Kurucu Fonksiyonlar (Constructors)

Kurucu fonksiyonlar üyesi oldukları sınıftan bir nesne yaratılırken kendiliğinden canlanırlar.

Bu tür fonksiyonlar bir nesnenin kurulması aşamasında yapılması gereken işleri, örneğin verilere uygun başlangıç değerleri atamak için kullanılırlar.

Kurucu fonksiyonlar üyesi oldukları sınıf ile aynı ismi taşırlar.

Kurucular parametre alırlar, ancak geri dönüş değerleri yoktur. Geri dönüş tipi olarak herhangi bir tip (void bile) yazılmaz.

Kurucu fonksiyonlar nesne yaratılırken sınıfın dışından çağırılacağından açık (public) üyeleri arasında yer almalıdırlar.

Kurucu fonksiyonlar işlevlerine ve yapılarına göre bazı alt gruplara ayrılırlar. İlk grupta parametre verilmeden çağrılabilen parametresiz kurucu fonksiyonlar yer alır.

Parametresiz Kurucu Fonksiyonlar (*Default Constructor*)

Bu tür kurucu fonksiyonların ya parametre listeleri boştur, ya da tüm parametrelerin bir başlangıç değeri vardır.

```
class Nokta{                    // Nokta Sınıfı
   int x,y;                     // Nitelikler: x ve y koordinatları
 public:
   Nokta();                     // Kurucu fonksiyon bildirimi
   bool git(int, int);          // Noktanın hareket etmesini sağlayan fonksiyon
   void goster();               // Noktanın koordinatlarını ekrana çıkartır
};

// Parametresiz Kurucu Fonksiyon
Nokta::Nokta()
{
   cout << "Kurucu fonksiyon çalışıyor..." << endl;
   x = 0;                       // Koordinatlar sıfırlanıyor.
   y = 0;
}
```

Bkz Örnek o41.cpp

Ana programda kurucu fonksiyonların çağrılması için özel bir deyim yazılmaz. Nesnelerin yaratıldığı satırlarda kurucu fonksiyon her nesne için bir defa çalışır.

Parametreli Kurucu Fonksiyonlar

Kurucu fonksiyonlar da diğer üye fonksiyonlar gibi gerektiğinde parametre alacak şekilde tanımlanabilirler.

Bu durumda sınıftan nesne yaratan programcılar, nesneleri tanımladıkları satırlarda kurucu fonksiyonlara uygun tipte ve sayıda argümanı vermek zorundadırlar.

```
class Nokta{                    // Nokta Sınıfı
    int x,y;                    // Nitelikler: x ve y koordinatları
 public:
   Nokta(int,int);              // Kurucu fonksiyon bildirimi
    bool git(int, int);         // Noktanın hareket etmesini sağlayan fonksiyon
    void goster();              // Noktanın koordinatlarını ekrana çıkartır
};
```

```
Nokta::Nokta(int ilk_x, int ilk_y)          // Parametreli Kurucu Fonksiyon
{
    cout << "Kurucu fonksiyon çalışıyor..." << endl;
    if ( ilk_x < 0 )                        // Verilen değer negatifse
            x = 0;                          // Koordinat sıfırlanıyor
    else
            x = ilk_x;
    if ( ilk_y < 0 )                        // Verilen değer negatifse
            y = 0;                          // Koordinat sıfırlanıyor
    else
            y = ilk_y;
}
// -------- Ana Program ------------
int main()
{
    Nokta n1(20,100), n2(-10,45);           // Kurucu 2 kez çalışır
    Nokta *pn = new Nokta(10,50);           // Kurucu 1 kez çalışır.
    // Nokta n3;                            // HATA! Parametresiz kurucu yok
    n1.goster();                            / n1'in koordinatları ekrana çıkartılıyor
    n2.goster();                            //n2'nin koordinatları ekrana çıkartılıyor
    pn->goster();                           //pn2'nin işaret ettiği nesne ekrana çıkar
    delete pn;
    return 0;
}
```

Bkz Örnek o42.cpp

Kurucu Parametrelerine Başlangıç Değeri Verilmesi:

Diğer fonksiyonlarda olduğu gibi kurucu fonksiyonların parametrelerine de başlangıç değeri verilebilir. Bu durumda nesne yaratılırken verilmeyen argümanların yerine parametrelerin başlangıç değerleri kullanılacaktır.

```
// Kurucu Fonksiyon
Nokta::Nokta(int ilk_x = 0, int ilk_y = 0)
{
    cout << "Kurucu fonksiyon çalışıyor..." << endl;
    if ( ilk_x < 0 )                    // Verilen değer negatifse
            x = 0;                      // Koordinat sıfırlanıyor
    else
            x = ilk_x;
    if ( ilk_y < 0 )                    // Verilen değer negatifse
            y = 0;                      // Koordinat sıfırlanıyor
    else
            y = ilk_y;
}
```

