Polymorphism &Virtual Functions

Course Code: CSC1102 &1103 Course Title: Introduction to Programming

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**11**	**Week No:**	**9 (1X1.5) 10(1X1.5)**	**Semester:**	
Lecturer:	*Name & email*				

Polymorphism

Definition of Polymorphism

- It means ‘Different Forms of the Same Thing’.
- In other words ‘One Name, Different Forms’.
- Real Life Example:

Imagine, you and your friend are walking inside your MidTerm exam room. Just before entering the room, your friend is saying, “You are my best friend, brother. Don’t worry about the exam. I got your back. I’ll slide my script a bit right from me, all you need to do is to take a peek and write.”

Polymorphism

Example of Polymorphism

Now, during the exam, no matter how much you poke your friend, your friend is neither responding nor sliding the script. You are really really upset with your friend.

And after the exam, your friend is like, “I’m sorry, brother. Please forgive me. I’m your best friend. Let me give you a treat. Lets have some fun in Canteen.”

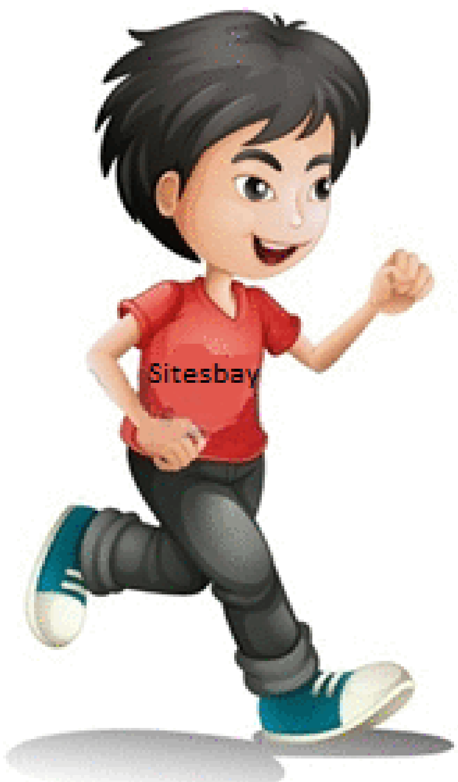
Example of Polymorphism

What do you learn from the story?

- Before the Exam: Your Friend is a Friend.
- During The Exam: Your Friend acts Like Enemy.
- After The Exam: Your Friend is a Friend Again.

The story highlights on different forms of your friend. Sometimes he is like a friend, sometimes he is like an enemy.

Example of Polymorphism



In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

At Home behave like Son

Polymorphism

- The word **polymorphism** means having many forms.
- Typically, polymorphism occurs when there is a hierarchy of classes and they are elated by inheritance.
- 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.

Polymorphism

- There are two types of polymorphism and these are:
  - Compile time polymorphism
    - Uses static or early binding
    - Example: Function and operator overloading
  - Run time polymorphism
    - Uses dynamic or Late binding
    - Example: Virtual functions

Polymorphism (Compile -time)

```cpp
#include <iostream>
using namespace std;
class printData
{
  public:
   void print(int i){
      cout << "Printing int:"<< i << endl;
   }
   void print(double  f) {
      cout <<"Printing float: " << f <<endl;
   }
   void print(char* c) {
     cout <<"Printing character:"<<c <<endl;
   }
   void print(int a, int b){
      cout << "Printing int:"<< a << b <<endl;
   }
```

```cpp
int main(void)
{
   printData pd;

   pd.print(5);
   pd.print(500.263);
         pd.print("Hello C++");
         pd.print(5, 10);

   return 0;
}
```

```
OUTPUT
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
Printing int: 5 10
```

Polymorphism (Run-time)

```cpp
#include <iostream>
using namespace std;
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0){
         width = a;
         height = b;
      }
      virtual void area(){
         cout << "Parent class area :"
              << 0 <<endl;
         return 0;
      }
};
```

```cpp
class Rectangle: public Shape{

    public:
    Rectangle( int a=0, int b=0)
      { width = a;
        height = b;
      }
      void area ()
      {
         cout << "Rectangle class area :"
              << width * height << endl;
      }
};
```

Pointers to Derived Classes

C++ allows base class pointers to point to derived class objects.

