

Operatörlere Yeni İşlevler Yüklenmesi (Operator Overloading)

C++'da hazır olarak var olan operatörlere (+, -, *, /, !, <<, ++ vs.) ilişkin fonksiyonlar yazarak bu operatörlerin sizin belirlediğiniz işlemleri yapmasını sağlayabilirsiniz. Operatör fonksiyonları bir sınıfın üyesi de olabilirler. Böylece o sınıftan yaratılan nesneler üzerinde işlemler yapan operatörler tanımlanmış olur. C++'da operatör kullanımı fonksiyon çağrılarına karşı düşmektedir. Operatörlere yeni işlevler yükleyerek yapılabilecek her şey normal fonksiyonlar ile de yapılabilir. Fonksiyon isimleri yerine operatörleri kullanmak programın yazılmasını ve okunmasını kolaylaştırabilir. Bu nedenle bir operatöre işlev yüklemek, eğer program daha kolay okunur ve anlaşılır olacaksa tercih edilmelidir.

Sınırlamalar:

- C++'da olmayan operatörlere işlev yüklenemez.
- Örneğin üs alma işlemi için '^' simgesine ya da '**' simgesine bir işlev yüklenemez.
- C++'da var olan operatörlerden bazılarını da yeni işlev yüklenemez. Bunlar;
 - nokta operatörü '.',
 - yaşam alanı belirleme operatörü '::',
 - koşul operatörü '?' ve
 - boyut operatörüdür 'sizeof'.
- C++ operatörleri birli ve ikili olmak üzere iki gruba ayrılabilir.
 - Birli operatörler, tek operand alırlar. Örneğin: -a, a++, !a
 - İkili operatörler ise, iki operand alırlar; a+b, a/b gibi.
- Operatörlere işlev yüklerken operand sayısı değiştirilemez.
- Operatörlerin öncelikleri değiştirilemez.
- Derleyicinin hazır veri tipleri üzerinde işlem yapan operatörlere yeni işlev yüklenemez.
 - Örneğin iki tamsayıyı toplayan + operatörü değiştirilemez.
- Oluşturulan operatörlerin en az bir operandının tipi bir sınıf olmalıdır.

Operatör Fonksiyonlar: Üye ya da Friend Fonksiyon?

Üye fonksiyon mu olacak?

- (), [], -> ya da atama operatörleri aşırı yükleniyorsa, mutlaka üye fonksiyon kullanılmalı.
- Üye fonksiyon kullanılacaksa, Operatörün sol tarafı mutlaka kendi sınıfından bir nesne olmalıdır. Eğer sol taraf kendi sınıfından bir nesne değilse, operatör fonksiyonu kesinlikle üye olamaz.
- Üye fonksiyon olmayacaksa! Eğer public yada private üyelere erişilmesi gerekiyorsa mutlaka friend fonksiyon olmalıdır. Operandlarına ulaşma hakkı olmalıdır.

Aşırı-yüklenemeyen Operatörler							
+	-	*	/	%	^	&	
~	!	=	<	>	+=	-=	*=
/=	%=	*=	&=	=	<<=	>>=	>>=
<<=	+=	-=	<=	>=	&&		++
--	->*	+	->	[]	()	new	delete
new[]	delete[]						

Aşırı-yüklenemeyen Operatörler				
.	*	::	?:	sizeof

Bu işlemlere başlamadan önce kopyalama kurucusu nedir, onu tanıyalım;

Kopyalama Kurucusu (Copy Constructor)

Kopyalama kurucusu başkibir çeşit kurucu fonksiyondur. Tek farkı kendinden önce yaratılmış nesnenin değerleri ile yeni nesneyi sınıftan oluşturmamızı sağlar. Bir nevi klonlama işlemidir. Her zaman olduğu gibi bir sınıf içerisinde tanımlanan fonksiyonların aşırı yüklenmesi durumundaki olay buradada geçerlidir. Nesneyi oluşturma biçiminize göre derleyici bu işi otomatik olarak kendi belirler. Alt kısımda buna örnek bir program ve ekran çıktısı görülmektedir.

Örnek 1:

```
#include <iostream>
// Bu örnek içerisinde copy constructor kullanımı gösterilmiştir...
using namespace std;
class deneme
{
private:
    int x,y;
public:
    deneme(deneme &); //kopyalama kurucusu (copy constructor)
    deneme(); // parametresiz kurucu fonksiyon (constructor)
    deneme(int ,int); // parametrelili kurucu fonksiyon
    ~deneme();// yıkıcı(destructor)
};

deneme::deneme(int a,int b)
{
    cout<<"Parametrelili kurucu fonksiyon calisti.."<<endl;
    x=a;
    y=b;
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
}

deneme::deneme ()
{
    cout<<"Kurucu fonksiyon calisti.."<<endl;
    x=5;
    y=10;
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
}

deneme::deneme(deneme &alias)
{

    cout<<"Kopyalama kurucusu calisti"<<endl;
    x=alias.x;
    y=alias.y;
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
}

deneme::~deneme ()
{
    cout<<"Yıkici fonksiyon calisti..";
}

int main()
{
```

```

cout<<"d diye bir nesne yaratacagim ve c de onun kopyasi olacak\n";
deneme d,c(d); // c nesnesi d nin deęerleri ile doęuyor...
cout<<"simdi e yi c nesnesini kullanarak kopyalıyoruz...\n";
deneme e(c); // e nesnesi kopyalanmıř c den doędu.
cout<<"alt kisimda f nesnesini 3 ve 4 ilk deęerleri ile yaratıyoruz..\n";
deneme f(3,4); // parametre alarak yaratılan f nesnesi
cout<<"g nesnesi ise f den kopyalanıyor, klonlanıyor...\n";
deneme g(f); // ve görüldüęü gibi bunun kopyası olan g...

// dikkat ! nesnenin yařam alanı biterse yıkıcı fonksiyon çalışacak ve
// nesneyi hafızadan silecektir, kopyalama kurucusunu kullanırken buna
// dikkat etmeyi unutmayınız...

system("pause");
return 0;
}

```

PROGRAMIN ÇIKTISI

```

-----
d diye bir nesne yaratacagim ve c de onun kopyasi
olacak
Kurucu fonksiyon calisti..
x=5
y=10
Kopyalama kurucusu calisti
x=5
y=10
simdi e yi c nesnesini kullanarak kopyalıyoruz...
Kopyalama kurucusu calisti
x=5
y=10
alt kisimda f nesnesini 3 ve 4 ilk deęerleri ile
yaratıyoruz..
Parametrelili kurucu fonksiyon calisti..
x=3
y=4
g nesnesi ise f den kopyalanıyor, klonlanıyor...
Kopyalama kurucusu calisti
x=3
y=4
Press any key to continue . . .

