Polymorphism

Feza BUZLUCA
Istanbul Technical University
Computer Engineering Department
http://faculty.itu.edu.tr/buzluca
http://www.buzluca.info

1

Overview

- Introduction
- Polymorphism
- Normal Member Functions Accessed with Pointers
- Virtual Member Functions Accessed with Pointers
- Late Binding
- Abstract Classes
- Virtual Destructors

2

OOP

There are three major concepts in object-oriented programming:

1. Encapsulation (Classes)

   Data abstraction, information hiding (public: interface, private: implementation)

2. Inheritance

   Is-a relationship, generalization-specialization, reusability

3. Polymorphism

   Run-time decision for function calls (dynamic method binding)

Polymorphism

- 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 (teachers and principals) have to print different kinds of information
- However, the minister does not need to send a different message to each group
- One message works for everyone because everyone knows how to print his personal information
- Besides, the minister does not need to know the type of person to whom the message is to be sent

Polymorphism

- Polymorphism means "taking many shapes"
- The minister's single instruction is polymorphic because it looks different to different kinds of personnel
- The minister does not need to know the type of the person to whom the message is sent
- 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 gets the message

The Sender of the Message Does Not Need to Know the Type of the Receiving Object

- This sounds a little like function overloading, but polymorphism is a different, and much more powerful, mechanism
- One difference between overloading and polymorphism that has to do with which function to execute is when the choice is made
  - With function overloading, the choice is made by the compiler (compile-time)
  - With polymorphism, it is made while the program is running (run-time)

Normal Member Functions Accessed with Pointers

- Example of what happens when
  - base class and derived classes all have functions with same name
  - these functions are accessed using pointers, but without using virtual functions (without polymorphism)

```
class Teacher {                          // base class
   string name;
   int numOfStudents;
 public:
   Teacher( const string &, int );       // constructor of base
   void print() const;
};

class Principal : public Teacher {       // derived class
   string SchoolName;
 public:
   Principal( const string &,  int , const string & );
   void print() const;
};
```

- Both classes have a function with the same name: print
- However, these functions are not virtual (not polymorphic)

Normal Member Functions Accessed with Pointers

```
// show is a function that operates on Teachers and Principals
void show(const Teacher * tp)
{
   tp->print();       // which print
}

// only to test the show function
int main()
{
   Teacher t1( "Teacher 1", 50 );
   Principal p1( "Principal 1", 40, "School" );
   Teacher *ptr;
   char c;
   cout << "Teacher or Principal " ;
   cin >> c;
   if ( c == 't')  ptr = &t1;
   else    ptr = &p1;
   show( ptr );                // which print ??
    :
```

See Example e81.cpp

- Now the question is, when you execute the statement

```
tp->print();
```

what function is called? Is it Teacher::print() or Principal::print()?

Normal Member Functions Accessed with Pointers

- Principal class is derived from class Teacher
- Both classes have a member function with the name print()
- In main(), the program defines a pointer to class Teacher
- The main() program
  - sometimes puts the address of the Teacher object, and
  - other times the address of a derived class object (Principal)

  in the base class pointer as in the line

```
ptr = &p1;
```

- Remember that it is all right to assign an address of derived type to a pointer of the base
  - This address is sent to the function show() over a pointer to base