Bu sınıftan aşağıdaki gibi nesneler yaratılabilir.

```
Nokta n1(15,75);        // x=15,  y=75
Nokta n2(100);          // x=100, y=0
```

Bu fonksiyon parametresiz bir kurucu olarak da kullanılabilir.

```
Nokta n3;               // x=0, y=0
```

Bir Sınıfta Birden Fazla Kurucu Fonksiyon Olması:

Bir sınıfta birden fazla kurucu fonksiyon bulunabilir. Fonksiyon isimlerinin yüklenmesi (*function overloading*) konusunda da değinildiği gibi isimleri aynı olan bu fonksiyonların bir belirsizlik olmadan çağrılabilmeleri için parametrelerinin tipleri ve/veya sayıları farklı olmalıdır.

```
Nokta::Nokta()                          // Parametresiz kurucu fonksiyon
{
.............                           // Gövdesi önemli değil
}

Nokta::Nokta(int ilk_x, int ilk_y)      // Parametreli kurucu fonksiyon
{
.............                           // Gövdesi önemli değil
}
```

İki kurucuya sahip bir Nokta sınıfından farklı şekillerde nesneler yaratılabilir.

```
Nokta n1;                               // Parametresiz kurucu çalışır
Nokta n2(30,10);                        // Parametreli kurucu çalışır
```

Örnek sınıfta tek parametreli bir kurucu olmadığından aşağıdaki satır hatalıdır.

```
Nokta n3(10);                           // HATA! Bir parametreli kurucu yok
```

Kurucuların Nesne Dizileri ile Kullanılması

Bir sınıfta parametresiz bir kurucu fonksiyon varsa bu sınıftan bir nesne dizisi yaratıldığında dizinin her elemanı için kurucu fonksiyon kendiliğinden canlanır.

```
Nokta dizi[10];                 // Parametresiz kurucu 10 defa çalışır
```

Eğer bir parametre alan kurucu fonksiyona sahip bir sınıftan nesne dizisi yaratılacaksa, kurucuya gerekli argümanları göndermek için **başlangıç değerleri listesi** kullanılır. Bu listenin başı ve sonu kıvırcık parantezler ({ }) ile belirlenir. Ayrıca kurucuya gönderilecek her değer de kıvırcık parantezler içine yazılır.

```
// Kurucu Fonksiyon
Nokta::Nokta(int ilk_x, int ilk_y = 0)

Nokta dizi[]= { {10} , {20} , Nokta(30,40) };     // 3 elemanlı nesne dizisi
```

Eğer Nokta sınıfında yukarıdaki kurucu fonksiyona ek olarak parametresiz bir kurucu fonksiyon da yer alsaydı aşağıda gösterildiği gibi 5 elemanlı bir dizi yaratılabilirdi.

```
Nokta dizi[5]= { {10} , {20} , Nokta(30,40) };     // 5 elemanlı nesne dizisi
```

Yukarıda başlangıç değerleri listesine sadece üç değer vardır. Bu durumda dizinin ilk üç elemanına listedeki değerler sırasıyla gönderilecektir. Son iki eleman için ise parametresiz kurucu fonksiyon çalıştırılacaktır.

Kurucu Fonksiyonlarda İlk Değer Atama (*Constructor Initializers*)

Kurucu fonksiyonlarda nesnelerin verilerine ilk değerlerini atamak için C++'nın atama deyiminden daha farklı bir yapı da kullanılabilmektedir. Özellikle sabit verilere ilk değerlerini atamak için bu yapının kullanılması zorunludur.

Bir sınıfın kurucu fonksiyonunda sabit veriye başlangıç değeri atanmaya çalışılırsa derleme hatası oluşur, çünkü sabit bir veri bir atama işleminin solunda yer alamaz.

```
class C{                        // Örnek C sınıfı
    const int ci;               // sabit üye veri
    int x;                      // sabit olmayan üye veri
  public:
    C() {                       // Kurucu fonksiyon
        x = -2;                 // Doğru, çünkü x sabit değil
        ci = 0;                 // HATA! Çünkü ci sabit
    }
};
```

Aşağıdaki gibi bir yazım da derleme hatasına neden olur.

```
class C{                                // Örnek C sınıfı
    const int ci = 10;                  // HATA! sabit üye veri
    int x;                              // sabit olmayan üye veri
};
```

