Lecture 8

Polymorphism

Topics

- Member Functions with Pointers
- Polymorphic Calling
- Abstract Classes and Pure Virtual Functions

Polymorphism

- In object-oriented programming, polymorphism (many-shapes) means that a call to an overridden virtual member function will cause a different function to be executed depending on the **type of object** that is called.
- Occurs only in classes that are related by inheritance**.**
- The polymorphism allows a run-time decision (not compile-time) for calling an overridden member function (either base class function or derived class function).
- In general, the polymorphism term refers to the run-time polymorphism.

Example : Teacher and Principal classes

- Suppose the minister of education wants to send a general directive to all employees: “Print the employee information.”
- Different kinds of employees (Teacher or Principal) have to print different information.
- But the minister doesn’t need to send a different directive to each group.

Member Functions that accessed with Pointers

The base class Teacher and derived class Principal have **print** functions with the same name (member function overriding).

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

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


- In main program, a pointer to Teacher is defined.
- The non-member **show** function is called sometimes with a Teacher object, sometimes with a Principal object.

```
// Test the show function
int main()  {
   Teacher    t1 ("Teacher1", 100);
   Principal   p1 ("Principal1", 150, "School1");
   Teacher  *ptr;  // Pointer to base class

   char  choice;
   cout << "Teacher or Principal (t , p)  ?  :  " ;
   cin >> choice;

   if  (choice   ==   't')   // Run-time decision
      ptr   =   & t1;   // Teacher
   else
      ptr   =   & p1;   // Principal

    show ( ptr );  // Calling the nonmember show function
}
```

Nonmember show function

- The **show** function is written as a non-member function, to operate on Teacher and Principal classes.

```
void  show  (Teacher *  tp)
// Parameter tp is pointer to base class
{
    tp -> print ();
}
```

- Problem: When tp->print() is executed, always Teacher::print() will be called. The Principal::print() will not be called, even when tp pointer points to a Principal.

- Solution: In order to fix the mentioned problem, the print function of Teacher class (base class) should be declared as virtual. So that the print function becomes polymorphic.

Virtual Member Functions that accessed with Base Pointers

- Place the keyword **virtual** in front of the **print** function in **base class Teacher**.
- Now, different print functions are executed, depending on the contents of ptr pointer in main program.
- The print() function is defined **polymorphic** (run-time) by declaring it virtual.

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

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

Benefit of Polymorphism

- Polymorphism provides flexibility when calling an overridden member function (such as print).
- In the example, the non-member **show** function has no information about the exact type of object (base or derived) pointed by the input parameter.
- The show function can operate on Teacher class, and any other class derived from **Teacher** class.
- If a new class is added to program, for example **InternTeacher**, there is no need to change the show function.

Nonmember show function with const reference argument

- In C++ it is preferred to use **constant references** instead of pointers, when passing parameters to functions.
- The same program can be rewritten as follows.

```
// Show function operates on Teachers and Principals
void  show  (const  Teacher  &   tp)
// Parameter tp is reference to base class
{
   tp . print ();  //polymorphic calling
}
```

```
int main()  {
   Teacher    t1 ("Teacher1", 100);
   Principal   p1 ("Principal1", 150, "School1");
   char  choice;
   cout << "Teacher or Principal (t , p)  ?  :  " ;
   cin >> choice;
   if  (choice  ==  't')   // Run-time decision
         show (t1);    // Teacher
   else  show (p1);    // Principal
}
```

Topics

- Member Functions with Pointers
- Polymorphic Calling
- Abstract Classes and Pure Virtual Functions

Difference between Polymorphic and Non-polymorphic calling of functions

The virtual function mechanism works **only with pointers** to objects and with references, not with objects themselves.

```
int main()
{
    Teacher    t1 ("Teacher1", 100);
    Principal  p1 ("Principal1", 150 , "School1");

    t1 . print ();       // not polymorphic call
    p1 . print ();       // not polymorphic call

    Teacher * ptr;  // pointer to base class (allows polymorphism)

    ptr   =   & t1;
    ptr -> print ();     //polymorphic call

    ptr   =   & p1;
    ptr -> print ();     //polymorphic call
}
```

