**Polymorphism**

Polymorphism is an important concept of object-oriented programming. It simply means more than one form. That is, the same entity (function or operator) behaves differently in different scenarios. For example,

The + operator in C++ is used to perform two specific functions. When it is used with numbers (integers and floating-point numbers), it performs addition.

int a = 5;

int b = 6;

int sum = a + b; // sum = 11

And when we use the + operator with strings, it performs string concatenation. For example,

string firstName = "abc ";

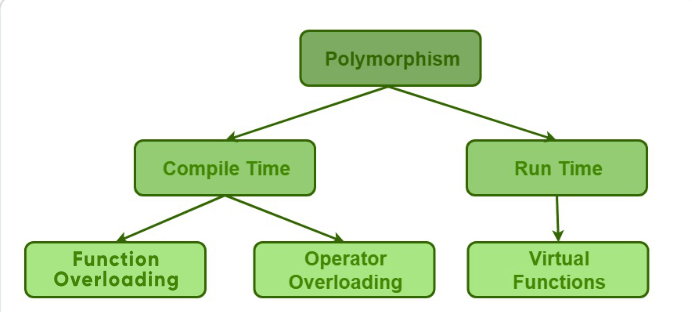
string lastName = "xyz";

// name = "abc xyz"

string name = firstName + lastName;

**In C++ polymorphism is mainly divided into two types:**

* Compile time Polymorphism
* Runtime Polymorphism



1. **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.

* + [Function Overloading](https://www.geeksforgeeks.org/function-overloading-c/): When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

|  |
| --- |
| // C++ program for function overloading  #include <iostream.h>    using namespace std;  class BCA  {      public:        // function with 1 int parameter      void func(int x)      {          cout << "value of x is " << x << endl;      }        // function with same name but 1 double parameter      void func(double x)      {          cout << "value of x is " << x << endl;      }        // function with same name and 2 int parameters      void func(int x, int y)      {          cout << "value of x and y is " << x << ", " << y << endl;      }  };    int main()  {        BCA obj1;        // Which function is called will depend on the parameters passed      // The first 'func' is called      obj1.func(7);        // The second 'func' is called      obj1.func(9.132);        // The third 'func' is called      obj1.func(85,64);      return 0;  } |

* + **Output:**
  + value of x is 7
  + value of x is 9.132
  + value of x and y is 85, 64
  + In the above example, a single function named *func* acts differently in three different situations which is the property of polymorphism.
  + [Operator Overloading](https://www.geeksforgeeks.org/operator-overloading-c/): C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands, adds them and when placed between string operands, concatenates them.  
    **/ C++ program to overload ++ when used as prefix**
  + **#include <iostream>**
  + **using namespace std;**
  + **class Count {**
  + **private:**
  + **int value;**
  + **public:**
  + **// Constructor to initialize count to 5**
  + **Count() : value(5) {}**



* + **// Overload ++ when used as prefix**
  + **void operator ++()**
  + **{**
  + **value = value + 1;**
  + **}**
  + **void display() {**
  + **cout << "Count: " << value << endl;**
  + **}**
  + **};**
  + **int main()**
  + **{**
  + **Count count1;**
  + **// Call the "void operator ++()" function**
  + **++count1;**
  + **count1.display();**
  + **return 0;**
  + **}**
  + **Output**
  + **Count: 6**
  + **Here, we have overloaded the ++ operator, which operates on objects of Count class (object count1 in this case).**
  + **We have used this overloaded operator to directly increment the value variable of count1 object by 1.**
  + **This is also a compile-time polymorphism.**
  + [Runtime polymorphism](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/): This type of polymorphism is achieved by Function Overriding.

**C++ Function Overriding**

In C++ inheritance, we can have the same function in the base class as well as its derived classes. When we call the function using an object of the derived class, the function of the derived class is executed instead of the one in the base class. So, different functions are executed depending on the object calling the function. This is known as function overriding in C++.

For example,

// C++ program to demonstrate function overriding

#include <iostream>

using namespace std;

class Base {

public:

virtual void print()

{

cout << "Base Function" << endl;

}

};

class Derived : public Base {

public:

void print() {

cout << "Derived Function" << endl;

}

};

int main()

{

Derived derived1;

// Call print() function of Derived class

derived1.print();

return 0;

}

Output

Derived Function

Here, we have used a print() function in the Base class and the same function in the Derived class

When we call print() using the Derived object derived1, it overrides the print() function of Base by executing the print() function of the Derived class.

It's a runtime polymorphism because the function call is not resolved by the compiler, but it is resolved in the runtime instead.

**C++ Virtual Functions**

In C++, we may not be able to override functions if we use a pointer of the base class to point to an object of the derived class.

Using virtual functions in the base class ensures that the function can be overridden in these cases.

Thus, virtual functions actually fall under function overriding. For example,

// C++ program to demonstrate the use of virtual functions

#include <iostream>

using namespace std;

class Base {

public:

virtual void print()

{

cout << "Base Function" << endl;

}

};

class Derived : public Base

{

public:

void print() {

cout << "Derived Function" << endl;

}

};

int main() {

Derived derived1;

// pointer of Base type that points to derived1

Base\* base1 = &derived1;

// calls member function of Derived class

base1->print();

return 0;

}

Output

Derived Function

Here, we have used a virtual function print() in the Base class to ensure that it is overridden by the function in the Derived class.

Virtual functions are runtime polymorphism.

Why Polymorphism?

Polymorphism allows us to create consistent code. For example,

Suppose we need to calculate the area of a circle and a square. To do so, we can create a Shape class and derive two classes Circle and Square from it.

In this case, it makes sense to create a function having the same name calculateArea() in both the derived classes rather than creating functions with different names, thus making our code more consistent.