9

Normal 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

10

Virtual Member Functions Accessed with Pointers

- Let us 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;                           // also virtual (polymorphic)
};
```

See Example e82.cpp

Virtual Member Functions Accessed with Pointers

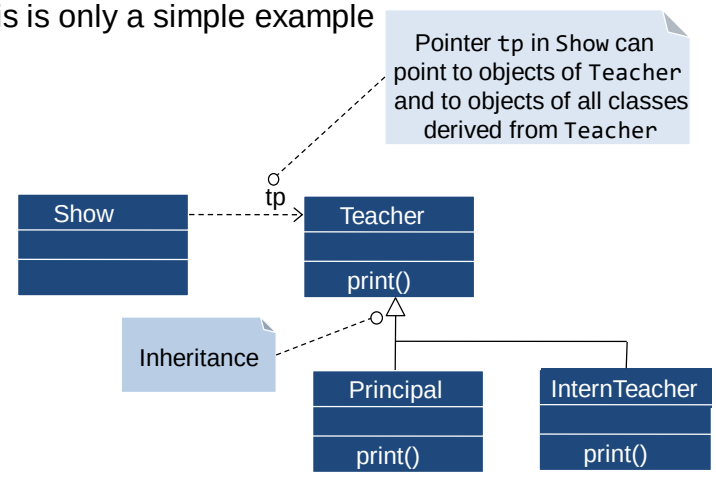
- Now, different functions are executed, depending on the contents of ptr
- Functions are called based on the contents of the pointer, not on the type of the pointer
- This is polymorphism at work
- We have made print() polymorphic by designating it virtual

Benefits of Polymorphism

- Polymorphism provides flexibility
- In our example, the show() function:
  - has no information about the type of object pointed by the input parameter
  - can have the address of an object of any class derived from the Teacher class
  - can operate on any class derived from the Teacher
- If we add a new teacher type (a new class) to the system (e.g., InternTeacher), we do not need to change the show function
- The same thing is true if we discard a class derived from Teacher from the system

UML Diagram of the Design

- We will look at the The Unified Modeling Language (UML) in Lecture 9
- This is only a simple example



Pass-by-Reference Version

- Note: In C++, we prefer to use references instead of pointers when passing parameters
- We can write the same program as follows:

```
// show is a function that operates on Teachers and Principals
void show (const Teacher &tp)
{
  tp.print();       // which print
}
// Only to test the show function
int main()
{
  Teacher t1( "Teacher 1", 50 );
  Principal p1( "Principal 1", 40, "School" );
  char c;
  cout << "Teacher or Principal " ;
  cin >> c;
  if (c == 't') show(t1);
  else  show(p1);
   :
```


Late Binding

- How does the compiler know what function to compile?
- In e81.cpp, without polymorphism, the compiler has no problem with the expression tp->print();
  - It always compiles a call to the print() function in the base class
- However, in e82.cpp, the compiler does not know what class the contents of `tp` 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?

Late Binding

- Which version of print() does the compiler call?
- In fact, when compiling the program, the compiler does not "know" which function to call
- So, instead of a simple function call, it places a special piece of code there
- 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 tp and calls the appropriate print() function: Teacher::print() or Principal::print()

Late Binding

- 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 programming power and flexibility

Late Binding: 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 address of the object is available in the `this` pointer, which the member 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 is not stored in the object
- 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

Late Binding: How It Works

- In the example `e82.cpp`, 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 (`vptr`) of the class
- These objects 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 `vptr` and then uses this to access the appropriate member function address in the class `vtable`
- Thus, for virtual functions, the object itself determines what function is called, rather than the compiler

Late Binding:Example

- Example: Assume `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

Late Binding:Example

- Each object of `Teacher` and `Principal` will contain a pointer to the virtual table of the class

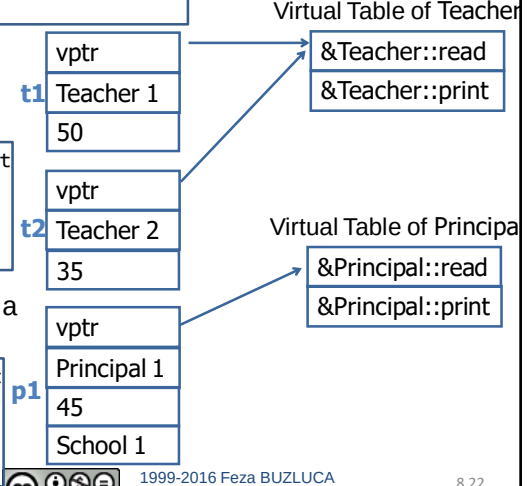
```
int main() {
   Teacher t1( "Teacher 1", 50 );
   Teacher t2( "Teacher 2", 35 );
   Principal p1( "Principal 1", 45 , "School 1" );
   :
}
```

- 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 was not a virtual function:

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



Do Not 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()
{
   Teacher t1("Teacher 1", 50);
   Principal p1("Principal 1", 40, "School");
   t1.print();                      // not polymorphic
   p1.print();                      // not polymorphic
   return 0;
}
```