Topics

- Member Functions with Pointers
- Polymorphic Calling
- Abstract Classes and Pure Virtual Functions

Abstract Classes

- To write polymorphic member functions, programmer should write **derived classes**.
- Sometimes there may be no need to create any base class objects, but only derived class objects are required. The base class exists only as a beginning point for deriving other classes.
- Those kind of **base classes** are considered as ***abstract classes***, which means that no actual objects can be created from it.
- An abstract class can not be used itself to directly define objects of that class.
- In order to use an abstract class, some other classes must be derived from that abstract class.

Pure Virtual Functions

- There is a way to tell the compiler that a class is abstract: Define at least one ***pure virtual function*** in the class.
- A pure virtual function is a virtual function with no coding body.
- The body of the virtual function in the base class is removed, and the notation (**=0**) is added to the function declaration.

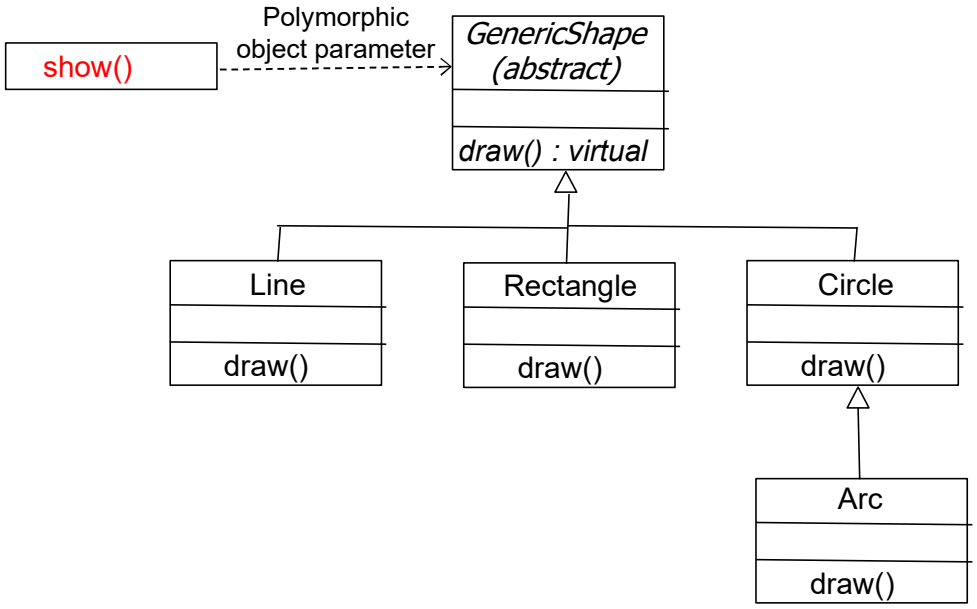
Example : Abstract Base Class (GenericShape)

```
class   GenericShape     // Abstract base class
{
  protected:
     int  x, y;   //  Starting coordinates of shape (pixels)

  public:
    GenericShape (int xin, int yin)    { x = xin; y = yin; }  // Constructor

    virtual  void  draw ( )  =  0;
    // Pure virtual function makes the class abstract
};
```

```
int main()
{
   GenericShape  obj;           // Error: (Object of abstract class not allowed)
   GenericShape  * pobj;        // Pointer to abstract class is allowed
   pobj = new GenericShape;  // Error: (Object of abstract class not allowed)
}
```

Class Diagrams for Geometric Shapes

- The class diagrams below contain the abstract base class and non-abstract derived classes for geometric shapes.
- The ***show*** function is a non-member function.



