Chapter:7

Polymorphism

Introduction

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

What is Polymorphism in C++?

- Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

What is Polymorphism in C++?

- Polymorphism in C++ means, the same entity behaves differently in different scenarios.
- Consider this example:

The " +" operator in c++ can perform two specific functions at two different scenarios i.e when the "+" operator is used in numbers, it performs addition.

```
int a = 6;
int b = 6;
int sum = a + b;    // sum =12
```

What is Polymorphism in C++?

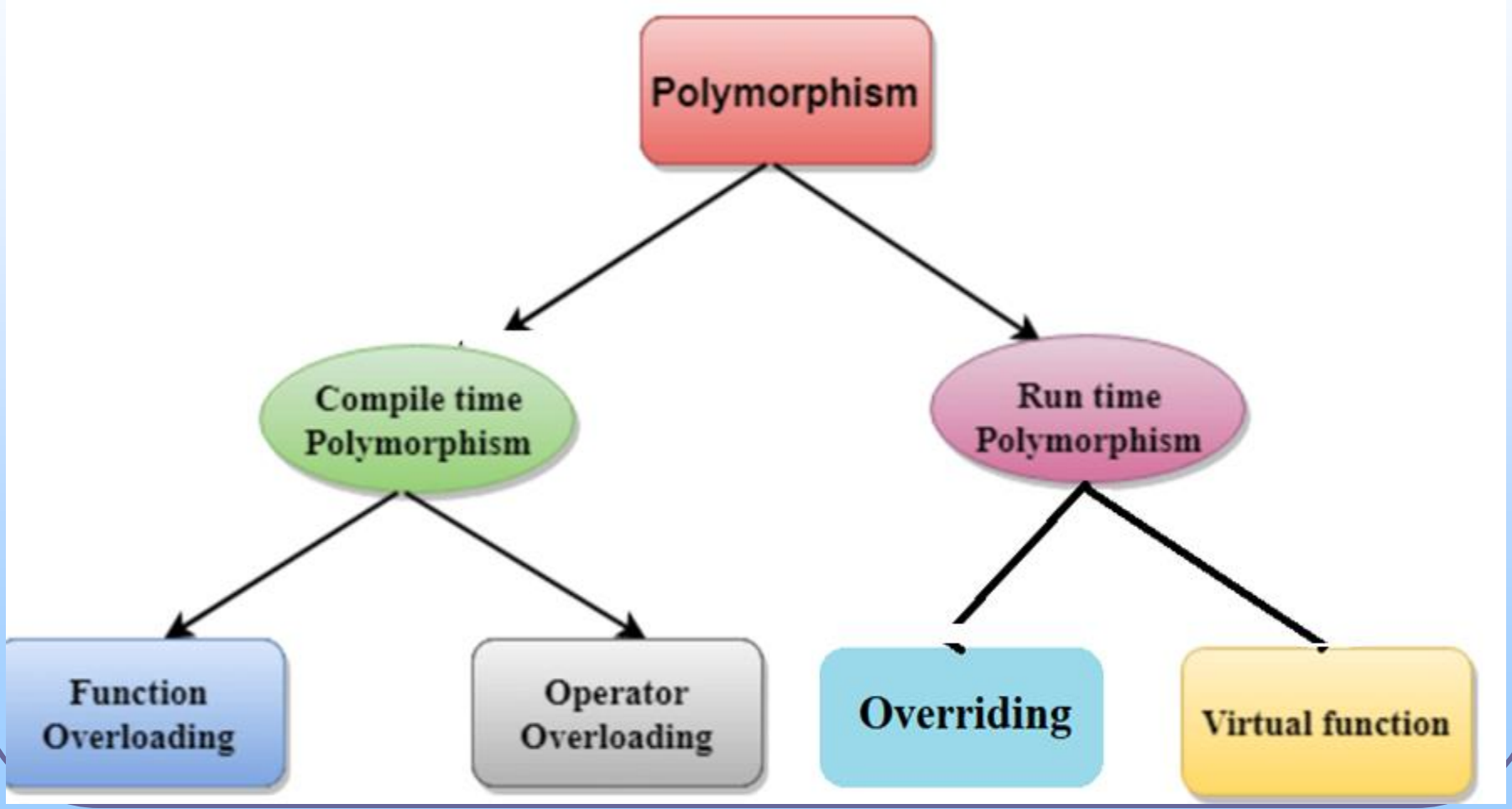
- And the same “+” operator is used in the string, it performs concatenation.

```
string firstName = "Great ";

string lastName = "Learning";

string name = firstName + lastName;
```