- Calling virtual functions is a time-consuming process because of indirect calls via tables
- Do not declare functions as virtual if it is not necessary

Linked List of Objects and Polymorphism

- The most common ways to use virtual functions are with an array of pointers to objects and linked lists of objects
- Examine the example: **See Example e83.cpp**
- Remember: there is a list class in the C++ standard library
- You do not need to write a class to define linked lists

Abstract Classes

- To write polymorphic functions, we need to have derived classes
- However, we sometimes do not 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
- We call this kind of base class 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, a truck, or an ambulance, but it cannot make a generic vehicle
  - The factory must know the details about what kind of vehicle to make before it can actually make one

Pure Virtual Functions

- It would be nice if, having decided to create an abstract base class, we could instruct the compiler to prevent any class user from ever making an object of that class
- This would give us more freedom in designing the base class because we would not 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: We 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

A Graphics Example

```
class GenericShape {                    // abstract base class
 protected:
   int x, y;
 public:
   GenericShape( int x_in, int y_in ){ x = x_in; y = y_in; }   // constructor
   virtual void draw() const =0;        // pure virtual function
};
class Line: public GenericShape {       // Line class
 protected:
   int x2, y2;                          // end coordinates of line
 public:
   Line(int x_in,int y_in,int x2_in,int y2_in):GenericShape(x_in,y_in),  x2(x2_in),y2(y2_in)
   { }
   void draw() const;                   // virtual draw function
};
void Line::draw() const
{
   cout << "Type: Line" << endl;
   cout << "Coordinates of end points: " << "X1=" << x <<  " ,Y1=" << y <<
                                        " ,X2=" << x2 <<  " ,Y2=" << y2 << endl;
}
```


A Graphics Example

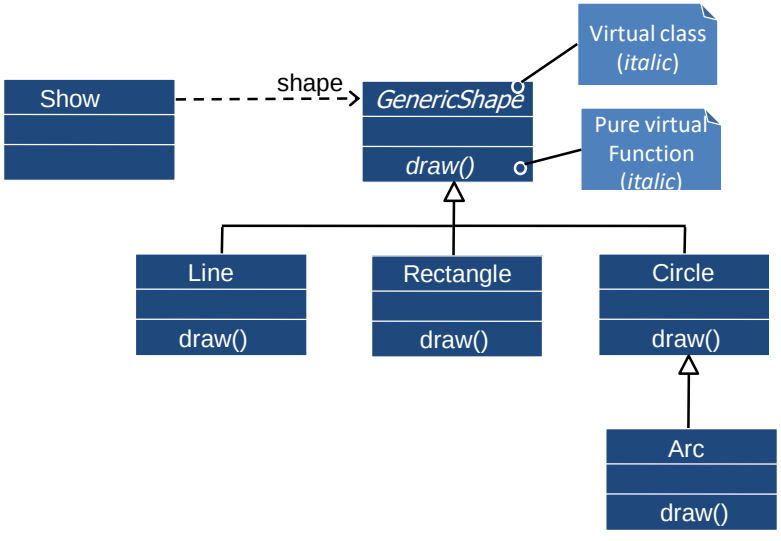
```
class Rectangle:public GenericShape{       // Rectangle class
 protected:
   int x2,y2;                               // coordinates of 2nd corner point
 public:
   Rectangle(int x_in,int y_in,int x2_in,int y2_in):GenericShape(x_in,y_in),
                                            x2(x2_in),y2(y2_in)
    { }
   void draw()const;                        // virtual draw
};
void Rectangle::draw()const
{
   cout << "Type: Rectangle" << endl;
   cout << "Coordinates of corner points: " << "X1=" << x << " ,Y1=" << y <<
                                              " ,X2=" << x2 <<  " ,Y2=" << y2 << endl;
}

class Circle:public GenericShape{          // Circle class
 protected:
   int radius;
 public:
   Circle(int x_cen,int y_cen,int r):GenericShape(x_cen,y_cen),  radius(r)
   { }
   void draw() const;                       // virtual draw
};
```