Line class

```
class Line : public GenericShape {  // Line class is derived
 protected:
   int x2, y2;      // Ending coordinates of line

 public:
   Line (int xin, int yin, int xin2, int yin2) :  // Constructor
         GenericShape (xin, yin),  // Base constructor
         x2 (xin2) , y2 (yin2)
         { }
   void draw() ;   // Virtual draw function
};
```

```
void Line :: draw()
{
   cout << "Type: Line" << endl;
   cout << "Coordinates of Start & End points: "
         << "X1=" << x <<  " ,Y1=" << y
         << " ,X2=" << x2 <<  " ,Y2=" << y2 << endl;
}
```

Rectangle class

```
class  Rectangle  :  public  GenericShape       // Rectangle class is derived
{
 protected:
    int  x2, y2;  // Coordinates of second corner point

 public:
    Rectangle (int xin,  int yin,  int xin2,  int yin2) :  // Constructor
          GenericShape (xin,  yin),  // Base constructor
          x2 (xin2),  y2 (yin2)
          { }
    void draw() ;   // Virtual draw
};
```

```
void  Rectangle  ::  draw()
{
    cout << "Type: Rectangle" << endl;
    cout << "Coordinates of First & Second corner points: "
          << "X1=" << x <<  " ,Y1=" << y
          << " ,X2=" << x2 <<  " ,Y2=" << y2 << endl;
}
```


Circle class

```
class  Circle  :  public  GenericShape    // Circle class is derived
{
 protected:
    int  radius;

 public:
    Circle (int xcen,  int ycen,  int r) :  // Constructor
          GenericShape (xcen,  ycen),  // Base constructor
          radius (r)
          { }
    void draw() ;   // Virtual draw
};
```

```
void  Circle  ::  draw()
{
    cout << "Type: Circle" << endl;
    cout << "Coordinates of Center point: "
          << "X=" << x <<  " ,Y=" << y << endl;
    cout << "Radius= " << radius << endl;
}
```

Non-member show function and Main program

```
// Function calls draw function of different shapes
void show (const Generic_shape & obj )
{
   // Can take references to different shapes.
   // Which draw function will be called?
   obj . draw();   // It is unknown at compile-time.
}
```

```
int main() {    // Main program for testing
   Line       L1  (1, 1, 100, 250);
   Circle     C1  (100, 100, 20);
   Rectangle  R1  (30, 50, 250, 140);
   Circle     C2  (300, 170, 50);

   // The show function can take
   // different shapes as argument.
   show (L1);
   show (C1);
   show (R1);
   show (C2);
}
```


Arc class

- If a new class is defined for a new shape by deriving it from an existing class, it is not necessary to modify the **show** function.
- Example: Programmer can add an **Arc** class, which will not effect the show function.

```
class Arc : public Circle {   // Arc class is derived
 protected:
   int sa, ea;      // Starting Angle & Ending Angle
 public:
   Arc (int xcen, int ycen, int r, int s, int e) : // Constructor
        Circle (xcen, ycen, r),   // Base constructor
        sa (s), ea (e) {}
   void draw() ;   // Virtual draw
};
```

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

Example : Polymorphic Array of pointers

- A polymorphic array of pointers can be used to store the memory addresses of different types of derived objects.
- The base class must be used in declaration of the polymorphic array.

```
int main()  {
   // Define a polymorphic array of pointers
   GenericShape  *    array [5];
   array [0]   =   new  Line        (1, 1, 100, 250);
   array [1]   =   new  Circle      (100, 100, 20);
   array [2]   =   new  Rectangle   (30, 50, 250, 140);
   array [3]   =   new  Circle      (300, 170, 50);
   array [4]   =   new  Arc         (300, 170, 50, 45, 90);

  // Call the non-member show function
  // for each object in array
   for (int i=0;  i<=4;  i++)
        show ( array [i] );
}
```

- Alternative method : Calling the draw member functions directly from main program.

```
// Call the member draw function directly
//  of each object in array
   for (int i=0;  i<=4;  i++)
        array [i]  ->  draw ();
```


Example : Polymorphic Array of pointers to Objects