Çok Şekillilik (Polymorphism)

Binnur Kurt
kurt@ce.itu.edu.tr

Bilgisayar Mühendisliği Bölümü
İstanbul Teknik Üniversitesi

Sınıf Yapısı

Kalıtım

5 Çok Şekillilik

Nesneye dayalı programlamanın üç temel kavramı :
1. Sınıflar,
2. Kalıtım,
3. **Çok Şekillilik** (C++'da ***sanal fonksiyonlar*** ile sağlanır)

5

Polymorphism

Çok Şekillilik

Gerçek hayattaki nesneler farklı sınıflardan olsalar da, yada farklı davranışları gerçekleştirseler de, bazı ortak işlevlere sahip olabilmektedirler. Örnek olarak **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 (**Area()**) tüm nesnelerde için çalışmalıdır. Çünkü her bir tip nesne kendi sınıfından nesnelerin alanını nasıl hesaplayacağını bilmektedir:

kare→Area() ;

daire→Area();

üçgen→Area();

5

Polymorphism

Çok Şekillilik

Area() işlevi farklı tipte nesnelerde farklı biçimler aldığından ***çok şekillik*** gösteren bir metodtur.

Çok şekillilik, genellikle birbirleri ile kalıtımla ilişkili sınıflar arasında gerçekleşir. C++'da çok şekilliğin anlamı, bir üye fonksiyona 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.

5 Polymorphism

Normal Sınıf Üyelerine Çok Şekillilik Meknaizması Kullanılmadan İşaretçiler ile Erişim

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


5 Polymorphism

```
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.

5 Polymorphism

Aşağıdaki satır yürütüldüğünde

ptr→Area();

Square:: Area() fonksiyonu mu?

yoksa

Cube::Area() fonksiyonu mu?

çağırılır.

5 Polymorphism

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

5 Polymorphism

```
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();
}
```


5 Polymorphism

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() fonksiyonunun 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.

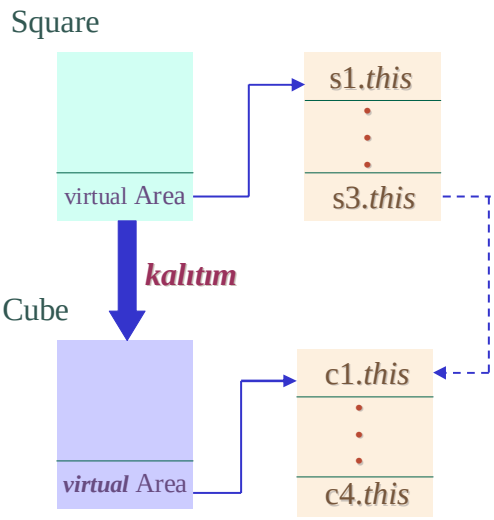
5 Polymorphism

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 bir biçimde) **this** işaretçisini ekler. Bu parametre her ilgili fonksiyon çağrısında derleyici tarafından aktarılacaktır; **this** işaretçisi nesnenin üyeleri ile fonksiyonları arasındaki yegane bağlantıyı oluşturmaktadır.

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

5 Polymorphism

```
Square s1(1),s2(4),s3(3);
Cube c1(2),c2(7) ;
Cube c3(8),c4(8) ;
Square *p        ;
...
p = &c2          ;
...
p->Area()        ;
...
p = &s3          ;
...
p->Area()        ;
```



5 Polymorphism

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

5 Polymorphism

Bunu Nesneler ile Denemeyin

Sanal fonksiyon mekanizması sadece nesne işaretçileri ile kullanımda çalışır.

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

5 Polymorphism

Uyarı

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

Burada **Square::Area()** *virtual değil*

5 Polymorphism

Nesne Bağlantılı Listesi ve Çok Şekillilik

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

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

```
void main(){
    Cube   c1(50);
    Square s1(40);
    Cube   c2(23);
    Square s2(78);
    Square *listPtr;       // Pointer of the linked list
    /*** Construction of the list ***/
    listPtr=&c1;
    c1.next=&s1;
    s1.next=&c2;
    c2.next=&s2;
    s2.next=0L;
    /*** Printing all elements of the list ***/
    while (listPtr){
        cout << listPtr->Area() << endl ;
        listPtr=listPtr->next;
    }
}
```


Soyut Sınıflar

Çok şekilliliğin aralarında kalıtım ilişkisi olan sınıflar arasında oluştuğunu gördük. Bu kalıtım ilişkisi, bazen ilgilendiğimiz problemin doğası gereği otomatik olarak kurulabilirken, bazen de böyle bir ilişki doğrudan oluşmaz. Bu durumda, sınıflar arasında böyle bir ilşkiyi yaratmak için ortak bir temel sınıf tasarlanır. Diğer tüm sınıflar bu ortak sınıftan türetilir. Bu çeşit sınıfları soyut sınıf olarak adlandırıyoruz. Gerçekte, program uzayında soyut sınıftan nesneler oluşmayacaktır. Soyut sınıflar programcının kafasında yarattığı bir sınıftır. Gerçekte problem uzayında soyut sınıftan nesnelere rastlanmaz.

5 Polymorphism

Pure Virtual Methods

Derleyiciye sanal sınıf yaratmak istediğimizi belirtmek için sınıf üyeleri içinde en az bir pure virtual method yazmamız yeterli omaktadır:
Bunun için gövdesi olmayan bir virtual metod tanımını hemen izleyen =0 yazılmalıdır.

5 Polymorphism

Örnek

```
class Shape{          // Abstract base class
  protected:
     int x,y;
  public:
    Shape(int x_in,int y_in){ x=x_in; y=y_in;}  // Constructor
    virtual void draw()=0;      // pure virtual function
};
class Line:public Shape{       // Line class
  protected:
     int x2,y2;            // End coordinates of line
  public:
     Line(int x_in,int y_in,int x2_in,int y2_in):Shape(x_in,y_in){
          x2=x2_in;
          y2=y2_in;
     }
     void draw(){ line(x,y,x2,y2); }      // virtual draw function
};
```

5 Polymorphism

```
class Rectangle:public Line{          // Rectangle class
  public:
    Rectangle(int x_in,int y_in,int x2_in, int y2_in):
                                              Line(x_in,y_in,x2_in,y2_in){}
    void draw(){ rectangle(x,y,x2,y2); }     // virtual draw
};
class Circle:public Shape{    // Circle class
  protected:
    int radius;
  public:
    Circle(int x_cen,int y_cen,int r):Shape(x_cen,y_cen){
      radius=r;
    }
    void draw() { circle(x,y, radius); }       // virtual draw
};
/* A function to draw different shapes ***/
void show(Shape &shape){    // Which draw function will be called?
  shape.draw();    // It 's unknown at compile-time
}
```


5 Polymorphism

```
void main()
{
  int gdriver = DETECT, gmode, errorcode;
  initgraph(&gdriver, &gmode, "\\tc\\bgi"); //To graphics mode
  Line Line1(1,1,100,250);
  Circle Circle1(100,100,20);
  Rectangle Rectangle1(30,50,250,140);
  Circle Circle2(300,170,50);
  show(Circle1);
  getch();
  show(Line1);
  getch();
  show(Circle2);
  getch();
  show(Rectangle1);
  getch();
  closegraph();
}
```

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