```

Bir Operandlı (Unary) Operatörlere Yeni Bir İşlev Yüklenmesi

Birli operatörlere örnekler:

- arttırma (++),
- azaltma(--),
- eksileme(-5),
- mantıksal tümlleme (!) vs.

Bu operatörlere ilişkin fonksiyonlar bir sınıfın üyesi olarak yazıldıklarında hiç parametre almazlar. Çünkü üzerinde çağırıldıkları nesne üzerinde işlem yaparlar. Bu operatörler normalde nesnenin solunda yer alırlar:

- !n
- -n
- ++n gibi.

Örnek 2: Bu program ön ek prefix olarak değeri arttırır fakat fonksiyon void olduğu için geriye değeri döndüremez.

```
// countpp1.cpp
// ++ operatörü ile değeri arttırma. Ör: ++sayac
#include <iostream>
using namespace std;
////////////////////////////////////
class Counter
{
private:
    unsigned int count;           //count
public:
    Counter() : count(0)          //constructor
    { }
    unsigned int get_count()       //return count
    { return count; }
    void operator ++ ()            //arttır ön ek (prefix)
    {
        ++count;
    }
};
////////////////////////////////////
int main()
{
    Counter c1, c2;                //define and initialize

    cout << "\nc1=" << c1.get_count(); //göster
    cout << "\nc2=" << c2.get_count();

    ++c1;                          //increment c1
    ++c2;                          //increment c2
    ++c2;                          //increment c2

    cout << "\nc1=" << c1.get_count(); //tekrar göster
    cout << "\nc2=" << c2.get_count() << endl;
    system("pause");
    return 0;
}
```

Ekran Çıktısı

```
c1=0
c2=0
c1=1
c2=2
```

Yukarıdaki örneğin bir dezavantajı c2=++c1 şeklinde bir atama işlemi yapılamamasıdır. Şimdi bunu nasıl çözeceğimize bakalım.

Örnek 3: ++ operatorü ile değeri arttırma ve bu değeri geri döndürme

```
#include <iostream>
using namespace std;
class Counter
{
private:
    unsigned int count;           //count
public:
    Counter() : count(0)          //constructor
```

```

    { }
    unsigned int get_count() //return count
    { return count; }

    Counter operator ++ () //increment count
    {
        ++count; //increment count
        Counter temp; //make a temporary Counter
        temp.count = count; //give it same value as this obj
        return temp; //return the copy
    }
};

int main()
{
    Counter c1, c2; //c1=0, c2=0

    cout << "\nc1=" << c1.get_count(); //display
    cout << "\nc2=" << c2.get_count();

    ++c1; //c1=1
    c2 = ++c1; //c1=2, c2=2 Artık bu tarz işlem OK!

    cout << "\nc1=" << c1.get_count(); //display again
    cout << "\nc2=" << c2.get_count() << endl;
    return 0;
}

```

Ekran Çıktısı

```

c1=0
c2=0
c1=2
c2=2

```

Yukarıda bu sorunu çözmüş bulunuyoruz.

Örnek 4: Bu işlemi kökten çözmenin yöntemi...

```

// countpp3.cpp
// counter değişkenini ++ operatorü ile arttırma, değer aktarma ve
// isimsiz geçici objeler kullanma... Burada ek olarak birde değer alan kurucu
// koyduk dikkat!
#include <iostream>
using namespace std;
////////////////////////////////////
class Counter
{
private:
    unsigned int count; //count değişkeni
public:
    Counter() : count(0) //constructor değer almayan {count=0;} ile aynı
    { }
    Counter(int c) //constructor, bir değer alan
    { count=c; }
    int get_count() //return count
    { return count; }
    Counter operator ++ () //count değerini arttır
    {
        ++count; // count değerini arttır ve
        return Counter(count); // geçici bir tampon obje döndür
    } // ve objeyi tek değer alan kurucu ile kur
}

```

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

```
int main()
{
    Counter c1, c2;                                //c1=0, c2=0

    cout << "\nc1=" << c1.get_count();             //göster
    cout << "\nc2=" << c2.get_count();

    ++c1;                                           //c1=1
    c2 = ++c1;                                     //c1=2, c2=2

    cout << "\nc1=" << c1.get_count();             //tekrar göster
    cout << "\nc2=" << c2.get_count() << endl;
    return 0;
}
```

Ekrana Çıktısı

```
c1=0
c2=0
c1=2
c2=2
```

++ Operatörü İçin Postfix ve Prefix (++ i , i ++)

// overloaded ++ operator ön ek ve son ek olarak ikisininide yapar, değeri //döndürür. Eşitleme işlemi yapılabilir. Ör: c2=++c1; veya c2=c1++; gibi...

```
#include <iostream>
```

```
using namespace std;
```

```
class Counter
```

```
{
private:
    unsigned int count;                //count
public:
    Counter() : count(0)               //constructor no args
    { }
    Counter(int c) : count(c)          //constructor, one arg
    { }
    unsigned int get_count() const     //return count
    { return count; }
}
```

```
Counter operator ++ ()                //count değerini arttır (prefix)
```

```
{
    //
    return Counter(++count);          //isimsiz geçici obje döndür
}                                     //ve count değerini initialize et
```

```
Counter operator ++ (int)             //(postfix)
```

```
{
    //isimsiz geçici obje döndür
    return Counter(count++);          //bu count değeri ile objeyi initialize et
}                                     //count değerini arttır
```

```
};
int main()
{
    Counter c1, c2;                                //c1=0, c2=0
    cout << "\nc1=" << c1.get_count();             //display
    cout << "\nc2=" << c2.get_count();
    ++c1;                                           //c1=1
    c2 = ++c1;                                     //c1=2, c2=2 (prefix)
```