- If we have
  - class B_Class{ … };
  - class D_Class: public B_Class{ … };

- Then we can write –
  - B_Class *p1;   // pointer to object of type B_Class
  - D_Class d_obj; // object of type D_Class

- Both statement are valid:

  p1 = &d_obj;

  B_Class *p2 = new D_Class;

Pointers to Derived Classes (contd.)

- Using a base class pointer (pointing to a derived class object) we can access only those members of the derived object **that were inherited from the base**.
- This is because the **base pointer** has knowledge only of the base class.
- It knows nothing about the members added by the derived class.

```
#include <iostream>

using namespace std;

class base

{

public:

   void show()

{  cout << “show base"<<endl;  }

};

class derived : public base

 {

public:

   void show()

{ cout << “show derived"<<endl;   }};
```

```
void main() {

   base b1;

   b1.show();

   derived d1;

   d1.show();

   base *ptrb;

   ptrb = &b1;

   ptrb->show();

   ptrb = &d1;

  ptrb->show();

}All the function calls here are
    statically bound
```

Pointers to Derived Classes (contd.)

- While it is permissible for a base class pointer to point to a derived object, the reverse is <u>not true</u>.
  - base b1;
  - derived *pd = &b1; // compiler error
- We can perform a **downcast** with the help of type-casting, but should use it with caution (see next slide).

Pointers to Derived Classes (contd.)

If we have–

class base { };

class derived : public base { };

class xyz { }; // having no relation with “base” or “derived”

Pointers to Derived Classes (contd.)

- **Then if we write –**

```
base objb, *ptrb;

derived objd;

ptrb = &objd; // ok

derived *ptrd;

ptrd = ptrb ;// compiler error

ptrd = (derived *)ptrb; // ok, valid down casting

xyz obj;// ok

ptrd = (derived *)&obj; // invalid casting, no compiler error, but may cause
     run-time error

ptrd = (derived *)&objb; // invalid casting, no compiler error, but may cause
     run-time error }
```

Pointers to Derived Classes (contd.)

- In fact using type-casting, we can use pointer of any class to point to an object of any other class.
  - The compiler will not complain.
  - During run-time, the address assignment will also succeed.
  - But if we use the pointer to access any member, then it may cause run-time error.

Pointers to Derived Classes (contd.)

- Pointer arithmetic is relative to the data type the pointer is declared as pointing to.
- If we point a base pointer to a derived object and then increment the pointer, it will not be pointing to the next derived object.
- It will be pointing to (what it thinks is) the next base object !!!
- **Be careful about this.**

Important Point on Inheritance

- In C++, only public inheritance supports the perfect IS-A relationship.

- In case of private and protected inheritance, we cannot treat a derived class object in the same way as a base class object
  - Public members of the base class becomes private or protected in the derived class and hence cannot be accessed directly by others using derived class objects

- **If we use private or protected inheritance, we cannot assign the address of a derived class object to a base class pointer directly.**
  - We can use type-casting, but it makes the program logic and structure complicated.

Virtual Functions

- A virtual function is a member function that is declared within a base class and redefined (called ***overriding***) by a derived class.
- It implements the "one interface, multiple methods" philosophy that underlies polymorphism.
- The keyword **virtual** is used to designate a member function as virtual.
- Supports run-time polymorphism with the help of base class pointers.

Virtual Functions (contd.)

- While redefining a virtual function in a derived class, the function signature must match the original function present in the base class .So, we call it ***overriding***, not overloading.
- When a virtual function is redefined by a derived class, the keyword **virtual** is not needed (but can be specified if the programmer wants).
- The “virtual”-ity of the member function continues along the inheritance chain.
- A class that contains a virtual function is referred to as a ***polymorphic class***.

```
#include <iostream>

using namespace std;

class base {

public:

 virtual void show()

{  cout << "show base"<<endl;

}

};

class derived : public base {

public:

   void show() { cout << "show
    derived"<<endl;    }

};
```

```
void main() {

   base b1;

   b1.show();

   derived d1;

   d1.show();

   base *ptrb;

   ptrb = &b1;

   ptrb->show();

   ptrb = &d1;

  ptrb->show();

}
```

```cpp
#include <iostream>
using namespace std;
class base {
public:
virtual void show()
{  cout << "show base"<<endl;   }
};
class derived1 : public base {
public:
   void show()
{ cout << "show derived
  1"<<endl;    }
};
class derived2 : public base {
public:
   void show()
{ cout << "show derived
  2"<<endl;    }
};
```

```cpp
void main() {
   base *ptrb;
   derived1 objd1;
   derived2 objd2;
   int n;
   cout<<" enter a number"<<endl;
   cin >> n;
   if (n ==1)
   ptrb = &objd1;
   else
   ptrb = &objd2;
   ptrb->show(); // guess what ?}
```

