düşünülmüştür. Örneğin biz bir grup int değeri bir vektörde aşağıdaki gibi saklarız.

```
for(int i = 0; i < 100; ++i)
    v.Add(new Int(i));
```


C#' da yani genel olarak .NET sınıf sisteminde int, long gibi türler *object* sınıfından türemiş yapılardır. Aslında sistem Java'daki sistemin aynısıdır, bunlar normal olarak stackte saklanır. Fakat *object* türüne dönüştürülecekleri zaman otomatik heapte yer tahsis edilerek sınıf biçimine dönüştürülür. Örneğin;

```
int i;  
object o;  
i = o; //boxing  
k = (int) o; //unboxing
```

$o = i$; işlemi yapıldığında stackte bulunan i 'nin heapte bir kopyası çıkartılır ve artık o referansı bu heapteki kopyayı gösterir.

DK