**C++ Polymorphism**

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation and polymorphism.

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

**There are two types of polymorphism in C++:**

* **Compile time polymorphism:** It is achieved by function overloading and operator overloading which is also known as static binding or early binding.
* **Runtime polymorphism:** It is achieved by method overriding which is also known as dynamic binding or late binding.

## **C++ Runtime Polymorphism Example : hides the void eat() on Animal class.**

|  |
| --- |
| #include<iostream>  using namespace std;  class Animal{  public:  void eat(){  cout << "eating" << endl;  }  };  class dog : public Animal{  public:  void eat(){  cout << "Fucking dog is eating" << endl;  }  };  int main(){  dog d = dog();  d.eat();  return 0;  }  //Fucking dog is eating |

## **C++ Runtime Polymorphism Example: By using two derived class…**

|  |
| --- |
| #include<iostream>  using namespace std;  class shape{  public:  int height, width;    public:  shape(int a, int b){  height = a;  width = b;  }  int area (){  cout << "parent class area" << endl;  return 0;  }  };  class Rectangle : public shape{  public:  Rectangle( int a = 10, int b = 15) : shape(a, b) {  //constructer  }  int area(){  return height \* width;  }  };  class Triangle : public shape{  public:  Triangle( int a = 10, int b = 20) : shape(a, b) {  //constructer  }  int area(){  return (height \* width)/2;  }  };  int main() {  shape \*shape;  Rectangle rec(10,7);  Triangle tri(10,5);  // store the address of Rectangle  shape = &rec;    // call rectangle area.  shape->area();  // store the address of Triangle  shape = &tri;    // call triangle area.  shape->area();    return 0;  }  **/\*parent class area**  **parent class area\*/** |

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this –

|  |
| --- |
| #include<iostream>  using namespace std;  class shape{  public:  int height, width;    public:  shape(int a, int b){  height = a;  width = b;  }  virtual int area (){  cout << "parent class area" << endl;  return 0;  }  };  class Rectangle : public shape{  public:  Rectangle( int a = 10, int b = 15) : shape(a, b) {  //constructer  }  int area(){  return height \* width;  }  };  class Triangle : public shape{  public:  Triangle( int a = 10, int b = 20) : shape(a, b) {  //constructer  }  int area(){  return (height \* width)/2;  }  };  int main() {  shape \*shape;  Rectangle rec(10,7);  Triangle tri(10,5);  // store the address of Rectangle  shape = &rec;    // call rectangle area.  cout << "area of a rectangle is : " << shape->area() << endl;  // store the address of Triangle  shape = &tri;    // call triangle area.  cout << "area of triangle is : " << shape->area() << endl;    return 0;  }  /\*area of a rectangle is : 70  area of triangle is : 25\*/ |

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in \*shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

## **Virtual Function**

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

## **Pure Virtual Functions**

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following −

class Shape {

protected:

int width, height;

public:

Shape(int a = 0, int b = 0) {

width = a;

height = b;

}

// pure virtual function

virtual int area() = 0;

};

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.