Bu problemin kurucu fonksiyonlarda ilk değer atama yapısı (*constructor initializer*) kullanılarak çözülür. Kuruculardaverilere ilk değer atamak için kurucu fonksiyonun imzasından sonra iki nokta üst üste (:) konur, ardından ilk değer atanacak verinin adı gelir ve parantez içinde atanacak değer yazılır.

```
class C{                        // Örnek C sınıfı
    const int ci;               // sabit üye veri
    int x;                      // sabit olmayan üye veri
  public:
    C() : ci(0) {               // Kurucu fonksiyon. ci'ye sıfır atanıyor.
        x = -2;                 // Doğru, çünkü x sabit değil
    }
};
```

Tüm üye verilere değer atamak için bu yapı kullanılabilir. Eğer kurucu fonksiyonun değer atamaktan başka bir görevi yoksa gövdesi boş kalabilir.

```
class C{                        // Örnek C sınıfı
    const int ci;               // sabit üye veri
    int x;                      // sabit olmayan üye veri
  public:
    C() : ci(0), x(-2)          // Kurucu fonksiyon, ilk değerler atanıyor
    { }                         // Gövde boş
};
```

Yok Edici Fonksiyonlar (*Destructors*)

Bu fonksiyonlar üyesi oldukları sınıftan yaratılmış olan bir nesne bellekten kaldırılırken kendiliğinden çalışırlar.

Bir nesnenin bellekten kaldırılması için ya nesnenin yaşam alanı sona ermelidir (tanımlandığı blok sona ermiştir) ya da dinamik bellekte tanımlanmış olan bir nesne delete operatörü ile bellekten silinmelidir.

Yok edici fonksiyonlar da kurucular gibi sınıf ile aynı ismi taşırlar, ancak isimlerinin önünde 'tilda' simgesi (~) yer alır. Yok ediciler parametre almazlar ve geriye değer döndürmezler. Bir sınıfta sadece bir adet yok edici fonksiyon olabilir.

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

C++ standart arşivinde katarları tanımlamak için **string** adında hazır bir sınıf vardır.

```
// Kurucu Fonksiyon
String::String(const char *gelen_veri)
{
    cout<< "Kurucu çalıştı" << endl;
    boy = strlen(gelen_veri);              // gelen dizinin boyu hesaplandı
    icerik = new char[boy +1];             // icerik için yer ayrıldı. +1 null için
    strcpy(icerik, gelen_veri);            // gelen veri icerik'in gösterdiği yere
 }
void String::goster()
{
    cout<< icerik << ", " << boy << endl;       // Katar içeriği ve boyu ekrana
}

// Yok edici Fonksiyon
String::~String()
{
    cout<< "Yok edici çalıştı" << endl;
    delete[] icerik;
}
```

```
//-------- Ana Program ---------------
int main()
{
    String string1("Katar 1");
    String string2("Katar 2");
    string1.goster();
    string2.goster();
    return 0;          // Yok edici iki defa çalışır
}
```

Bkz Örnek o43.cpp

Kopyalama Kurucusu (*Copy Constructor*)

Kopyalama kurucusu özel bir kurucu fonksiyondur.

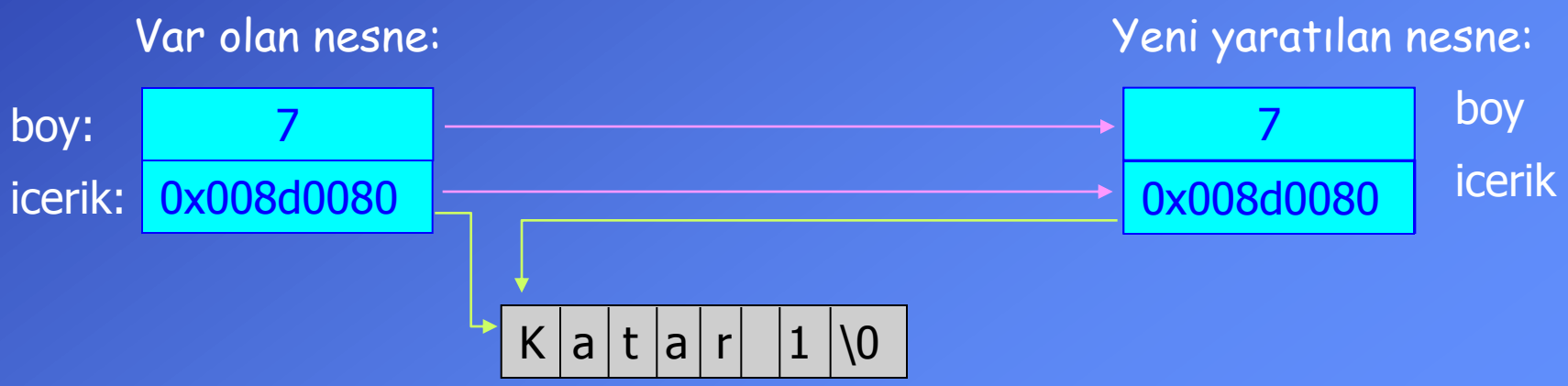
Diğer kurucu fonksiyonlar gibi bir nesne yaratılırken kendiliğinden canlanırlar.

