8 Polymorphism

Content

- Polymorphism
- Virtual Members
- Abstract Class

Polymorphism

- There are three major concepts in object-oriented programming:
  1. Classes,
  2. Inheritance,
  3. Polymorphism, which is implemented in C++ by virtual functions.
- In real life, there is often a collection of different objects that, given identical instructions (messages), should take different actions. Take teacher and principal, for example.
- Suppose the minister of education wants to send a directive to all personnel: “Print your personal information!” Different kinds of staff (teacher or principal) have to print different information. But the minister doesn’t need to send a different message to each group. One message works for everyone because everyone knows how to print his or her personal information.

Polymorphism

- Polymorphism means “taking many shapes”. The minister’s single instruction is polymorphic because it looks different to different kinds of personnel.
- Typically, polymorphism occurs in classes that are related by inheritance. In C++, polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
- This sounds a little like function overloading, but polymorphism is a different, and much more powerful, mechanism. One difference between overloading and polymorphism has to do with which function to execute when the choice is made.
- With function overloading, the choice is made by the compiler (compile-time). With polymorphism, it’s made while the program is running (run-time).

Normal Member Functions Accessed with Pointers

```
class Square {       // Base Class
  protected:
    double edge;
  public:
    Square(double e):edge(e){  } //Base class constructor
    double area(){ return( edge * edge ) ; }
};
```

```
class Cube : public Square { // Derived Class
  public:
    Cube(double e):Square(e){} // Derived class cons.
    double area(){ return( 6.0 * edge * edge ) ; }
};
```

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

- ptr = &C;
- Remember that it's perfectly all right to assign an address of one type (Derived) to a pointer of another (Base), because pointers to objects of a derived class are type compatible with pointers to objects of the base class.
- Now the question is, when you execute the statement

  ptr->area();

what function is called? Is it Square::area() or Cube::area()?

Virtual Member Functions Accessed with Pointers

Let's make a single change in the program: Place the keyword **virtual** in front of the declaration of the area() function in the base class.

```
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 ) ; }
};
```

square.cpp

```
int 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();
}
```

Virtual Member Functions Accessed with Pointers

The function in the base class (Teacher) is executed in both cases. The compiler ignores the **contents** of the pointer ptr and chooses the member function that matches the **type** of the pointer.

Let's make a single change in the program: Place the keyword **virtual** in front of the declaration of the print() function in the base class.

```
class Teacher{                                  // Base class
      string *name;
      int numOfStudents;
  public:
      Teacher(const string &, int);             // Constructor of base
      virtual void print() const;               // A virtual (polymorphic) function
};

class Principal : public Teacher{               // Derived class
      string *SchoolName;
  public:
      Principal(const string &,  int , const string &);
      void print() const;                       // It is also virtual (polymorphic)
};
```

Late Binding

- Now, different functions are executed, depending on the contents of ptr. Functions are called based on the contents of the pointer ptr, not on the type of the pointer. This is polymorphism at work. I've made print() polymorphic by designating it virtual.
- How does the compiler know what function to compile? In e81.cpp, the compiler has no problem with the expression
- ptr->print();
- It always compiles a call to the print() function in the base class. But in e82.cpp, the compiler doesn't know what class the contents of ptr may be a pointer to. It could be the address of an object of the Teacher class or the Principal class. Which version of print() does the compiler call? In fact, at the time it's compiling the program, the compiler doesn't know what to do, so it arranges for the decision to be deferred until the program is running.

Late Binding

- At runtime, when the function call is executed, code that the compiler placed in the program finds out the type of the object whose address is in ptr and calls the appropriate print() function: Teacher::print() or Principal::print(), depending on the class of the object.
- Selecting a function at runtime is called **late binding** or **dynamic binding**. (Binding means connecting the function call to the function.)
- Connecting to functions in the normal way, during compilation, is called *early binding* or *static binding*. Late binding requires a small amount of overhead (the call to the function might take something like 10 percent longer) but provides an enormous increase in power and flexibility.

How It Works

- Remember that, stored in memory, a normal object—that is, one with no virtual functions—contains only its own data, nothing else.
- When a member function is called for such an object, the compiler passes to the function the address of the object that invoked it. This address is available to the function in the this pointer, which the function uses (usually invisibly) to access the object's data.
- The address in this is generated by the compiler every time a member function is called; it's not stored in the object and does not take up space in memory.
- The ***this*** pointer is the only connection that's necessary between an object and its normal member functions.