Ekrana Çıktısı

```
c1=0
c2=0
c1=2
c2=2
c1=3
c2=2
```

```
cout << "\nc1=" << c1.get_count();    //display
cout << "\nc2=" << c2.get_count();
c2 = c1++;                             //c1=3, c2=2 (postfix)
cout << "\nc1=" << c1.get_count();    //display again
cout << "\nc2=" << c2.get_count() << endl;
return 0;    }
```

Örnek 5: Aşağıdaki örnekte ++ operatörüne, karmaşık sayıların reel kısmını 0.1 arttırma işlevi yüklenmiştir.

```
.
void ComplexT::operator++()
{
    re=re+0.1;
}
int main()
{
    ComplexT z(1.2, 0.5);
    ++z; // operator++ fonksiyonu canlanır
    z.goster();
    return 0;
}
```

Örnek 6: Eğer bu operatörün bir atama deyiminde kullanılması isteniyorsa operatöre ilişkin fonksiyon nesnenin referansını geri döndürülmelidir. (Örnek 4deki kullanıma, ek olarak böylede yapılabilir.)

```
// ++ operatörüne yeni bir işlev yüklenmesi
// Operatör karmaşık sayıların reel kısmını 0.1 kadar arttırmaktadır.

#include <iostream>
using namespace std;

// Karmaşık (Kompleks) sayıları tanımlamak için oluşturulan sınıf
class ComplexT{
    double re,im;                // reel ve sanal kısımlar
public:
    ComplexT(double re_in=0,double im_in=1): re(re_in),im(im_in)
    {};                          // Kurucu (gövdesi boş)
    const ComplexT& operator++( );    // ++ Operatörü
    void goster() const;
};

// ++ operatörü
// Karmaşık sayıların reel kısmını 0.1 kadar arttırmaktadır.
const ComplexT & ComplexT::operator++()
{
    re=re+0.1;                  // reel kısım arttırılıyor
    return *this;               // Nesnenin referansı döndürülüyor
}

// Nesne ile ilgili bilgileri ekrana çıkaran metot
void ComplexT::goster() const
{
    cout << "re=" << re << " im=" << im << endl;
}
```

Ekran Çıktısı:

```
re=1.2 im=0.5
re=1.3 im=0.5
re=1.3 im=0.5
```

```
//---- Ana Program ----
int main()
{
    ComplexT z1(1.2,0.5),z2;    z1.goster();    z2= ++z1;    z1.goster();
    z2.goster();    return 0;
}
```

Bilindiği gibi ++ ve -- operatörleri operandların hem soluna hem de sağına yazılabilirler. Bir atama deyimi ile birlikte kullanıldıklarında operatörlerin yazılma şekli önemli olur. Buna göre önceden arttırma (azaltma) ya da sonradan arttırma (azaltma) anlamına gelirler.

```
z2= ++ z1; // önceden arttırma
z2 = z1++; // sonradan arttırma
z2= -- z1; // önceden azaltma
z2 = z1--; // sonradan azaltma
```

Operatör fonksiyonu operator++() şeklinde parametresiz olarak yazılırsa, önceden arttırma operatörüne bir işlev yüklenmiş olur. Sonradan arttırma operatörüne bir işlev yüklemek için fonksiyon bir parametrelili olarak yazılır operator++(int). Buradaki parametrenin amacı sadece iki fonksiyonu birbirinden ayırmaktır. Fonksiyon çağrılırken herhangi bir parametre gönderilmez.

Örnek 7 : Karmaşık sayılar hesabı.

```
// ++ operatörünün hem önceden hem de sonradan arttırma için kullanılması
// Operatör karmaşık sayıların reel kısmını 0.1 kadar arttırmaktadır.

#include <iostream>
using namespace std;

// Karmaşık (Kompleks) sayıları tanımlamak için oluşturulan sınıf
class ComplexT{
    double re,im;    // reel ve sanal kısımlar
public:
    ComplexT(double re_in=0,double im_in=1): re(re_in),im(im_in)
    {};    // Kurucu (gövdesi boş)
    const ComplexT& operator++();    // önceden arttırma ++ operatörü
    ComplexT operator++(int);    // sonradan arttırma ++ operatörü
    void goster() const;
};

// Önceden arttırma ++ operatörü
// Karmaşık sayıların reel kısmını 0.1 kadar arttırmaktadır.
const ComplexT & ComplexT::operator++()
{
    re=re+0.1;    // reel kısım arttırılıyor
    return *this;    // Nesnenin referansı döndürülüyor
}

// Sonradan arttırma ++ operatörü
// Karmaşık sayıların reel kısmını 0.1 kadar arttırmaktadır.
ComplexT ComplexT::operator++(int)    // sonradan arttırma operatörü
{
    ComplexT gecici;
    gecici = *this;    // nesnenin orijinal değeri
    re=re+0.1;    // reel kısım arttırılıyor
    return gecici;    // eski (artmamış) değer döndürülüyor
}
```

```
// Nesne ile ilgili bilgileri ekrana çıkaran metot
void ComplexT::goster() const
{
    cout << "re=" << re << " im=" << im << endl;
}

//---- Ana Program ----
int main()
{
    ComplexT z1(1.2,0.5),z2;
    z2= ++z1;          // operator ++() çalışır
    z1.goster();
    z2.goster();
    z2= z1++;          // operator ++(int) çalışır
    z1.goster();
    z2.goster();
    return 0;
}
```

Ekran Çıktısı:

```
re=1.2 im=0.5
re=1.3 im=0.5
re=1.3 im=0.5
re=1.4 im=0.5
re=1.3 im=0.5
```

+ Operatörüne Yeni Bir İşlev Yüklenmesi

Örnek 8: Bu örnekte basit olarak iki adet tamsayı değerli objeler olacak ve biz bunları sanki bir tamsayı topluyormuş gibi + operatörü ile toplatacağız..

// **+ operatörüne yeni bir işlev yüklenmesi**. Kopyalama kurucusu da kullanıldı...