Var olan bir nesnedeki verileri yeni yaratılan nesnenin içine kopyalarlar. Böylece yeni yaratılan nesne var olan eski bir nesnenin kopyası olur.

Bu fonksiyonlar giriş parametresi olarak aynı sınıftan bir nesneye referans alırlar. Bu kopyası çıkarılacak olan nesneye bir referanstır.

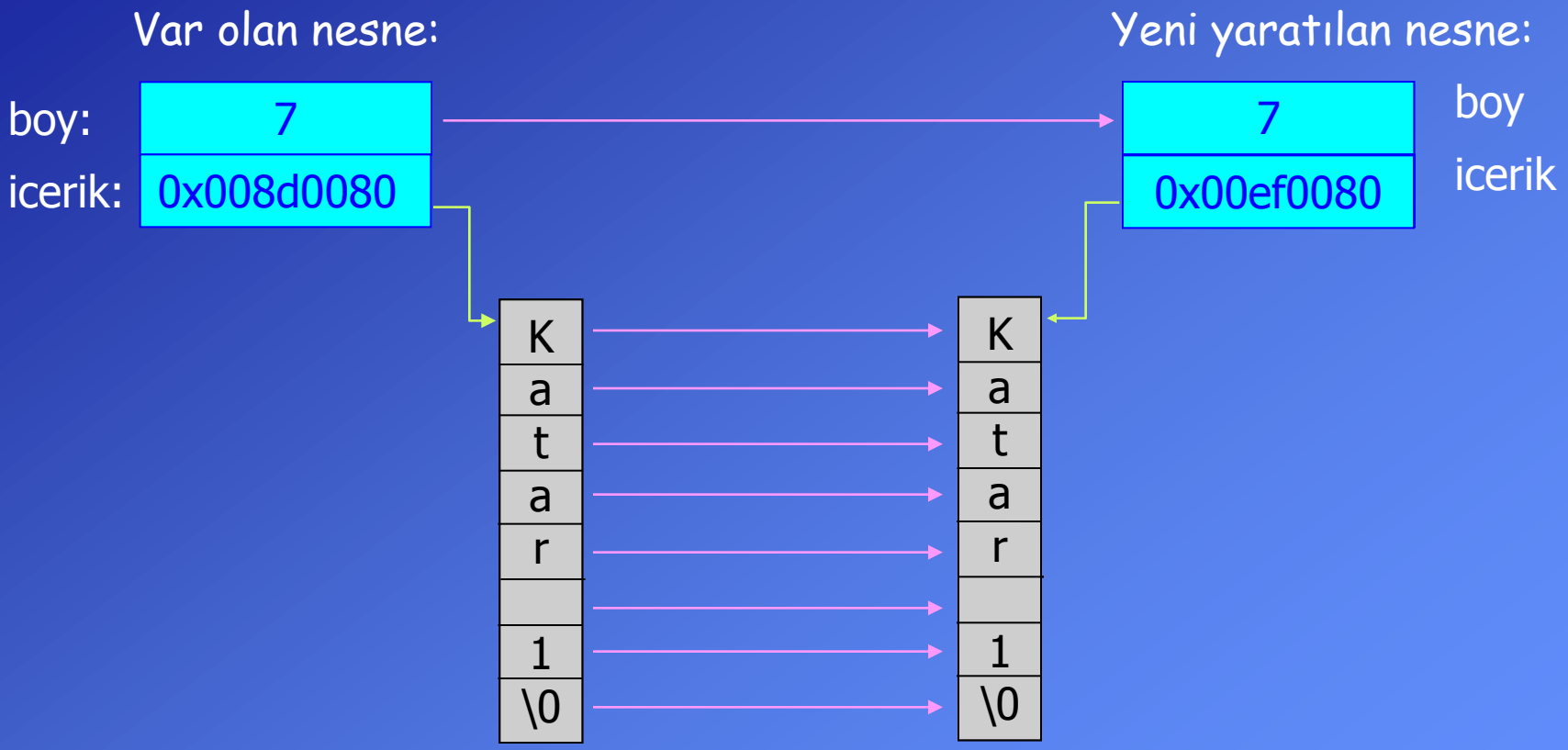
Eğer programcılar sınıflarının içine bir kopyalama kurucusu koymazlarsa, derleyici standart bir kurucuyu sınıfa yerleştirir. Standart kopyalama kurucusu bir nesnenin elemanlarını bire bir yeni nesnenin veri alanlarına kopalar. İçinde işaretçi olmayan nesneler için bu genellikle yeterlidir.

Örneğin bir önceki örnekteki String sınıfı için derleyicinin yerleştireceği kopyalama kurucusu aşağıdaki işlemleri yapacaktır.



