# ÇOK ŞEKİLLİLİK

Binnur Kurt
kurt@cs.itu.edu.tr

*Bilgisayar Mühendisliği Bölümü*
*İstanbul Teknik Üniversitesi*



Sınıf Yapısı

Kalıtım

## Çok Şekillilik

## Çok Şekillilik Nedir ?

Gerçek hayattaki nesneler farklı sınıflardan olsalar da bir çok durumda farklı davranışları gerçekleştirseler de, bazı ortak işlevlere sahiptirler. Örnek olarak bir **kare**, **daire**, **üçgen** sınıflarından nesneleri ele alalım:

Tüm bu nesnelere "Alan Hesapla" mesajını göndermiş olalım. Farklı tipte nesneler (**kare**, **daire**, **üçgen**) farklı alan hesabına sahiptir. Ancak farklı tipte nesnelere farklı mesaj göndermeye gerek yoktur. Bu işlem için tek bir tür mesaj (**Alan**) tüm nesnelerde için çalışmalıdır. Çünkü her bir tip nesne kendi sınıfından nesnelerin alanını nasıl hesaplayacağını bilmektedir:

**kare→Alan() ;**

**daire→Alan();**

**üçgen→Alan();**



## ÇOK ŞEKİLLİLİK devam

**Alan** işlevi farklı tipte nesnelerde farklı biçimler aldığından ***çok şekilli*** bir komuttur.

Genellikle çok şekillilik birbirleri ile kalıtımla ilişkili sınıflar arasında gerçekleşir. C++'da çok şekilliğin anlamı, bir üye fonksiyonuna yapılan çağrının, farklı nesnelerde nesnenin tipine bağlı olarak farklı fonksiyonların çağrılmasına neden olmasıdır.

Bu, biraz fonksiyona işlev yüklemeyi çağrıştırmaktadır. Ancak çok şekillilik daha güçlü ve farklı bir mekanizma sunmaktadır. Önemli fark, hangi fonksiyonun çağrılacağına ne zaman karar verildiğinde ortaya çıkmaktadır.

Fonksiyon yüklemede bu karar derleyici tarafından derleme aşamasında verilirken, çok şekillikte bu karar yürütme zamanında verilmektedir.

## Normal Sınıf Üyelerine Çok Şekillilik Mekanizması Kullanılmadan İşaretçiler ile Erişim

```
class Square {       // Temel sinif
  protected:
    double edge;
  public:
    Square(double e):edge(e){  }  // temel sinif kurucusu
    double Area(){ return( 6.0 * edge ) ; }
};
class Cube : public Square { // Turetilmis sinif
  public:
    Cube(double e):Square(e){} // Turetilmis sinif kurucusu
    double Area(){ return( 6.0 * edge * edge ) ; }
};
```



```
void main(){
    Square S(2.0)                                   ;
    Cube   C(8.0)                                   ;
    Square *ptr                                     ;
    char      c                                     ;
    cout << “Square or Cube"; cin >> c              ;
    if (c==‘s') ptr=&S                              ;
       else     ptr=&C                              ;
    ptr→Area();          // which Area ???
}
```

Cube sınıfı Square sınıfından türetilmiştir. Her iki sınıfta Area() üye mesajını içermektedir. main() fonksiyonunda, Square ve Cube sınıflarından birer nesne ve Square sınıfına bir işaretçi tanımlanmıştır. Ardından türetilmiş sınıftan nesnenin adresi temel sınıfa işaret eden işaretçiye atanmıştır:

```
ptr = &C;
```

Bu geçerli bir atamadır

Aşağıdaki satır yürütüldüğünde

ptr→Area();

Square:: Area() fonksiyonu mu?

yoksa

Cube::Area() fonksiyonu mu?

çağırılır.



## İşaretçiler ile Erişilen Virtual Üye Fonksiyonları

Şimdi programda tek bir değişiklik yapalım: temel sınıftaki **Area**() fonksiyonunun önüne **virtual** anahtar sözcüğünü koyalım.

```
class Square {       // Temel sinif
  protected:
    double edge;
  public:
    Square(double e):edge(e){  }  // temel sinif kurucusu
     virtual double Area(){ return( 6.0 * edge ) ; }
};
class Cube : public Square { // Turetilmis sinif
  public:
    Cube(double e):Square(e){} // Turetilmis sinif kurucusu
    double Area(){ return( 6.0 * edge * edge ) ; }
};
```

```
void main(){
   Square S(2.0)                          ;
   Cube   C(8.0)                          ;
   Square *ptr                            ;
   char      c                            ;

   cout << “Square or Cube"; cin >> c     ;
   if (c==‘s') ptr=&S                     ;
      else     ptr=&C                     ;
   ptr→Area();
}
```



Şimdi ptr’nin içeriğine göre farklı fonksiyonlar çağırılacaktır. Fonksiyonlar ptr’nin tipine göre değil içeriğine göre çağırılmaktır. Bu şekilde çok şekillilik sağlanır. virtual anahtar sözcüğü, Area() fonksiyonunu çok şekilli olmasını sağlamaktadır.