A Graphics Example

```
void Circle::draw() const
{
   cout << "Type: Circle" << endl;
   cout << "Coordinates of center point: " << "X=" << x <<  " ,Y=" << y << endl;
   cout << "Radius: " << radius << endl;
}
// A function to draw different shapes
void show(const Generic_shape &shape)  // can take references to different shapes
{                                      // which draw function will be called?
   shape.draw();                       // unknown at compile time
}
int main()                             // a main function to test the system
{
   Line line1( 1, 1, 100, 250 );
   Circle circle1( 100, 100, 20 );
   Rectangle rectangle1( 30, 50, 250, 140 );
   Circle circle2( 300, 170, 50 );
   show( circle1 );            // show function can take different shapes as argument
   show( line1 );
   show( circle2 );
   show( rectangle1 );
   return 0;
}
```

See Example e84a.cpp

A Graphics Example: UML Diagram of the Design



A Graphics Example: Add New Shape

- If we write a class for a new shape by deriving it from an existing class, we do not need to modify the show function
- This function can also show the new shape
- For example, we can add an Arc class to our graphics library, which will not affect the show function

```
class Arc: public Circle {                  // Arc class
 protected:
   int sa, ea;                              // start and end angles
 public:
   Arc(int x_cen,int y_cen,int r, int a1, int a2): Circle(x_cen,y_cen,r),
                                                   sa(a1),ea(a2)
     {}
   void draw() const;                       // virtual draw
};

void Arc::draw() const
{
   cout << "Type: Arc" << endl;
   cout << "Coordinates of center point: " << "X=" << x <<  " ,Y=" << y << endl;
   cout << "Radius: " << radius << endl;
   cout << "Start and end angles: " << "SA=" << sa <<  " ,EA=" << ea << endl;
}
```

See Example e84b.cpp

Virtual Constructors

- Can constructors be virtual?
- No, they cannot be virtual
- When we are creating an object, we usually already know what kind of object we are creating and can specify this to the compiler
- Thus, there is not much of a need for virtual constructors
- Also, an object's constructor sets up its virtual mechanism (the virtual table) in the first place
- We do not see the code for this, of course, just as we do not see the code that allocates memory for an object
- Virtual functions cannot even exist until the constructor has finished its job, so constructors cannot be virtual

Virtual Destructors

See Example e85.cpp

- 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
- But the output of e85.cpp 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


```
class Base {
 public:
   int data;
   virtual void f(){}
   ~Base() { cout << "Base destructor" << endl; }   // Destructor is not virtual
};
class Base2 {
 public:
   int data;
   virtual void f(){}
   ~Base2() { cout << "Base destructor" << endl; }   // Destructor is not virtual
};
class Derived : public Base, public Base2 {
 public:
   int data;
   void f(){}
   ~Derived() { cout << "Derived destructor" << endl; } // Non-virtual
};
int main()
{
   Base* pb;                    // pb can point to objects of Base and Derived
   pb = new Derived;            // pb points to an oject of Derived
   delete pb;
   cout << "Program terminates" << endl;
   return 0;
}
```

See Example e85.cpp

Virtual Destructors

- This is the same problem we saw before with ordinary (nondestructor) functions
- If a function is not virtual, only the base class version of the function will be called when it is 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

Rewrite: Example e85.cpp