String sınıfı için verilen örnekte de görüldüğü gibi derleyicinin sağladığı kopyalama fonksiyonu sadece nesnenin elemanlarını kopyalamaktadır.

İşaretçilerin işaret ettiği veriler kopyalanamaz. Bu alanların da kopyalanması isteniyorsa programcı kendi kopyalama fonksiyonunu yazmalıdır.



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

String::String(const String &gelen_nesne)   // Kopyalama kurucusu
{
    boy = gelen_nesne.boy;
    icerik = new char[boy + 1];             //  +1 null karakteri icin
    strcpy(icerik, gelen_nesne.icerik);
}

int main()                                      // Ana fonksiyon
{
   String string1("Katar 1");
   string1.goster();
   String diger = string1;                      // Kopyalanma kurucusu çalışır
   String baska(string1);                       // Kopyalanma kurucusu çalışır
   diger.goster();
   baska.goster();
   return 0;
 }
```

Bkz Örnek o44.cpp

Sabit Nesneler ve Sabit Fonksiyonlar

Diğer veri tiplerinde olduğu gibi bir nesne de sabit (const) olarak tanımlanabilir. Bunun anlamı nesnenin veri alanlarının program boyunca doğrudan ya da dolaylı olarak (fonksiyon çağırarak) değiştirilemeyeceğidir.

```
const Nokta sn(10,20);        // Sabit nokta
```

Derleyiciler sabit olarak tanımlanan nesnelerin içeriklerinin değişmemesi için bu nesnelerin üye fonksiyonlarının çağırılmasına izin vermezler.

Sınıfın yazarları üye veriler üzerinde değişiklik yapmayan fonksiyonları da sabit (const) olarak bildirmelidirler. Sabit nesneler için sadece sabit fonksiyonlar çağırılabilirler.

```
class Nokta{                          // Nokta Sınıfı
   int x,y;                           // Nitelikler: x ve y koordinatları
 public:
   Nokta();                           // Kurucu fonksiyon bildirimi
   bool git(int, int);                // Noktanın hareket etmesini sağlayan fonksiyon
   void goster() const;               // sabit fonksiyon, koordinatları ekrana çıkartır
};
```

```
void Nokta::goster() const
{
    cout << "X= " << x << ", Y= " << y << endl;
}

int main()
{
    const Nokta sn(10,20);          // sabit nokta
    Nokta n(0,50);                  // sabit olmayan nokta
    sn.goster();                    // Doğru
    sn.git(30,15);                  // HATA!
    n.git(100,45);                  // Doğru
    return 0;
}
```

Sınıfın verileri üzerinde değişiklik yapmayan fonksiyonların sabit olarak tanımlanmaları hata olasılığını ve hata çıktığında incelenmesi gereken fonksiyonların sayısını azaltmaktadır. Bu nedenle bu özelliğe uyan tüm üye fonksiyonlar sabit olarak tanımlanmalıdır.

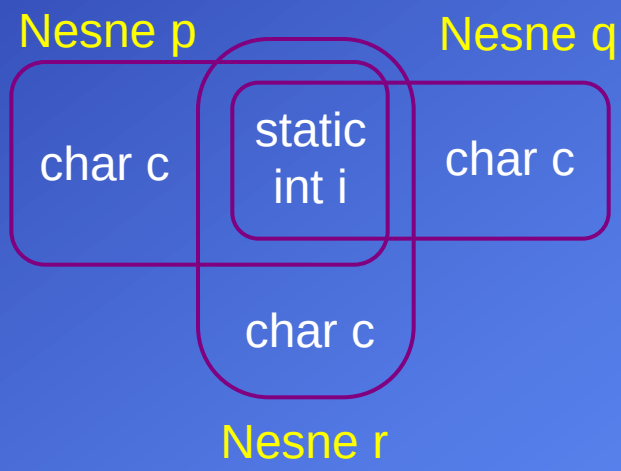
Bkz Örnek o45.cpp

Statik Üyeler

Bir sınıftan tanımlanan her nesne için bellekte farklı veri alanları yaratılır. Ancak bazı durumlarda tüm nesnelerin ortak bir veriyi (bellek gözünü) paylaşmaları gerekli olabilir.

Bellekte sadece tek kopyasının yaratılması istenen üye veriler **static** olarak tanımlanmalıdırlar.