How It Works

- With virtual functions, things are more complicated. When a derived class with virtual functions is specified, the compiler creates a table—an array—of function addresses called the **virtual table**.
- The Teacher and Principal classes each have their own virtual table. There is an entry in each virtual table for every virtual function in the class. Objects of classes with virtual functions contain a pointer to the virtual table of the class. These object are slightly larger than normal objects.
- In the example, when a virtual function is called for an object of Teacher or Principal, the compiler, instead of specifying what function will be called, creates code that will first look at the object's virtual table and then uses this to access the appropriate member function address. Thus, for virtual functions, the object itself determines what function is called, rather than the compiler.

Example: Assume that the classes Teacher and Principal contain two virtual functions.

```
class Teacher{                                  // Base class
      string *name;
      int numOfStudents;
  public:
      virtual void read();                      // Virtual function
      virtual void print() const;               // Virtual function
};

class Principal : public Teacher{               // Derived class
     string *SchoolName;
  public:
     void read();                               // Virtual function
     void print() const;                        // Virtual function
};
```

Virtual Table of Teacher

&Teacher::read
&Teacher::print

Virtual Table of Principal

&Principal::read
&Principal::print

Objects of Teacher and Principal will contain a pointer to their virtual tables.

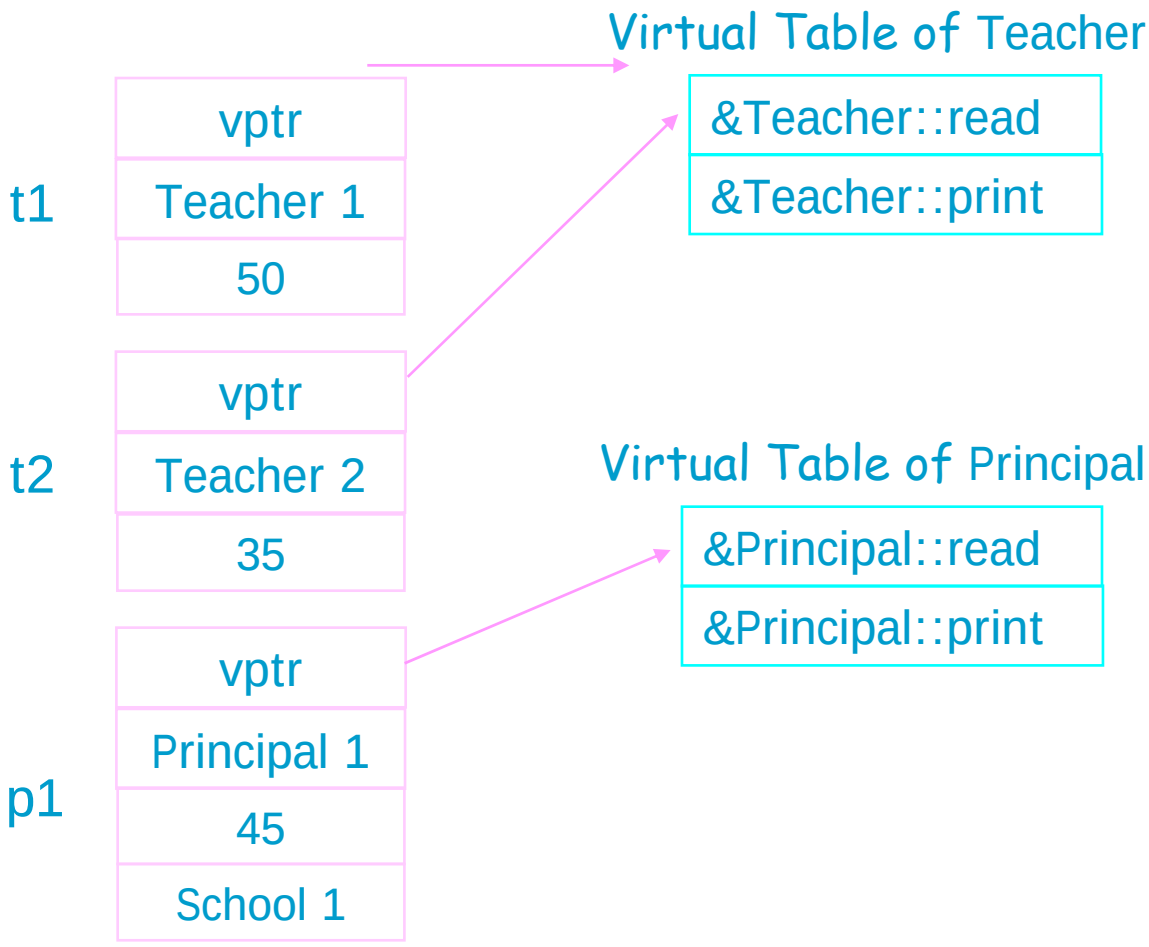
```
int main(){
   Teacher t1("Teacher 1", 50);
   Teacher t2("Teacher 2", 35);
   Principal p1("Principal 1", 45 , "School 1");
   :
}
```