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

```
class deneme
{
private:
    int x,y;
public:
    deneme(deneme &); //kopyalama kurucusu (copy constructor)
    deneme(); // parametresiz kurucu fonksiyon (constructor)
    deneme(int ,int); // parametrelili kurucu fonksiyon
    ~deneme(); // yıkıcı (destructor)
    deneme& deneme::operator+(deneme&); // + operatörü için overloading
};
```

// bu sayede nesneleri sanki sayıymış gibi toplayabileceğiz...

```
deneme& deneme::operator+(deneme& s)
{
    cout<<" + operatoru calisti"<<endl;
    int xx,yy;
    xx=x+s.x;
    yy=y+s.y;
    cout<<" xx= "<<xx<<" yy= "<<yy<<endl;
}
```

```
deneme::deneme(int a,int b)
{
```

```

        cout<<"Parametreli kurucu fonksiyon calisti.."<<endl;
        x=a;
        y=b;
        cout<<"x="<<x<<endl;
        cout<<"y="<<y<<endl;
    }

    deneme::deneme()
    {
        cout<<"Kurucu fonksiyon calisti.."<<endl;
        x=5;
        y=10;
        cout<<"x="<<x<<endl;
        cout<<"y="<<y<<endl;
    }
    deneme::deneme(deneme &alias)
    {

        cout<<"Kopyalama kurucusu calisti"<<endl;
        x=alias.x;
        y=alias.y;
        cout<<"x="<<x<<endl;
        cout<<"y="<<y<<endl;
    }
    deneme::~deneme()
    {
        cout<<"Yikici fonksiyon calisti..";
    }

    int main()
    {
        deneme d,c(3,2),e; // c nesnesi d nin deęerleri ile doęuyor...

        e=d+c;

        // aslında alt kısım ile ust kısım aynı işi yapıyor...

        e=d.operator+(c);

        deneme x; // yeni bir nesne oluşturalım
        // eęer sınıf içerisinde pointer yoksa; eşittir işlemi için
        // ek olarak operator overload etmenize gerek yoktur...
        d=c; // gördüğünüz gibi bu işlem nesneleri birbirine sorunsuzca eşitliyor.
        x=d+c;
        system("pause");
        return 0;
    }

```

EKRAN ÇIKTISI

```

Kurucu fonksiyon calisti..
x=5
y=10
Parametreli kurucu fonksiyon
calisti..
x=3
y=2
Kurucu fonksiyon calisti..
x=5
y=10
+ operatoru calisti
xx= 8 yy= 12
+ operatoru calisti
xx= 8 yy= 12
Kurucu fonksiyon calisti..
x=5
y=10
+ operatoru calisti
xx= 6 yy= 4

```

Örnek 9:

```

// + operatörüne yeni bir işlev yüklenmesi
#include <iostream>
using namespace std;

// Karmaşık (Kompleks) sayıları tanımlamak için oluşturulan sınıf
class ComplexT{
    double re,im; // reel ve sanal kısımlar
public:
    ComplexT(double re_in=0,double im_in=1); // Kurucu
    ComplexT operator+(const ComplexT & ) const; // + operatörünün fonksiyonu
    void goster() const;
};

```

```
//Parametrsiz çağırılabilen kurucu. Çünkü ilk değerleri var.
ComplexT::ComplexT(double re_in,double im_in)
{
    re=re_in;
    im=im_in;
    cout<< endl << "Kurucu calisti. ";
}

// + operatörü
ComplexT ComplexT::operator+(const ComplexT &c) const
{
    double yeni_re, yeni_im;
    yeni_re=re+c.re;
    yeni_im=im+c.im;
    return ComplexT(yeni_re,yeni_im);
}

// Nesne ile ilgili bilgileri ekrana çıkaran metot
void ComplexT::goster() const
{
    cout << endl << "re=" << re << " im=" << im;
}

int main()
{
    ComplexT z1(1,1),z2(2,2),z3;
    z3=z1+z2; // z3=z1.operator+(z2); yapmakla aynıdır
    z3.goster();
    return 0;
}
```

Ekrana Çıktısı:

```
Kurucu calisti.
Kurucu calisti.
Kurucu calisti.
Kurucu calisti.
re=3 im=3
```

Yukarıdaki örnekte ComplexT sınıfına + operatörü için bir metot eklenmiştir. Böylece + operatörünün karmaşık sayıları toplaması sağlanmıştır.

Atama Operatörüne “=” Yeni Bir İşlev Yüklenmesi

```
void ComplexT::operator=(const ComplexT& z) // Gereksiz
{
    re = z.re;
    im = z.im;
}
```

Bu fonksiyon yazılmasaydı derleyicinin yerleştireceği fonksiyon da aynı işi yapardı. Eğer sınıfta bir atama fonksiyonu varsa, artık derleyici tüm atama işleri için bu fonksiyonu kullanır.

Örnek 10:

```
// Atama operatörüne yeni bir işlev yüklenmesi
// Karmaşık sayılar sınıfı ComplexT'ye = operatörü için bir metot eklenmiştir.
// Bu sınıf için bu metodun yazılması gereksizdir.
```

```
// Bu fonksiyon sınıftan kaldırılrsa da ana program doğru olarak çalışır.
// Bu program sadece atama fonksiyonunun yazılımını göstermek için hazırlanmıştır.
```

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

// Karmaşık (Kompleks) sayıları tanımlamak için oluşturulan sınıf
class ComplexT{
    double re,im; // reel ve sanal kısımlar
public:
    ComplexT(double re_in=0,double im_in=1): re(re_in),im(im_in)
    {}; // Kurucu (gövdesi boş)
    void operator=(const ComplexT & ); // = Operatörü
    void goster() const;
};

// = operatörü
void ComplexT::operator=(const ComplexT &c)
{
    re = c.re; // Atamalar yapılıyor
    im = c.im;
    cout << "Atama fonksiyonu calisti" << endl; // Çalıştığını ekranda görmek için
}