```
class A{
   char c;
   static int i;
};
int main()
{
    A p,q,r;
 :
}
```



Statik üyeler nesne tanımlanmadan önce bellekte yaratılırlar.

Statik üyeler de diğerleri gibi özel (private) veya açık (public) olabilirler. Açık statik üyeler, global veriler gibi programın tüm alanlarından erişilebilirler. Bunun için sınıfın ismi ve 'scop' operatörü (::) kullanılır. **A::i= 5;**

Statik üyeler özel (private) olarak tanımlanırsa bu üyelere doğrudan erişmek mümkün olmaz. Henüz nesne yaratılmadan önce bu verilere başlangıç değeri atamak için statik fonksiyonlar tanımlanır.

Bkz Örnek o46.cpp

Nesnelerin Fonksiyonlara Parametre Olarak Aktarılması

Aksi zorunlu olmadıkça, nesneler fonksiyonlara **referanslar** yoluyla aktarılmalı. Benzer şekilde fonksiyonlardan geri döndürülen nesneler için de eğer mümkünse referanslar kullanılmalıdır.

Parametre aktarımında referanslar kullanılmazsa nesnelerin içerdiği değerler yığına kopyalanır (*call by value*). Eğer sınıfın içinde bir kopyalama fonksiyonu varsa yığına kopyalama işi için de programcının yazdığı bu fonksiyon canlanacaktır.

Bkz. Örnek o47.cpp

Nesnelerin fonksiyonlara değer olarak aktarılması hem bellekte daha fazla yer harcanmasına neden olur hem de programın çalışmasını yavaşlatır. Bu nedenle, eğer aksi zorunlu değilse, fonksiyonlara nesnelerin değerleri değil referansları aktarılmalıdır.

Referans yoluyla aktarılan nesnelerin içeriklerinin fonksiyon içinde değiştirilmesi istenmiyorsa bu aktarım sabit referanslar ile yapılmalıdır.

Bkz. Örnek o48.cpp

```
ComplexT & ComplexT::topla(const ComplexT & c) const
{
  ComplexT sonuc;          // Yerel nesne
  sonuc.re=re+c.re;        // Alt alanlar toplanıyor
  sonuc.im=im+c.im;
  return sonuc;            // HATA!
}
```

Hatırlatma: Yerel değişkenler referans yolu ile geri döndürülemez. Bu nedenle yandaki fonksiyon hatalıdır.

Fonksiyonlardaki Geçici Nesnelerin Azaltılması

Örnek 4.8'de toplama fonksiyonunda iki nesneyi toplamak için geçici bir nesne yaratılmaktadır. Bu işlem kurucu ve yok edici fonksiyonların çalışmasına neden olmaktadır.

Ardından bu geçici nesne fonksiyondan geri gönderileceği için (return gecici;) yığına kopyalanmaktadır.

Bu işlemleri azaltmak için geçici bir nesne yaratmak yerine, işlemler geçici değişkenler üzerinde yapılır. Daha sonra fonksiyondan değer döndürülürken nesne yaratılır.