Types of Polymorphism in C++:



Compile Time Polymorphism

- In compile-time polymorphism, a function is called at the time of program compilation. We call this type of polymorphism as early binding or Static binding.
- Function overloading and operator overloading is the type of Compile time polymorphism.

Overloading Functions

- **Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.**

Overloading Functions

- Let's see the simple example of function overloading where we are changing number of arguments of add() method.

Example:1

```
#include <iostream.h>
class Cal
{
    public:
    int add(int a,int b)
    {  return a + b; }
    int add(int a, int b, int c)
    {  return a + b + c;  }
};
```

Example:1

```
int main(void)
{
    Cal C;
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
   return 0;
}
```

Example:1

The Output:

30

55

Example:2

- Let's see the simple example when the type of the arguments vary.

```
#include<iostream.h>
class math
{
  public:
  int mul(int a, int b)
  {
    return a*b;
  }
```

Example:2

```
float mul(double x, int y)
 {
    return x*y;
  }
};
```

Example:2

```
int main()
{
    math m;
    cout << "R1 is: " <<m.mul(6,7)<< endl;
    cout <<"R2 is: "  <<m.mul(0.2,3)<< endl;
    return 0;
}
```

Example:2

The Output:

R1 is: 42

R2 is: 0.6

Function With Default Arguments

- A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument. In case any value is passed, the default value is overridden.

Example:

```
#include<iostream>
class math
{
  public:
  void fun(int i)
  {  cout << "Value of i is : " <<i<< endl;   }
  void fun(int a,int b=9)
  {  cout << "a is : " <<a<< endl<<b is : " <<b; }
};
```

Example:

```
int main()
{
    math m;
    m.fun(12);
    return 0;
}
```

Example:

- The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

Function with pass by reference

- ***Pass-by-reference*** **means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called function. The called function can modify the value of the argument by using its reference passed in.**

- **This can be useful when you need to change the value of the arguments:**

Example:

```
#include<iostream>
class math
{
  void fun(int x)
  {  cout << "Value of x is : " <<x<<endl;  }
  void fun(int &b)
  {  cout << "Value of b is : " <<b<< endl;  }
};
```

Example:

```
int main()
{
  Math m;
  int a=10;
  m.fun(a); // error, which f()?
  return 0;
}
```

Example:

- **The above example shows an error "call of overloaded 'fun(int&)' is ambiguous". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).**

Runtime Polymorphism

- **In a Runtime polymorphism, functions are called at the time the program execution. Hence, it is known as late binding or dynamic binding.**

Function overriding

- In function overriding, we give the new definition to base class function in the derived class. At that time, we can say the base function has been overridden. It can be only possible in the ‘derived class’. In function overriding, we have two definitions of the same function, one in the superclass and one in the derived class. The decision about which function definition requires calling happens at runtime

Example:

```
#include <iostream>
using namespace std;
class Base
 {
   public:
     void print()
     {  cout << "Base Function" << endl;    }
};
```

Example:

```
class Derived : public Base
 {
   public:
    void print()
     {  cout << "Derived Function" << endl;  }
};
```

Example:

```
int main()
 {
    Derived D;
    D.print();
   return 0;
}
```

The Output

Derived Function