MC68000-like assembly counterpart of the statement ptr->print(); Here ptr contains the address of an object.

```
move.l    ptr, this  ; this to object
movea.l   ptr, a0    ; a0 to object
movea.l   (a0), a1 ; a1<-vptr
jsr       4(a1)      ; jsr print
```

If the print() function would not a virtual function:

```
move.l    ptr, this  ; this to object
jsr       teacher_print
   or
jsr       principal_print
```



Don’t Try This with Objects

Be aware that the virtual function mechanism works only with pointers to objects and, with references, not with objects themselves.

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

Calling virtual functions is a time-consuming process, because of indirect call via tables. Don’t declare functions as virtual if it is not necessary.

Warning

```
class Square {       // Base
  protected:
     double edge;
  public:
     Square(double e):edge(e){  }  // Base Class Constructor
      virtual double Area(){ return( edge * edge ) ; }
};
class Cube : public Square { // Derived Class
  public:
     Cube(double e):Square(e){} // Derived Class Constructor
     double Area(){ return( 6.0 * Square::Area() ) ; }
};
```

Here, **Square::Area()** *is not virtual*

Homogeneous Linked Lists and Polymorphism

Most frequent use of polymorphism is on collections such as linked list:

```
class Square {
  protected:
     double edge;
  public:
     Square(double e):edge(e){  }
     virtual double area(){ return( edge * edge ) ; }
     Sqaure *next ;
};
class Cube : public Square {
  public:
     Cube(double e):Square(e){}
     double area(){ return( 6.0 * edge * edge ) ; }
};
```

```
int main(){
    Circle   c1(50);
    Square s1(40);
    Circle   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;
    while (listPtr){ // Printing all elements of the list
        cout << listPtr->Area() << endl ;
        listPtr=listPtr->next;
    }
}
```

Abstract Classes

► 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 are called 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.

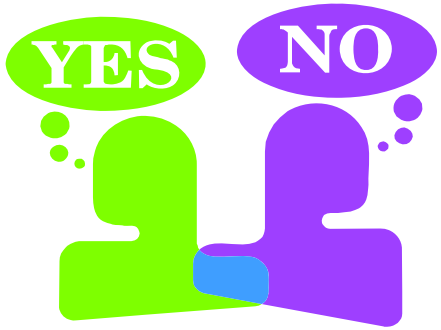
► Actually, a class is an abstract class only in the eyes of humans.

Pure Virtual Classes

► 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.

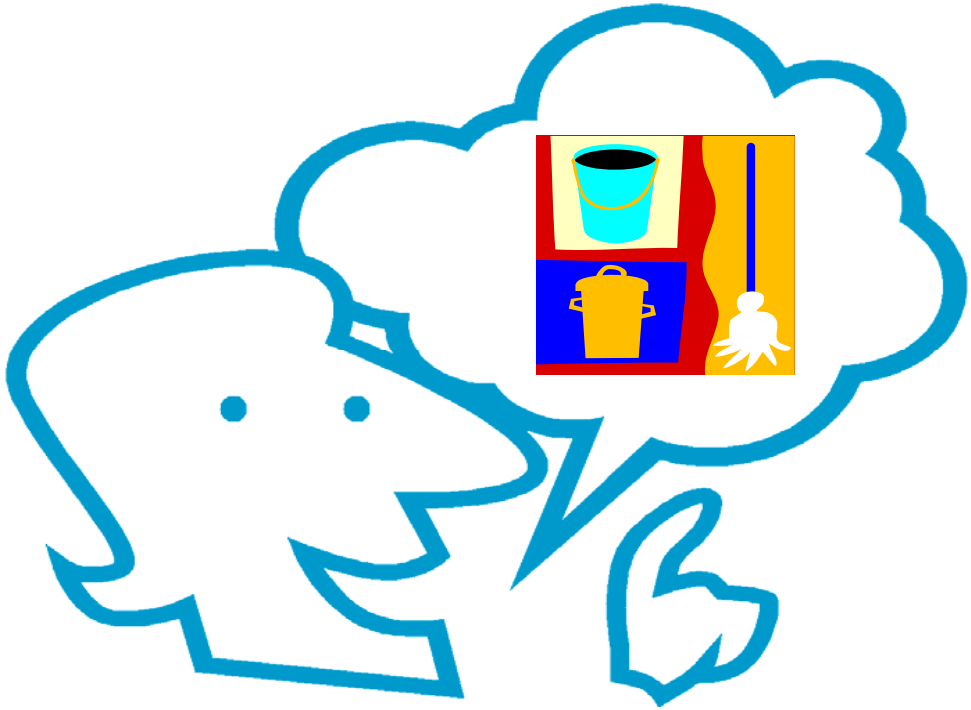
► A pure virtual function is a virtual function with no body. The body of the virtual function in the base class is removed, and the notation **=0** is added to the function declaration.