// Nesne ile ilgili bilgileri ekrana çıkaran metot
void ComplexT::goster() const
{
    cout << "re=" << re << " im=" << im << endl;
}

int main()
{
    ComplexT z1(1,2),z2;
    z2.goster(); // Atamadan önceki içerik
    z2=z1;
    z2.goster(); // Atamadan sonraki içerik
    return 0;
}
```

Ekran Çıktısı:

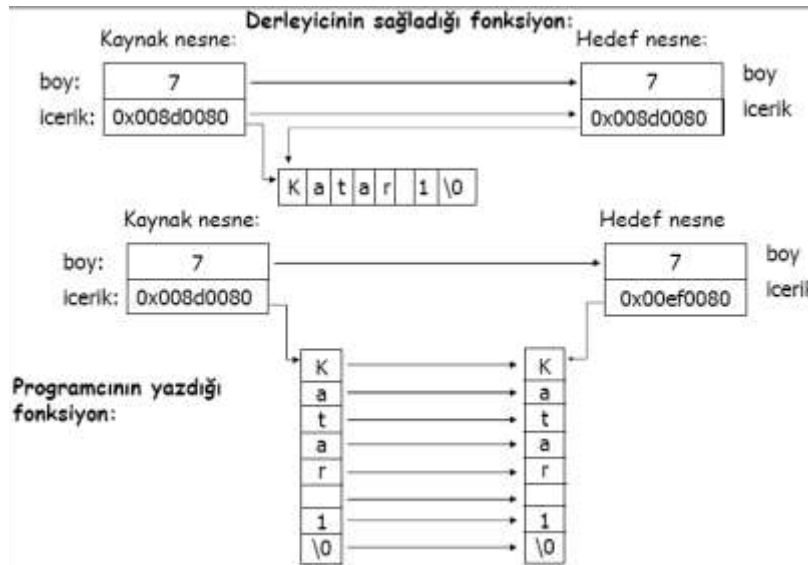
```
re=0 im=1
Atama fonksiyonu calisti
re=1 im=2
```

Atama işlemi programlarda çok sık kullanıldığından C++ derleyicisi her sınıfa atama operatörü için bir fonksiyon yerleştirir. Derleyicinin yerleştirdiği fonksiyon bir nesnenin verilerini var olan başka bir nesnenin veri alanlarına bire bir kopyalar. Eğer bu bire bir atama işlemi o sınıf için yeterli ise programcının atama operatörü için bir fonksiyon yazmasına gerek kalmaz. İçinde işaretçi olmayan sınıflar için genellikle derleyicinin sağladığı fonksiyon yeterlidir. Örneğin karmaşık sayılar için yukarda gösterilen atama fonksiyonu gereksizdir.

Ancak her sınıf için derleyicinin sağladığı atama fonksiyonu yeterli olmayabilir. Özellikle içinde işaretçi olan sınıflarda programcının atama operatörüne ilişkin fonksiyonu yazması gerekebilir. Kopyalama kurucusundakine benzer bir problem burada da vardır. Aşağıda örnek String sınıfı için atama fonksiyonu yazılmıştır.

```
class String{    // Örnek (karakter katarı) String sınıfı
    int boy;    // Katarın boyu
    char *icerik; // Katarın içeriği
public:
    String();    // Parametresiz kurucu
    String(const char *);    // Kurucu
    String(const String &);    // Kopyalama kurucusu
    void operator=(const String &); // Atama operatörü
    void goster();    // Katarları ekrana çıkaran üye fonksiyon
    ~String();    // Yok edici fonksiyon
};
```

```
// Atama Operatörü
void String::operator=(const String &gelen_nesne)
{
    cout<< "Atama operatoru calisti" << endl;
    boy = gelen_nesne.boy;
    delete [] icerik;    // Eski içerik belleğe iade ediliyor
    icerik = new char[boy + 1];    // +1 null karakteri için
    strcpy(icerik, gelen_nesne.icerik);
}
```



Bir önceki örnekte gösterilen atama fonksiyonu geriye bir değer döndürmemektedir (`void`). Bu nedenle bu operatörü kaskad olarak yazmak (`a = b = c` gibi) mümkün değildir. Operatörleri kaskad bağlayabilmek için operatöre ilişkin fonksiyon, üzerinde işlem yapılan nesnenin referansını geri

göndermelidir. Eğer operatör fonksiyonlarının geri dönüş değeri tipleri void olarak yazılırsa bu operatörler peş peşe bağlanamaz.

```
// Atama Operatörü
const String& String::operator=(const String &gelen_nesne)
{
    cout<< "Atama operatoru calisti" << endl;
    boy = gelen_nesne.boy;
    delete [] icerik; // Eski icerik belleğe iade ediliyor
    icerik = new char[boy + 1]; // +1 null karakteri için
    strcpy(icerik, gelen_nesne.icerik);
    return *this; // Kendi adresini geri gönderiyor
}
```

Atama operatörü ile kopyalama kurucusu benzer işler yapmakla beraber farklı zamanlarda canlanırlar. Kopyalama kurucusu yeni bir nesne yarılırken canlanır ve eski bir nesnedeki bilgileri yeni nesneye kopyalar. Atama operatörü ise var olan bir nesneye başka bir nesnedeki değerler atanırken canlanır.

Örnek 11: // Atama operatörünün "=" Katar dizileri (String) için yüklenmesi

```
#include <iostream>
#include <cstring> // string fonksiyonları için
using namespace std;

class String{ // Örnek (karakter katarı) String sınıfı
    int boy; // Katarın boyu
    char *icerik; // Katarın içeriği
public:
    String(); // Parametresiz kurucu
    String(const char *); // Kurucu
    String(const String &); // Kopyalama kurucusu
    const String& operator=(const String &); // Atama operatörü
    void goster(); // Katarları ekrana çıkaran üye fonksiyon
    ~String(); // Yok edici fonksiyon
};

// Parametresiz Kurucu Fonksiyon
// Sadece NULL içeren bir boş katar oluşturur
String::String()
{
    cout<< "Parametresiz Kurucu calisti" << endl;
    boy = 0; // boş katarın boyu sıfır
    icerik = new char[1]; // icerik için yer ayrıldı, null için
    strcpy(icerik, ""); // boş katar
}

// Kurucu Fonksiyon
```

```
// Parametre olarak aldığı katarı nesnenin içeriğine kopyalar
String::String(const char *gelen_veri)
{
    cout<< "Kurucu calisti" << endl;
    boy = strlen(gelen_veri); // gelen katarın boyu hesaplandı
    icerik = new char[boy + 1]; // icerik için yer ayrıldı, +1 null için
    strcpy(icerik, gelen_veri); // gelen veri icerik'in gosterdigi yere kopyalanıyor
}