Run-time polymorphism

Virtual Destructors

- **Constructors cannot be virtual, but destructors can be virtual.**
- It ensures that the derived class destructor is called when a base class pointer is used while deleting a dynamically created derived class object.

Virtual Destructors (contd.)

Using non-virtual destructor

```
#include <iostream>
using namespace std;
class base {
public:
   ~base() { cout <<  "destructing base"<<endl;  }};
class derived : public base {
public:
   ~derived() {cout << "destructing derived"<<endl;  }}
void main() {
      base *p = new derived;
   delete p;
}
```

Virtual Destructors (contd.)

Using virtual destructor

```
#include <iostream>

using namespace std;

class base {

public:

  virtual ~base()

{

cout <<  "destructing
   base"<<endl;

}};
```

```
class derived : public base
    {

public:

   ~derived() {cout <<
    "destructing
    derived"<<endl;  }};

void main() {

      base *p = new
    derived;

   delete p;

}
```

Virtual functions are inherited

- Once function is declared as virtual, it stays virtual no matter how many layers of derived classes it may pass through.
  - //derived from derived, not base

```cpp
#include <iostream>

using namespace std;

class base {

public:

virtual void show()

{   cout << "show base"<<endl;
     }};

class derived1 : public base

{

public:

   void show()

{ cout << "show derived
     1"<<endl;    }};

class derived2 : public derived1
     {};
```

```cpp
void main() {

   base b1;

  derived1 d1;

    derived2 d2;

   base *pb = &b1;

   pb->show();

   pb = &d1;

   pb->show();

   pb = &d2;

   pb->show();

}
```

What if there is show in derived 2 ?

More About Virtual Functions

- Helps to guarantee that a derived class will provide its own redefinition.
- If we want to omit the body of a virtual function in a base class, we can use pure virtual functions.

  virtual ret-type func-name(param-list) = 0;
- Pure virtual function is a virtual function that has no definition in its base class.

Pure virtual function

```
Class figure{
   protected:
        double x,y;
   public:
   void set_dim(double I,
   double j){
   x=I;
   Y= j;
     }
   Virtual void
   show_area()=0 ;// pure
   function
   };
```

- It makes a class an ***abstract class***.
  - We cannot create any objects of such classes.
- It forces derived classes to override it own implementation.
  - Otherwise become abstract too and the complier will report an error.

```
Class figure{
   protected:
       double x,y;
   public:
   void set_dim(double I,
   double j=0)
   {
   x=I;
   Y= j;
     }
   Virtual void
   show_area()=0 ;// pure
   function

};

Class triangle: public
figure{
   Public:
   Void show_area()
   {cout<< x* 0.5 * y;}};
```

```
Class circle: public figure

{
    Public:
    // no definiton of show_area() will
    //cause error
    };

int main(){
    Figure *p;
    Triangle t;
    Circle c; // illegal – can’t create
    P= &t;
    P-> set_dim(10.0,5.0);
    P-> show_area();
    p= &c;
    P-> set_dim(10.0);
    P-> show_area();
    return0;}
```

Abstract class

- If a class has at least one pure virtual function, then that class is said to be abstract.
- An abstract class has one important feature: there are can be no object of the class.
- Instead, abstract class must be used only as a base that other classes will inherit.
- Even if the class is abstract, you still can use it to declare pointers, which are needed to support runt time polymorphism.

Applying Polymorphism

Early binding

- Normal functions, overloaded functions
- Nonvirtual member and friend functions
- Resolved at compile time
- Very efficient
- But lacks flexibility

Late binding

- Virtual functions accessed via a base class pointer
- Resolved at run-time
- Quite flexible during run-time
- But has run-time overhead; slows down program execution

Final Comments

- Run-time polymorphism is not automatically activated in C++.
- We have to use virtual functions and base class pointers to enforce and activate run-time polymorphism in C++.

Thank You