Are they the same or different?



►Not in the real world, but in our thoughts as an **abstraction** classification.

►A “Cleaning Utensil” does not exist, but specific kinds do!



Example in Visual C++ 6

```
class CGenericShape{          // Abstract base class
   protected:
      int x,y;
      CGenericShape *next ;
   public:
         CGenericShape(int x_in,int y_in,
                              CGenericShape *nextShape){
                  x=x_in;
                  y=y_in;
                  next = nextShape ;
          }  // Constructor
          CGenericShape* operator++(){return next;}
          virtual void draw(HDC)=0;      // pure virtual function
};
```

```
class CLine:public CGenericShape{       // Line class
  protected:
     int x2,y2;              // End coordinates of line
  public:
     CLine(int x_in,int y_in,int x2_in,int y2_in,
              CGenericShape *nextShape)
              :CGenericShape(x_in,y_in,nextShape){
             x2=x2_in;
             y2=y2_in;
     }
     void draw(HDC hdc){ // virtual draw function
           MoveToEx(hdc,x,y,(LPPOINT) NULL);
           LineTo(hdc,x2,y2); }
};
```

```
class CRectangle:public CLine{  // Rectangle class
  public:
    CRectangle (int x_in,int y_in,int x2_in,int y2_in,
                    CGenericShape *nextShape)
                   :CLine(x_in,y_in,x2_in,y2_in,nextShape)
                   { }
    void draw(HDC hdc){// virtual draw
         Rectangle(hdc,x,y,x2,y2);
    }
};
```

```
class CCircle:public CGenericShape{    // Circle class
  protected:
     int radius;
  public:
     CCircle (int x_cen,int y_cen,int r,
                    CGenericShape *nextShape)
                    :CGenericShape(x_cen,y_cen,nextShape)
     {
       radius=r;
     }
     void draw(HDC hdc) { // virtual draw
         Ellipse(hdc,x-radius,y-radius,x+radius,y+radius);
     }
};
```

```
void ShowShapes(CGenericShape &shape,HDC hdc)
{
      CGenericShape *p = &shape ;
        // Which draw function will be called?
   while (p!=NULL){
          p->draw(hdc); // It 's unknown at compile-time
          p = ++*p ;
          Sleep(100);
    }
}
```

PolyDraw.dsw

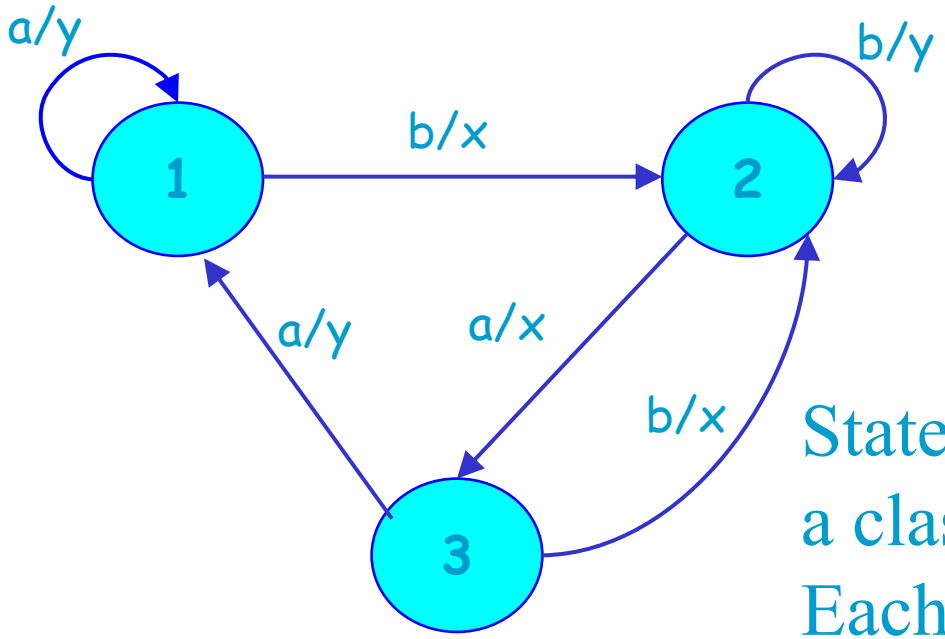
```
PAINTSTRUCT ps;
HDC hdc;

CLine Line1(50,50,150,150,NULL);
CLine Line2(150,50,50,150,&Line1) ;
CCircle Circle1(100,100,20,&Line2);
CCircle Circle2(100,100,50,&Circle1);
CRectangle Rectangle1(50,50,150,150,&Circle2);

switch (message) {
                case WM_PAINT:
                  hdc = BeginPaint (hwnd, &ps);
                      ShowShapes (Rectangle1,hdc);
                   EndPaint (hwnd, &ps);
                return 0;
```