// Kopyalama kurucusu
String::String(const String &gelen_nesne)
{
    cout<< "Kopyalama Kurucusu calisti" << endl;
    boy = gelen_nesne.boy;
    icerik = new char[boy + 1]; // +1 null karakteri için
    strcpy(icerik, gelen_nesne.icerik);
}

// Atama Operatörü
const String& String::operator=(const String &gelen_nesne)
{
    cout<< "Atama operatoru calisti" << endl;
    boy = gelen_nesne.boy;
    delete [] icerik; // Eski icerik belleğe iade ediliyor
    icerik = new char[boy + 1]; // +1 null karakteri için
    strcpy(icerik, gelen_nesne.icerik);
    return *this; // Kendi adresini geri gönderiyor
}

void String::goster()
{
    cout<< icerik << ", " << boy << endl; // Katar ve boyu ekrana yazılıyor
}

// Yok edici Fonksiyon
// icerik tarafından isaret edilen bellek geri veriliyor
String::~~String()
{
    cout<< "Yok edici calisti" << endl;
    delete [] icerik;
}

//----- Ana Program -----
int main() // Ana fonksiyon
{
    String s1("Katar 1");
    String s2 = s1; // Kopyalama kurucusu çalışır
    s2.goster();
    String s3 , s4;
    s3 = s4= s2; // Atama operatörü 2 defa çalışır
    s3.goster();
    s4.goster();
    return 0;
}
```

Ekran Çıktısı

```
Kurucu calisti
Kopyalama Kurucusu calisti
Katar 1, 7
Parametresiz Kurucu calisti
Parametresiz Kurucu calisti
Atama operatoru calisti
Atama operatoru calisti
Katar 1, 7
Katar 1, 7
```

```
}
```

İndis Operatörüne (Subscript Operator) “[]” Yeni Bir İşlev Yüklenmesi

Tüm operatör fonksiyonları için aynı kurallar geçerli olduğundan hepsini ayrı ayrı anlatmaya gerek yoktur. Ancak ilginç ve yararlı olabilecek bazı operatörler açıklanmıştır. Bunlardan biri de indis operatörüdür. Bu operatöre ilişkin fonksiyon iki farklı yapıda olabilir:

```
class C{
dönüş tipi & operator [] (parametre tipi);
ya da
const dönüş tipi & operator [] (parametre tipi) const; };
```

Birinci yazım şekli, eğer bu operatör ile nesnenin verileri değiştirilecekse kullanılır. Bu durumda operatör bir atama operatörünün solunda yer alabilir. İkinci yazım şeklinde ise fonksiyon sabit olarak tanımlanmıştır. Bu durumda operatör ile nesnenin verileri sadece okunabilir. c bir nesne olmak üzere c[i] ifadesi c.operator[](i) anlamına gelir.

Örnek 12:

İndis operatörüne String sınıfında kullanılmak üzere bir işlev yüklenecektir. Bu operatör bir katardaki i. karaktere erişilmesini sağlayacaktır. Eğer i sıfırdan küçük verilirse katardaki ilk elemana, eğer i katarın boyundan büyük verilirse katardaki son elemana erişilmiş olacaktır.

```
// İndis operatörünün "[ ]" Katar dizileri (String) için yüklenmesi

#include <iostream>
#include <cstring> // string fonksiyonları için
using namespace std;

class String{ // Örnek (karakter katarı) String sınıfı
    int boy; // Katarın boyu
    char *icerik; // Katarın içeriği
public:
    String(const char *); // Kurucu
    char & operator[](int i); // İndis operatörü
    void goster(); // Katarları ekrana çıkaran üye fonksiyon
    ~String(); // Yok edici fonksiyon
};

// Kurucu Fonksiyon
// Parametre olarak aldığı katarı nesnenin içeriğine kopyalar
String::String(const char *gelen_veri)
{
    cout<< "Kurucu calisti" << endl;
    boy = strlen(gelen_veri); // gelen katarın boyu hesaplandı
    icerik = new char[boy +1]; // icerik için yer ayrıldı, +1 null için
    strcpy(icerik, gelen_veri); // gelen veri icerik'in gosterdigi yere kopyalanıyor
}

// İndis Operatörü
char& String::operator[](int i)
{
    cout << "Indis operatoru calisti" << endl;
    if(i < 0)
        return icerik[0]; // İlk eleman gönderiliyor
    if(i >= boy)
        return icerik[boy-1]; // Son eleman gönderiliyor
    return icerik[i]; // i. eleman gönderiliyor
}
```

```

}

void String::goster()
{
    cout<< icerik << ", " << boy << endl; // Katar ve boyu ekrana yazılıyor
}

// Yok edici Fonksiyon
// icerik tarafından isaret edilen bellek geri veriliyor
String::~String()
{
    cout<< "Yok edici calisti" << endl;
    delete[] icerik;
}

//----- Ana Program -----
int main() // Ana fonksiyon
{
    String s1("Katar 1");
    s1[1] = 'X'; // Katarın 2. karakteri 'X' oldu
    s1.goster();
    cout << "Katarin 3 numarali elemani = " << s1[3] << endl;
    return 0;
}

```

Ekrana Çıktısı

```

Kurucu calisti
Indis operatoru calisti
KXtar 1, 7
Indis operatoru calisti
Katarin 3 numarali elemani = a

```

Fonksiyon Çağırma (Function Call) Operatörüne “()” Yeni Bir İşlev Yüklenmesi

Bu operatörü diğerlerinden ayıran özellik, operand sayısının programcı tarafından belirlenebilmesidir. Bu operatöre ilişkin fonksiyon aşağıdaki yapıda olur:

```

class C
{
    dönüş tipi operator () (parametre tipleri);
};

```

c bir nesne olmak üzere **c(i,j,k)** ifadesi **c.operator()(i,j,k)** anlamına gelir.

Örnek 13: Aşağıdaki örnekte fonksiyon çağırma operatörüne, karmaşık sayıları ekrana çıkarma işlevi yüklenmiştir. Bu örnekte operatör parametre almamaktadır.

