OBJECT ORIENTED PROGRAMMING THROUGH C++

UNIT-3

Inheritance:

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called “derived class” or “child class” and the existing class is known as the “base class” or “parent class”. The derived class now is said to be inherited from the base class.

The most important advantage of inheritance in C++ is code reusability. Without redefining the old class, you can add new properties to the derived class, and you even can redefine an inherited class member function.

Sub Class: The class that inherits properties from another class is called Subclass or Derived Class.

Super Class: The class whose properties are inherited by a subclass is called Base Class or Superclass.

Modes of Inheritance:

There are 3 modes of inheritance.

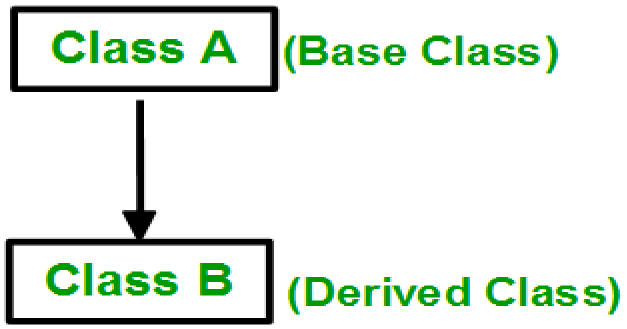
1. **Public Mode:** If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.
2. **Protected Mode:** If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
3. **Private Mode:** If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

Types Of Inheritance:-

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

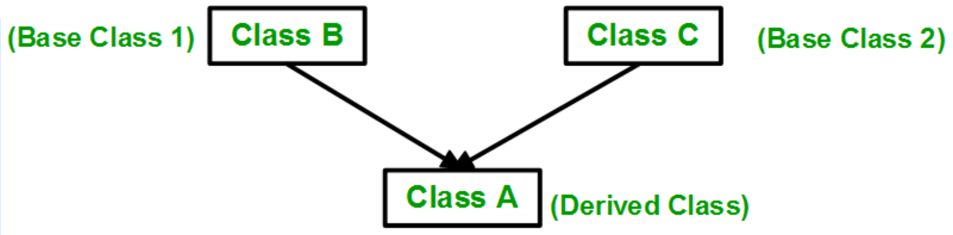
1. Single Inheritance:

In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.



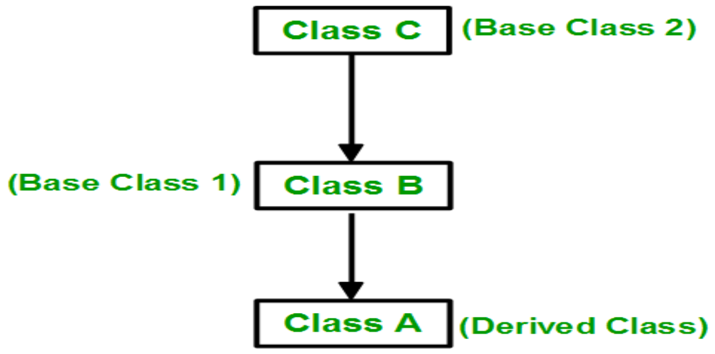
2. Multiple Inheritance:

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e., one **subclass** is inherited from more than one **base class**.



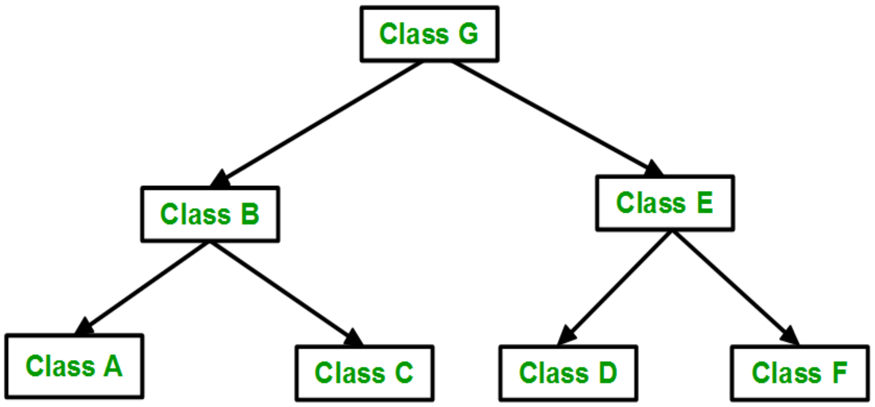
3. Multilevel Inheritance:

In this type of inheritance, a derived class is created from another derived class.



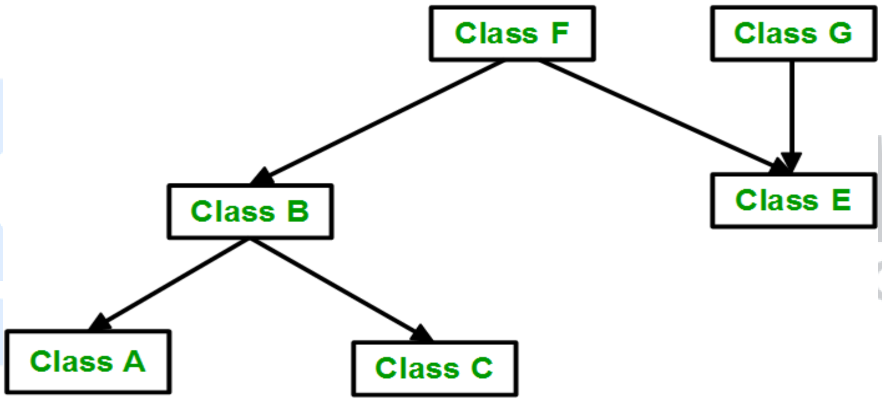
4. Hierarchical Inheritance:

In this type of inheritance, more than one subclass is inherited from a single base class. i.e., more than one derived class is created from a single base class.



5. Hybrid (Virtual) Inheritance:

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritances:



Function Overloading

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.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

Example:

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

// program of function overloading when number of arguments vary.

```
#include <iostream>
using namespace std;
class Cal {
  public:
static int add(int a,int b){
    return a + b;
  }
static int add(int a, int b, int c)
  {
    return a + b + c;
  }
};
int main(void) {
  Cal C;                          //    class object declaration.
  cout<<C.add(10, 20)<<endl;
  cout<<C.add(12, 20, 23);
 return 0;
}
```

Output:

```
30
55
```

Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax:

return_type class_name : : operator op(argument_list)

{

// body of the function.

}

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains at least one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

Example:

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

#include <iostream>

using namespace std;

```
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}
     void operator ++()       {
       num = num+2;
     }
     void Print() {
        cout<<"The Count is: "<<num;
     }
};
int main()
{
   Test tt;
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();
   return 0;
}
```

Output:

```
The Count is: 10
```

virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.

- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore, compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

Example:

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
 a->display();   //Late Binding occurs
}
```

Output:

```
Derived Class is invoked
```

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

```
virtual void display() = 0;
```

Example:

```
#include <iostream>
using namespace std;
class Base
{
   public:
   virtual void show() = 0;
};
```

```
class Derived : public Base
{
  public:
  void show()
  {
    std::cout << "Derived class is derived from the base class." << std::endl;
  }
};
int main()
{
  Base *bptr;
  //Base b;
  Derived d;
  bptr = &d;
  bptr->show();
  return 0;
}
```

Output:

```
Derived class is derived from the base class.
```