A Finite State Machine (FSM) Example

State : { 1 , 2 , 3 }
Input : { a, b }, x to exit
Output : { x , y }



States of the FSM are defined using a class structure.
Each state is derived from the same base class.

```
class State{                                                    // Base State (Abstract Class)
   protected:
     State * const next_a, * const next_b;                      // Pointers to next state
     char output;
   public:
     State(State &  a, State &  b):next_a(&a), next_b(&b) { }
     virtual State* transition(char)=0;                         // pure virtual function
};

class State1:public State{                                      // *** State1 ***
  public:
     State1(State &  a, State &  b):State(a, b) { }
     State* transition(char);
};
class State2:public State{                                      // *** State2 ***
  public:
     State2(State &  a, State &  b):State(a, b) { }
     State* transition(char);
};
class State3:public State{                                      // *** State3 ***
  public:
      State3(State &  a, State &  b):State(a, b) { }
      State* transition(char);
};
```

The transition function of each state defines the behavior of the FSM. It takes the input value as argument, examines the input, produces an output value according to the input value and returns the address of the next state.

```
State* State1::transition(char input)
{
    switch(input){
    case 'a':  output = 'y';
                return next_a;
    case 'b': output = 'x';
               return next_b;
    default : cout << endl << "Undefined input";
                cout << endl << "Next State: Unchanged";
              return this;
   }
}
```

The FSM in our example has three states.

```
class FSM{                                  // Finite State Machine
    State1 s1;
    State2 s2;
    State3 s3;
    State *current;                         // points to the current state
 public:
    FSM() : s1(s1,s2), s2(s3,s2), s3(s1,s2), current(&s1) { }   //Starting state is State1
    void run();
};
void FSM::run()  {
   char in;
   do {
      cout << endl << "Give the input value (a or b;  x: EXIT) ";
      cin >> in;
      if (in != 'x')
         current = current->transition(in);      // Polymorphic function call
      else
         curent = 0;                            // EXIT
    }  while(current);
}
```

The transition function of the current state is called.
Return value of this function determines the next state of the FSM.

Virtual Constructors?

- Can constructors be virtual?

  No, they can't be.

- When you're creating an object, you usually already know what kind of object you're creating and can specify this to the compiler. Thus, there's not a need for virtual constructors.

- Also, an object's constructor sets up its virtual mechanism (the virtual table) in the first place. You don't see the code for this, of course, just as you don't see the code that allocates memory for an object.

- Virtual functions can't even exist until the constructor has finished its job, so **constructors can't be virtual**.

Virtual Destructors

- Recall that a derived class object typically contains data from both the base class and the derived class.
- To ensure that such data is properly disposed of, it may be essential that destructors for both base and derived classes are called.

Virtual Destructors

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

Virtual Destructors

- But the output is
  Base Destructor
  Program terminates
- In this program bp is a pointer of Base type. So it can point to objects of Base type and Derived type. In the example, bp points to an object of Derived class, but while deleting the pointer only the Base class destructor is called.
- This is the same problem you saw before with ordinary (nondestructor) functions. If a function isn't virtual, only the base class version of the function will be called when it's invoked using a base class pointer, even if the contents of the pointer is the address of a derived class object. Thus in e85.cpp, the Derived class destructor is never called. This could be a problem if this destructor did something important.

To fix this problem, we have to make the base class **destructor virtual**.

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