```

// () operatörüne yeni bir işlev yüklenmesi
// Bu örnekte () operatörü karmaşık sayıları ekrana çıkarmak
// için kullanılmaktadır. Operatörün fonksiyonu parametre almamaktadır.
#include <iostream>
using namespace std;

// Karmaşık (Kompleks) sayıları tanımlamak için oluşturulan sınıf
class ComplexT{
    double re,im; // reel ve sanal kısımlar
public:
    ComplexT(double re_in=0,double im_in=0):re(re_in),im(im_in)
    {}; // Kurucu, gövdesi boş
    ComplexT operator+(const ComplexT & ) const; // + operatörünün fonksiyonu
    void operator()() const; // () operatörünün fonksiyonu
};

```

```
// + operatörü
ComplexT ComplexT::operator+(const ComplexT &c) const
{
    double yeni_re, yeni_im;
    yeni_re=re+c.re;
    yeni_im=im+c.im;
    return ComplexT(yeni_re,yeni_im);
}
```

```
// () operatörü
// Nesne ile ilgili bilgileri ekrana yazar
// Parametresiz fonksiyon çağırma operatörü, karmaşık sayıları ekrana yazar
void ComplexT::operator()() const
{
    cout << "re=" << re << " im=" << im << endl ;
}
```

```
int main()
{
    ComplexT z1(1,1),z2(2,2),z3;
    z1();
    z2();
    z3=z1+z2;
    z3();
    return 0;
}
```

Ekran Çıktısı <pre>re=1 im=1 re=2 im=2 re=3 im=3</pre>
--

Örnek 14: Bu örnekte ise fonksiyon çağırma operatörünün parametrelili olarak kullanımı gösterilmiştir.

Örnekteki operatör, bir karakter katarının (String) istenen bir miktarını, istenen bir bellek bölgesine kopyalamaktadır. 2 adet parametre almaktadır.

```
// Fonksiyon çağırma operatörünün "()" Katar dizileri (String) için yüklenmesi
// Operatör, katarın bir kısmını istenen bir bellek bölgesine kopyalar

#include <iostream>
#include <cstring> // string fonksiyonları için
using namespace std;

class String{ // Örnek (karakter katarı) String sınıfı
    int boy; // Katarın boyu
    char *icerik; // Katarın içeriği
public:
    String(); // Parametresiz kurucu
    String(const char *); // Kurucu
    void operator()(char *, int) const; // () operatörü
    ~String(); // Yok edici fonksiyon
};
// Parametresiz Kurucu Fonksiyon
// Sadece NULL içeren bir boş katar oluşturur
String::String()
{
```

```

    cout<< "Parametresiz Kurucu calisti" << endl;
    boy = 0; // boş katarın boyu sıfır
    icerik = new char[1]; // icerik için yer ayrıldı, null icin
    strcpy(icerik, ""); // boş katar
}

// Kurucu Fonksiyon
// Parametre olarak aldığı katarı nesnenin içeriğine kopyalar
String::String(const char *gelen_veri)
{
    cout<< "Kurucu calisti" << endl;
    boy = strlen(gelen_veri); // gelen katarın boyu hesaplandı
    icerik = new char[boy +1]; // icerik için yer ayrıldı, +1 null icin
    strcpy(icerik, gelen_veri); // gelen veri icerik'in gosterdigi yere kopyalanıyor
}

// Fonksiyon çağırma operatörü(): Bir katarın içeriğinin belli bir kısmını
// verilen bir bellek bölgesine kopyalar
// Parametre olarak hedef belleğin adresini ve kopyalanacak karakter sayısını alır
void String::operator()(char * hedef, int sayi) const
{
    if (sayi>boy) sayi=boy; // sayi katarın boyundan uzunsa
    for (int k=0; k< sayi; k++) hedef[k]= icerik[k];
}

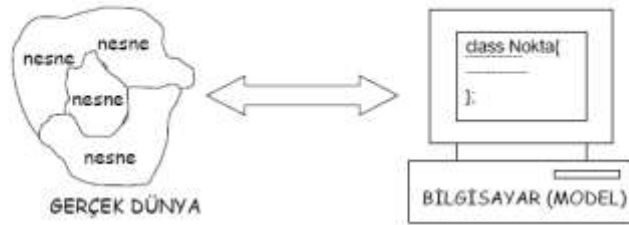
// Yok edici Fonksiyon
// icerik tarafından isaret edilen bellek geri veriliyor
String::~~String()
{
    cout<< "Yok edici calisti" << endl;
    delete[] icerik;
}

//----- Ana Program -----
int main() // Ana fonksiyon
{
    String s1("Ornek Program");
    char *c = new char[8]; // Hedef bellek bölgesi
    s1(c,7); // 7 karakter kopyalandı
    c[7]='\0'; // Katar sonunu belirtmek için (null)
    cout << c << endl; // Kopyalanan karakterler ekrana yazılıyor
    delete c; // Alınan bellek bölgesi iade ediliyor
    return 0;
}

```

Ekran Çıktısı
 Kurucu calisti
 Ornek P

Sınıf Yapısının Sağladıkları



- Modelleme kolaylığı
- Kolay okunur ve anlaşılır programlar
- Özel verilerin korunması
- Hataların yöresel kalması
- Grup çalışmalarında kolaylık
- Yeni veri tipleri yaratma yeteneği

Stream Operatörlerini Aşırı yüklemek

- << ve >> operatörlerini aşırı yükleyelim. Bu sayede kullanıcı tanımlı tiplerle giriş çıkış işlemi yapabiliriz.
- Sol tarafın türü ostream & ve istream & olacak.
- Sol taraf kendi sınıfından olmadığından, üye fonksiyon olarak tanımlayamayız.
- Private data üyelerine ulaşmak için friend fonksiyon olmalı.
- Eğer friend fonksiyon olarak tanımlanmamışlarsa; public'de tanımlı set ve get fonksiyonları üzerinden private data'ya ulaşırlar. Performans gereksinimi nedeniyle bu fonksiyonlar inline yapılır.

Örnek 16 :

```

1 // Fig. 8.3: fig08_03.cpp
2 // Overloading the stream-insertion and
3 // stream-extraction operators.
4 #include <iostream>
5
6 using std::cout;
7 using std::cin;
8 using std::endl;
9 using std::ostream;
10 using std::istream;
11
12 #include <iomanip>
13
14 using std::setw;
15
16 class PhoneNumber {
17     friend ostream &operator<< (ostream&, const PhoneNumber &);
18     friend istream &operator>> (istream&, PhoneNumber &);
19
20 private:
21     char areaCode[ 4 ]; // 3-digit area code and null
22     char exchange[ 4 ]; // 3-digit exchange and null
23     char line[ 5 ];     // 4-digit line and null
24 };
25

```

>> ve << operatörlerinin fonksiyon prototipine dikkat edin: friend fonksiyon olmaları gereklidir.