```
ComplexT ComplexT::topla(const ComplexT & z) const
{
   float  gecici_re, gecici_im;       // Bütün bir nesne yerine sadece geçici değişkenler
   gecici_re = re + z.re;             // Toplama işlemi yapılıyor
   gecici_im = im + z.im;
   return ComplexT(gecici_re,gecici_im);    // Nesne yaratılıyor, kurucu canlanır
}
```

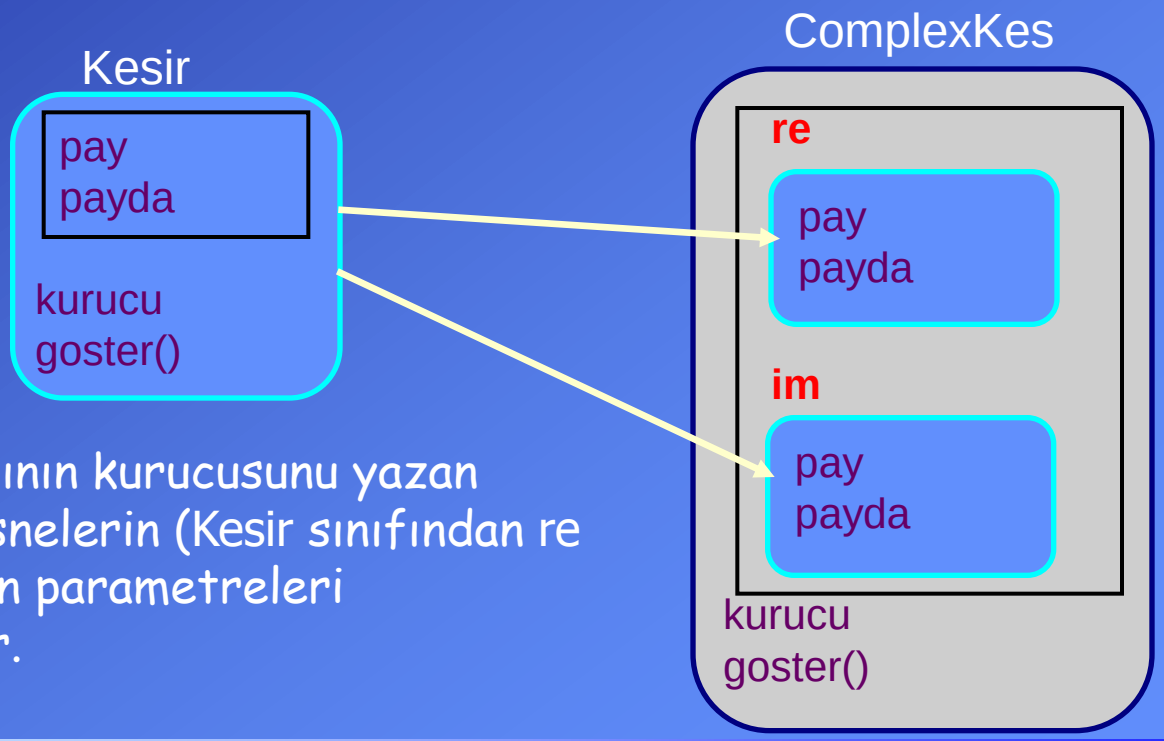
Bkz. Örnek o49.cpp

İç içe nesneler: Nesnelerin başka sınıfların üyesi olması

Bir sınıfın üyeleri sadece hazır veri tipleri (char, int, double....) olmak zorunda değildir. Bir sınıfın üyeleri başka sınıftan tanımlanmış nesneler de olabilirler.

Sınıflar arası bu ilişkiye sahip olma ilişkisi (*has a relation*) adı verilir.

Aşağıdaki örnekte karmaşık sayıları tanımlamak üzere bir sınıf (ComplexKes) oluşturulmuştur. Bu sınıfın reel ve sanal kısımları bayağı kesirlerden oluşacaklardır. Bayağı kesirler ise (Kesir) sınıfından tanımlanan nesneler ile oluşturulacaktır.



Burada ComplexKes sınıfının kurucusunu yazan programcı, kullandığı nesnelerin (Kesir sınıfından re ve im) kurucularına uygun parametreleri göndermekle yükümlüdür.

Örnek olarak hazırlanan Kesir sınıfı:

```
class Kesir{                                  // Bayağı kesirleri tanımlamak için
    int pay,payda;
 public:
    Kesir(int, int);                          // Kurucu
    void goster() const;
};

Kesir::Kesir(int py, int pyd)                 // KURUCU
{
    pay=py;
    if (pyd==0) payda=1;                      // Payda olarak 0 verilirse payada=1 olur
    else payda=pyd;
    cout << "Kesir'in kurucusu" << endl;
}

// Bayağı kesiri ekrana çıkaran fonksiyon
void Kesir ::goster() const
{
    cout << pay << "/" << payda << endl;
}
```

Örnek olarak hazırlanan ComplexKes sınıfı:

```
// Elemanları bayağı kesir olan karmaşık sayıları tanımlayan sınıf
class ComplexKes{
    Kesir re,im;                    //  üye nesneler
 public:
    ComplexKes(int,int);            // Kurucu
    void goster() const;
 };

// Kurucu fonksiyon
// Önce üye nesnelerin kurucularına gerekli parametreler gönderiliyor
ComplexKes::ComplexKes(int re_in,int im_in) : re(re_in,1), im(im_in,1)
{
    ....
}

