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 bazılarına 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. Yeni oluşturulan operatörlerin en az bir operandının tipi bir sınıf olmalıdır.

+ Operatörüne Yeni Bir İşlev Yüklenmesi

Aşağıdaki örnekte ComplexT sınıfına + operatörü için bir metot eklenecektir. Böylece + operatörünün karmaşık sayıları toplaması sağlanacaktır.

```
class ComplexT{            // Karmaşık (Kompleks) sayıları tanımlamak için oluşturulan sınıf
   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;
};

// + 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);
}

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

Bkz Örnek o51.cpp

Atama Operatörüne "=" Yeni Bir İşlev Yüklenmesi

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 aşağıda gösterilen atama fonksiyonu gereksizdir.

```
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.

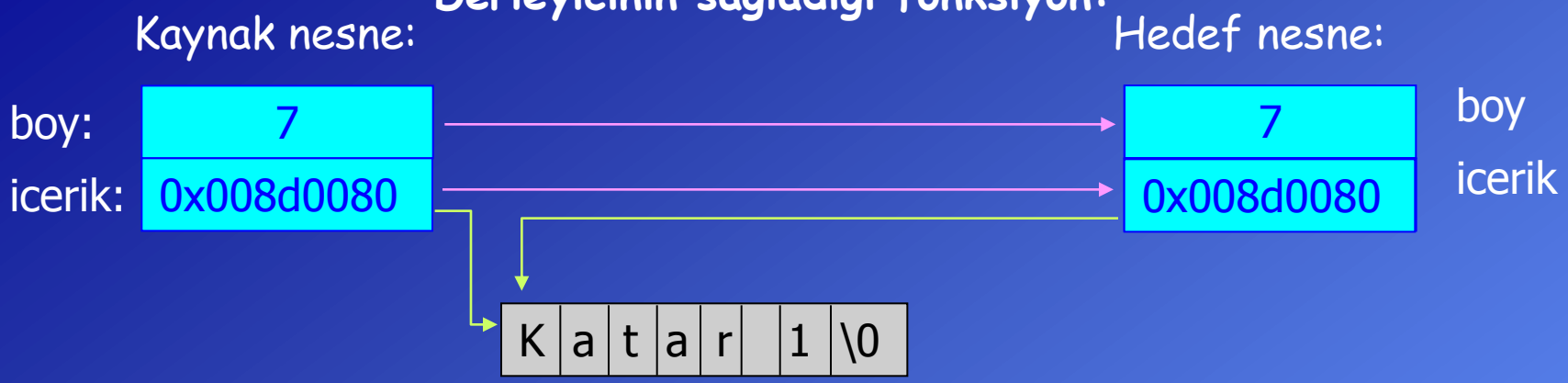
Bkz Örnek o52.cpp

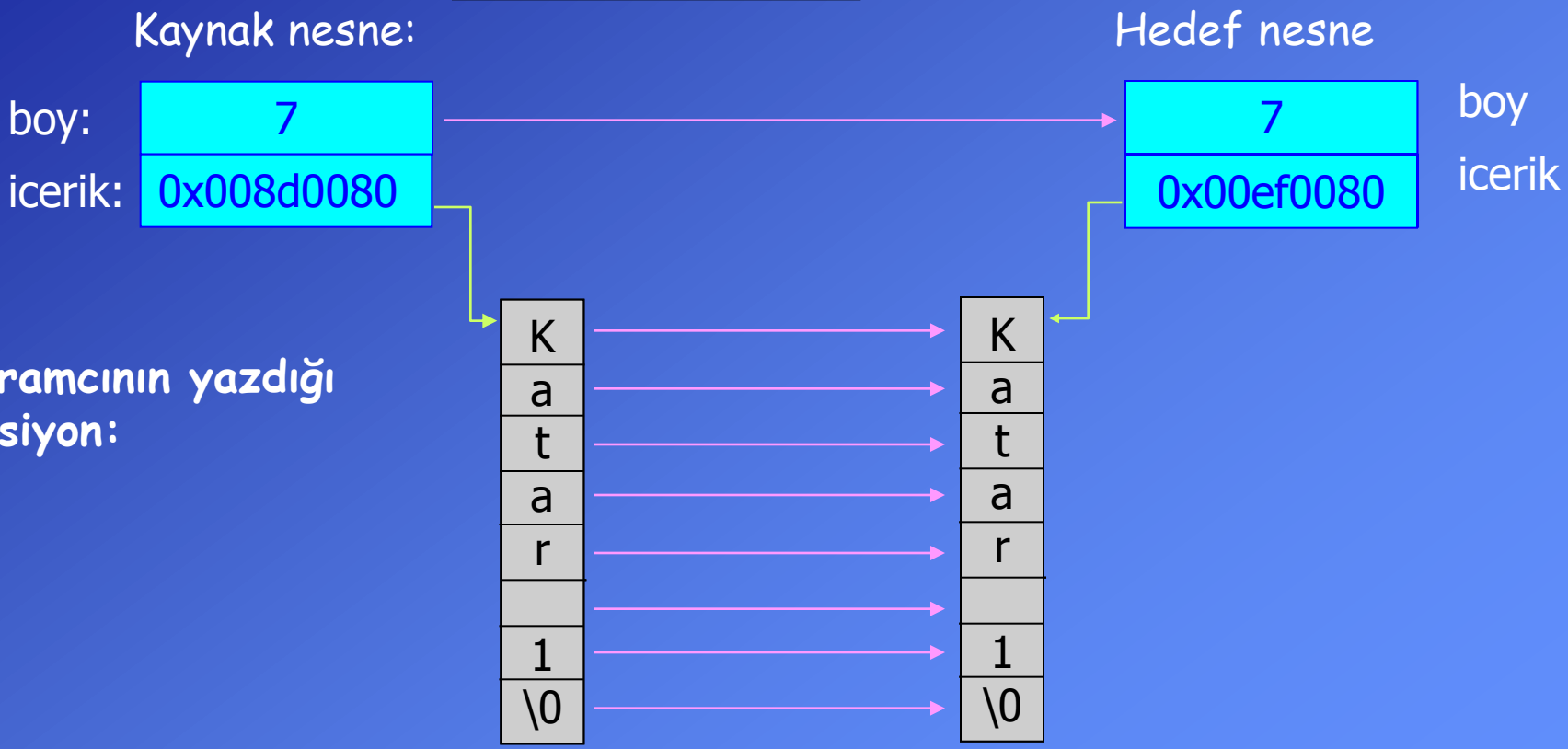
Ancak her sınıf için derleyicinin sağladığı atama fonksiyonu yeterli olmayabilir. Özellikle içinde işaretçi olan sınıflarda programcının atama operatörüne ilişkin fonksiyonu yazması gerekebilir. Kopyalama kurucusundakine benzer bir problem burada da vardır. Aşağıda örnek String sınıfı için atama fonksiyonu 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 icin
    strcpy(icerik, gelen_nesne.icerik);
}
```

Derleyicinin sağladığı fonksiyon:





Programcının yazdığı fonksiyon:

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

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.