```

26 // Overloaded stream-insertion operator (cannot be
27 // a member function if we would like to invoke it with
28 // cout << somePhoneNumber;).
29 ostream &operator<<( ostream &output, const PhoneNumber &num )
30 {
31     output << "(" << num.areaCode << " ) "
32         << num.exchange << "-" << num.line;
33     return output;    // enables cout << a << b << c;
34 }
35
36 istream &operator>>( istream &input, PhoneNumber &num )
37 {
38     input.ignore();           // skip (
39     input >> setw( 4 ) >> num.areaCode; // input area code
40     input.ignore( 2 );       // skip ) and space
41     input >> setw( 4 ) >> num.exchange; // input exchange
42     input.ignore();          // skip dash (-)
43     input >> setw( 5 ) >> num.line;    // input line
44     return input;           // enables cin >> a >> b >> c;
45 }
46
47 int main()
48 {
49     PhoneNumber phone;
50
51     cout << "Enter phone number in the form (123) 456-7890:\n";
52
53     // cin >> phone invokes operator>> function by
54     // issuing the call operator>>( cin, phone ).
55     cin >> phone;
56
57     // cout << phone invokes operator<< function by
58     // issuing the call operator<<( cout, phone ).
59     cout << "The phone number entered was: " << phone << endl;
60     return 0;
61 }

```

Fonksiyon çağırılırken cin >> phone; şeklinde, tanımlanırken operator>>(cin, phone); şeklinde olduğuna dikkat edin! cin fonksiyon içinde input, phone ise num adı ile fonksiyon tanımında yer alıyor.

```

Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212

```

Örnek 15 : << ve >> operatörlerini kullanarak nesneye veri okuyabiliyoruz ve nesneyi bir değişkenmiş gibi ekranda yazdırabiliyoruz. Bu örnek toplama için kullanılan örneğe ek olarak yapılmıştır.

```

#include <iostream>
// Örnek 2
// >> ve << operatörleri için overload işlemini diğer örnekle birleştirelim
// bu örnekte +,<<,>> ve kopyalama kurucusu için tüm işlemler
sergilenmektedir.,

```

```

using namespace std;
class deneme
{
private:
    int x,y;
public:
    deneme(deneme &); //kopyalama kurucusu (copy constructor)
    deneme(); // parametresiz kurucu fonksiyon (constructor)
    deneme(int ,int); // parametrelili kurucu fonksiyon
    ~deneme(); // yıkıcı (destructor)
    deneme& deneme::operator+(deneme&); // + operatörü için

```

overloading

```

/* Not!
-----
<< ve >> operatorlerini overload etmek için friend function olarak
tanımlamalısın...
Nesnenin iç yapısına göre input ve output işlemlerinde alınacak veya ekranda
gösterilecek değerler programcıya kalmıştır...

*/
        friend ostream& operator<<(ostream& yaz, deneme& );
        friend istream& operator>>(istream& oku, deneme&);

};

istream& operator>>(istream& oku, deneme& nesne)
{
    oku>>nesne.x>>nesne.y;
    return oku;
}

ostream& operator<<(ostream& yaz,deneme& nesne)
{
    yaz<<nesne.x<<': '<<nesne.y<<endl;
    return yaz;
}

// bu sayede nesneleri sanki sayıymış gibi toplayabileceğiz...
deneme& deneme::operator+(deneme& s)
{
    cout<<" + operatoru calisti"<<endl;
    int xx,yy;
    xx=x+s.x;
    yy=y+s.y;

    cout<<endl;
    cout<<"  xx= "<<xx<<"  yy= "<<yy<<endl;
}

deneme::deneme(int a,int b)
{
    cout<<"Parametreli kurucu fonksiyon calisti.."<<endl;
    x=a;
    y=b;
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
}

```

```

}
deneme::deneme()
{
    cout<<"Kurucu fonksiyon calisti.."<<endl;
    x=5;
    y=10;
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
}
deneme::deneme(deneme &alias)
{

    cout<<"Kopyalama kurucusu calisti"<<endl;
    x=alias.x;
    y=alias.y;
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;

}
deneme::~~deneme()
{
    cout<<"Yikici fonksiyon calisti..";
}

int main()
{

    deneme d,c(3,2),e; // c nesnesi d nin deęerleri ile doęuyor...
    e=d+c;
    // aslında alt kisim ile ust kisim ayni isi yapiyor...
    e=d.operator+(c);

    deneme x; // yeni bir nesne oluřturalım
    // eęer sınıf ierisinde pointer yoksa; eřittir iřlemi iin
    // ek olarak operator overload etmenize gerek yoktur...
    d=c; // grdęnz gibi bu iřlem nesneleri birbirine sorunsuzca eřitliyor.
    x=d+c;

    cout<<c;
    cin>>c;
    cout<<c;

    system("pause");
    return 0;
}

```

```

Kurucu fonksiyon calisti..
x=5
y=10
Parametrelili kurucu fonksiyon calisti..
x=3
y=2
Kurucu fonksiyon calisti..
x=5
y=10
+ operatoru calisti

xx= 8 yy= 12
+ operatoru calisti

xx= 8 yy= 12
Kurucu fonksiyon calisti..
x=5
y=10
+ operatoru calisti

xx= 6 yy= 4
3:2
111 ----> Bu deęerleri klavyeden biz girdik
6789 ----> Bu deęeri klavyeden biz girdik
111:6789 ----> Ekrana yazdırırken arasına :
koydurduk
Press any key to continue . . .

```