Peki derleyici hangi fonksiyonu çağıracağını derleme zamanında nasıl biliyor? virtual anahtar sözcüğünü kullanmadığımız ilk örnekte bir sorun yoktu: ptr→Area(); Her zaman temel sınıftaki Area() fonksiyonu çalıştırılır. Ama ikinci örnekte, derleyici ptr işaretçisinin hangi tipten bir sınıfın nesnesine işaret ettiğini bilmektedir. ptr yürütme zamanında Square sınıfından yada Cube sınıfından bir nesneye işaret edebilir. Bu durumda hangi Area() çalıştırılır? Derleyici derleme esnasında bu kararı veremeyeceğinden, yürütme zamanında bu kararı verecek şekilde kodu düzenler.

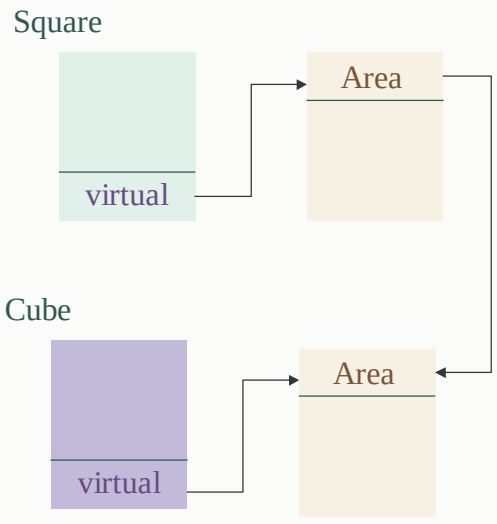
Yürütme zamanında, fonksiyon çağrısı oluştuğunda, derleyicinin yerleştirdiği bir kod, ptr nesne işaretçisinin hangi tipten bir nesneye işaret ettiğini algılar ve ilgili fonksiyonu çağırır: Square::Area() yada Cube::Area().

Buna late binding yada dynamic binding adı verilmektedir. Late binding bir miktar ek işlem yükü getirmektedir (yaklaşık %10 gibi). Ama bunun karşılığında yazdığımız programa büyük bir esneklik ve yetenek kazandırmaktadır.

Virtual tanımlaması içermeyen bir sınıfa ait nesne sadece üye alanaları bellekte yerleşik bulunur. İlgili nesne için bir üye fonksiyona çağrı yapıldığında, derleyici nesnenin adresini fonksiyona parametre olarak aktarır. Bu adresin this değişkeninde saklı bulunduğunu hatırlayınız. Derleyici, fonksiyonunun formal parametrelerine ek olarak (programcıya gizli biçimde) this işaretçisini ekler. Bu parametre her ilgili fonksiyon çarısında derleyici tarafından aktarılacaktır; this işaretçisi nesnenin üyeleri ile fonksiyonları arasındaki yegane bağlantıyı oluşturmaktadır.

virtual fonksiyonlar ise biraz daha karmaşık bir davranış göstermektedir. Türetilmiş sınıfta tanımlanan her virtual fonksiyon için derleyici, Virtual Table adı verilen bir dizi oluşturur. Square ve Cube sınıflarının her biri Virtual Table dizisine sahiptir. Bu dizilerde o sınıftaki her bir virtual fonksiyon için bir kayıt bulunur.





Bu örnekte, Square yada Cube sınıfından bir nesnenin virtual tanımlı fonksiyonuna bir çağrı yapıldığında, uygun üye fonksiyonunun çağrılması için gerekli işlemleri derleyici yerine, derleyicinin ürettiği bir kod gerçekleştirmektedir. Bu kod nesnenin virtual tablosunu tarayarak üye fonksiyonuna erişimi sağlamaktadır.