```
// Atama Operatörü
const String& 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 icin
    strcpy(icerik, gelen_nesne.icerik);
    return *this;                        // Kendi adresini geri gönderiyor
}
```

Bkz Örnek o53.cpp

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

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:

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 Operatörü
char& String::operator[](int i)
{
    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
}
//-------- 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;
}
```

Bkz Örnek o54.cpp

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

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

Bkz Örnek o55.cpp

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

Örnekteki operatör, bir karakter katarının (String) istenen bir miktarını, istenen bir bellek bölgesine kopyalamaktadır.

```
// 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 sayi=boy olur
    for (int k=0; k< sayi; k++) hedef[k]= icerik[k];
}

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

Bkz Örnek o56.cpp

Bir Operandlı (*Unary*) Operatörlere Yeni Bir İşlev Yüklenmesi

Birli operatörlere örnekler: arttırma (++), azaltma(--), eksileme(-5), mantıksal tümleme (!) 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: 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 canlanir
    z.goster();
    return 0;
}
```

Eğer bu operatörün bir atama deyiminde kullanılması isteniyorsa operatöre ilişkin fonksiyon nesnenin referansını geri döndürülmelidir.

```
// ++ 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
}

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

Bkz Örnek o57.cpp

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.

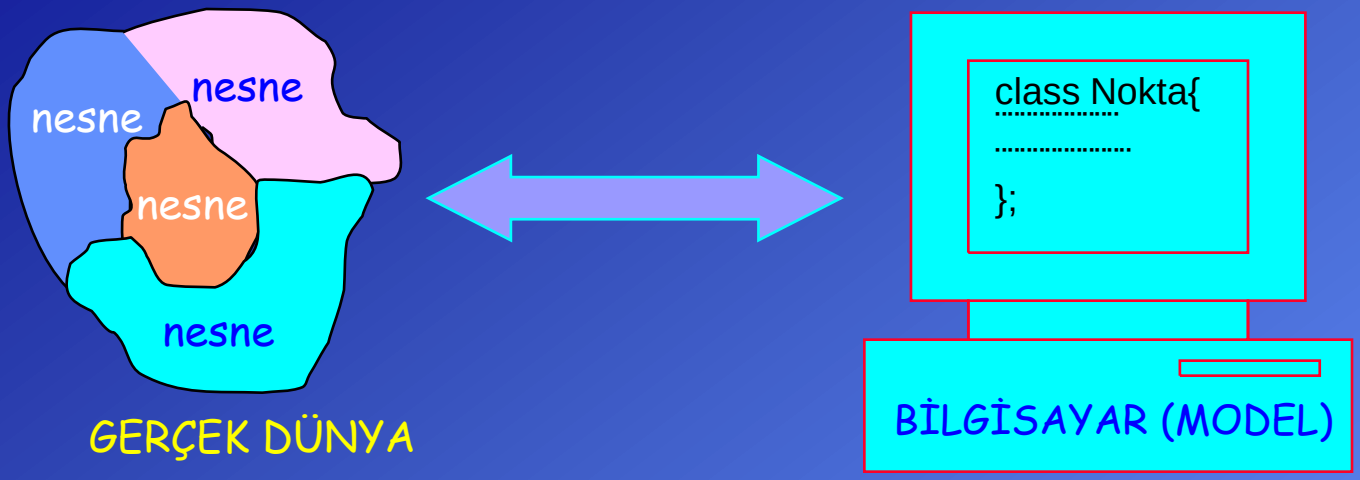
```
z2= ++ z1;     // önceden arttırma
z2 = z1++;     // sonradan arttırma
```

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

```
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
}
```

Bkz Örnek o58.cpp

Sınıf Yapısının Sağladıkları



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