// Karmaşık sayı ekrana çıkarılıyor
void ComplexKes::goster() const
{
    re.goster();
    im.goster();
}
```

Üye nesnelerin kurucuları canlanır

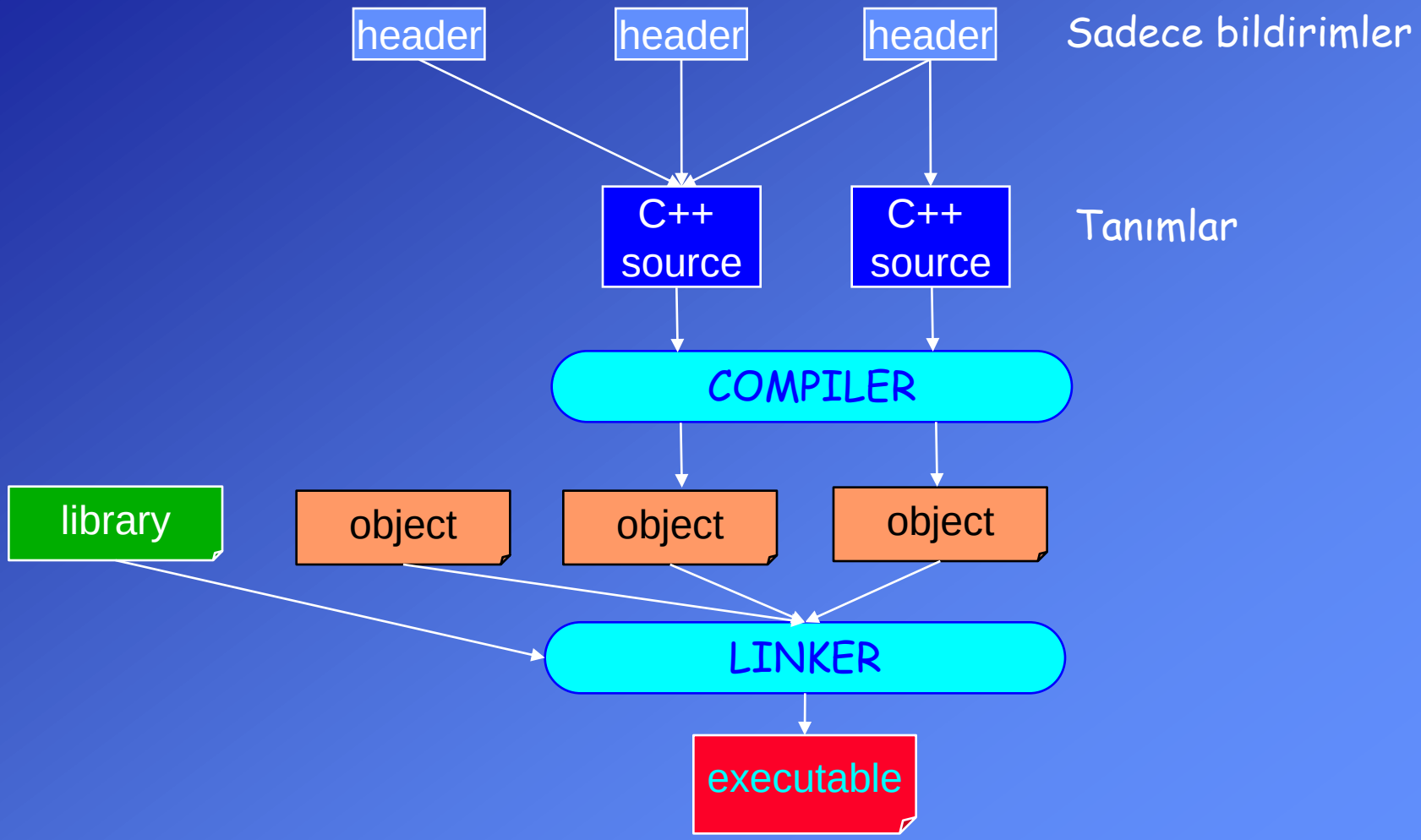
```
int main()                          // Ana program
{
    ComplexKes ck(2,5);             // Bir karmaşık sayı tanımlanıyor
    ck.goster();                    // Karmaşık sayı ekrana
    return 0;
}
```

Bkz. Örnek o410.cpp

Bkz. Örnek o411.cpp

Birden Fazla Dosya ile Çalışmak (Ayrı Ayrı Derleme)

Her sınıfı veya birbirleriyle ilgili sınıfları ayrı dosyalarda yazmak iyi bir yöntemdir. Yazılımın karmaşıklığının kontrolünü ve sınıfların farklı projelerde tekrar kullanılabilmesini kolaylaştırır.



Ayrı ayrı derleme yöntemi kullanılıyorsa, her dosyanın otomatik olarak derlenmesi ve bağlayıcı (linker) tarafından bütün parçaların ve arşivlerin birleştirilerek çalıştırılabilir (executable) bir dosya oluşturulmasınının sağlanması için bir yönteme ihtiyaç vardır.

Çözüm, Unix ortamında geliştirilen fakat başka ortamlarda da farklı şekillerde bulunan **make** programıdır.

Derleyici tedarikçileri projelerin oluşturulması için kendi araçlarını geliştirmişlerdir. Bu araçlar hangi dosyaların projede olduğunu sormakta, ve bu dosyalar arasındaki ilişkileri otomatik olarak belirlemektedir. Genellikle proje dosyası (*project file)* denilen ve **makefile**'a benzeyen bir yapı kullanılmaktadır. Bu dosya programlama ortamı (integrated development environment) tarafından kontrol edilmektedir.

Konfigürasyon ve proje dosyalarının kullanımı programlama ortamlarına göre değişiklik göstermektedir. Çalıştığınız programlama ortamına göre bunların nasıl kullanılacağı ile ilgili uygun dokümantasyonu bulmanız gerekmektedir.

e410.cpp'deki kesirli ve karmaşık sayılarla ilgili örneği tekrar yazacağız. Bu sefer, kesirli ve karmaşık sayılar sınıflarını ayrı dosyalara koyacağız.

Bkz. Örnek: e412.zip