Sanal fonksiyon mekanizması sadece nesne işaretçileri ile kullanımda çalışır.

```
void main()
{
   Square S(4);
   Cube C(8);
   S.Area();
   C.Area();
}
```

## Nesne Bağlantılı Listesi ve Çok Şekillilik

Sanal fonksiyonların en çok kullanım alanı bulduğu uygulama bağlantılı nesne liste yapılarıdır:

```
class teacher{       // Base class
      char *name;
      int numOfStudents;
  public:
      teacher(char *new_name,int nos){
            name=new_name;numOfStudents=nos;}  // Constructor of base
      virtual void print(){ cout << "Name: "<< name << endl;
            cout << " Num of Students:"<< numOfStudents << endl;}
      teacher *next;
};
class principal : public teacher{        // Derived class
      char *SchoolName;
  public:
      principal(char *new_name,int nos, char *sn):teacher(new_name,nos){
                  SchoolName=sn;}
      void print(){ teacher::print();
                cout << " Name of School:"<< SchoolName << endl;}
};
```



```
void main(){
    teacher t1("Teacher 1",50);
    principal p1("Principal 1",40,"School1");
    teacher t2("Teacher 2",60);
    principal p2("Principal 2",100,"School2");
    teacher *listPtr;       // Pointer of the linked list
    /*** Construction of the list ***/
    listPtr=&p1;
    p1.next=&t1;
    t1.next=&t2;
    t2.next=&p2;
    p2.next=0L;
    /*** Printing all elements of the list ***/
    while (listPtr){
        listPtr->print();
        listPtr=listPtr->next;
    }
}
```

## Soyut Sınıflar ve Saf Sanal Fonksiyonlar

To write polymorphic functions wee need to have derived classes. But sometimes we don't need to create any base class objects, but only derived class objects. The base class exists only as a starting point for deriving other classes. This kind of base classes we can call an ***abstract class***, which means that no actual objects will be created from it. Abstract classes arise in many situations. A factory can make a sports car or a truck or an ambulance, but it can't make a generic vehicle. The factory must know the details about what *kind* of vehicle to make before it can actually make one. Similarly, you'll see sparrows, wrens, and robins flying around, but you won't see any generic birds.

Actually, a class is an abstract class only in the eyes of humans. The compiler is ignorant of our decision to make it an abstract class.

It would be nice if, having decided to create an abstract base class, I could instruct the compiler to actively *prevent* any class user from ever making an object of that class. This would give me more freedom in designing the base class because I wouldn't need to plan for actual objects of the class, but only for data and functions that would be used by derived classes. There is a way to tell the compiler that a class is abstract: You define at least one ***pure virtual function*** in the class.
A pure virtual function is a virtual function with no body. The body of the virtual function in the base class is removed, and the notation =0 is added to the function declaration.



## Örnek Uygulama : Shape.cpp

```
class Shape {
    protected :
      TCanvas *myCanvas ;
      TColor  color    ;
    public :
      Shape(TCanvas *canvas,TColor c):myCanvas(canvas),color(c){}
      virtual void draw()=0 ;
      virtual void setParameters()=0 ;
} ;
```

```
class rectangle : public Shape {
       int x1,y1,x2,y2 ;
     public :
       rectangle(TCanvas *canvas,TColor color,int X1,int Y1,int X2,int Y2)
          :Shape(canvas,color),x1(X1),y1(Y1),x2(X2),y2(Y2){}
       void draw(){
           /* … */
       }
       void setParameters(){
         /* … */
       }
} ;
```



```
  class circle : public Shape {
         int r , x , y ;
       public :
         circle(TCanvas *canvas,TColor color,int R,int X,int Y)
            :Shape(canvas,color),x(X),y(Y),r(R){}
         void draw(){
             /* … */
         }
         void setParameters(){
             /* … */
         }
  } ;
```

```
class ellipse : public Shape {
        int rx , ry , x , y ;
      public :
        ellipse(TCanvas *canvas,TColor color,int Rx,int Ry,int X,int Y)
           :Shape(canvas,color),x(X),y(Y),rx(Rx),ry(Ry){}
        void draw(){
            /* … */
        }
        void setParameters(){
            /* … */
        }
} ;
```



## Sanal Fonksiyonlar ve Kurucu Fonksiyonlar

Kurucu Fonksiyonlar Sanal olabilir mi?
Hayır.
Bir nesne yaratıldığında, derleyici bu nesnenin hangi sınıftan olduğunu bilmektedir. Bu nedenle, sanal kurucu fonksiyonlara ihtiyaç yoktur.

## Sanal Yokedici Fonksiyonlar

```
class Base {
        public:
                ~Base() { cout << "\nBase destructor"; }
        };
class Derived : public Base {
        public:
                ~Derv() { cout << "\nDerived destructor"; }
        };
void main(){
        Base* pb = new Derived;
        delete pb;
        cout << endl << "Program terminates.“ << endl ;
}
```



```
class Base {
        public:
                virtual ~Base() { cout << "\nBase destructor"; }
        };
class Derived : public Base {
        public:
                ~Derv() { cout << "\nDerived destructor"; }
        };
void main(){
        Base* pb = new Derived;
        delete pb;
        cout << endl << "Program terminates.“ << endl ;
}
```