3.1 Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows −

Live Demo

```
#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
```

```
};

int main(void) {
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total area: 35
```

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way −

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

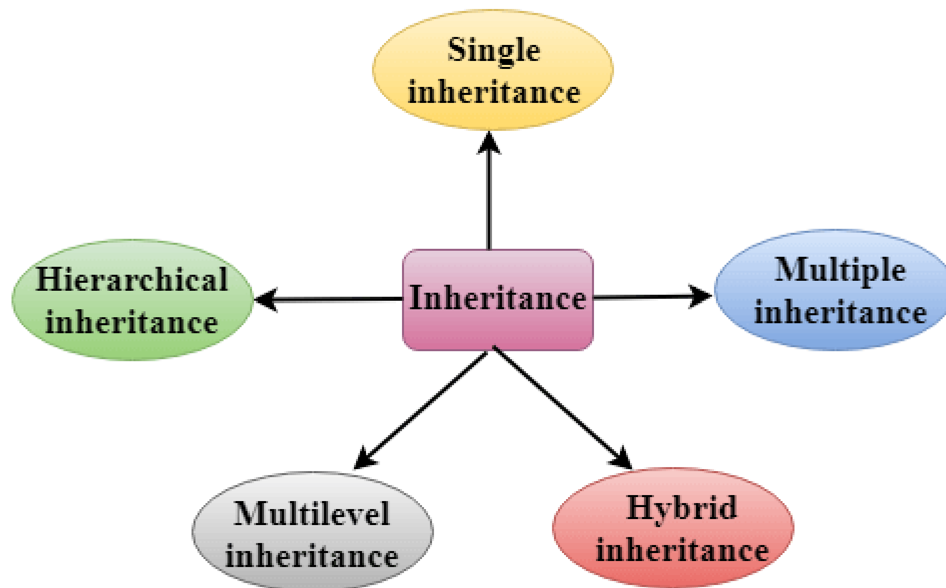
A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Types Of Inheritance

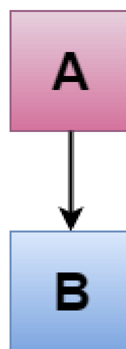
C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



C++ Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
     Programmer p1;
     cout<<"Salary: "<<p1.salary<<endl;
     cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
```

```
}
```
Output:

```
Salary: 60000
Bonus: 5000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking...";
     }
   };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```
Output:

```
Eating...
Barking...
```

Let's see a simple example.

```
#include <iostream>
using namespace std;
class A
{
    int a = 4;
    int b = 5;
    public:
    int mul()
    {
        int c = a*b;
        return c;
    }
};

class B : private A
{
```

```
    public:
    void display()
    {
        int result = mul();
        std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
    }
};
int main()
{
   B b;
   b.display();

    return 0;
}
```

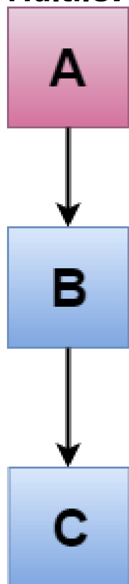
Output:

Multiplication of a and b is : 20

In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.



C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
1. #include <iostream>
2. using namespace std;
3.  class Animal {
4.    public:
```

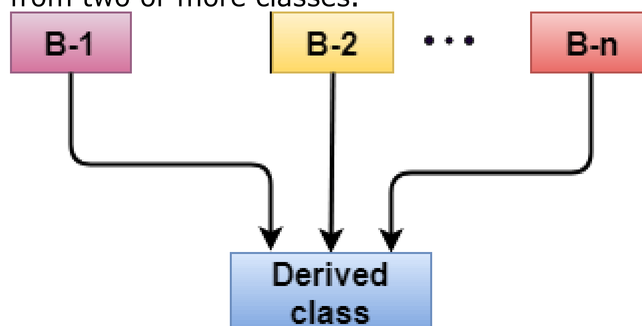
```
5.  void eat() {
6.     cout<<"Eating..."<<endl;
7.  }
8.    };
9.    class Dog: public Animal
10.   {
11.       public:
12.     void bark(){
13.    cout<<"Barking..."<<endl;
14.     }
15.   };
16.   class BabyDog: public Dog
17.   {
18.       public:
19.     void weep() {
20.    cout<<"Weeping...";
21.     }
22.   };
23. int main(void) {
24.    BabyDog d1;
25.    d1.eat();
26.    d1.bark();
27.     d1.weep();
28.     return 0;
29. }
```

Output:

```
Eating...
Barking...
Weeping...
```

C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class:

```
class D : visibility B-1, visibility B-2, ?
{
   // Body of the class;
}
```

Let's see a simple example of multiple inheritance.

```
#include <iostream>
using namespace std;
class A
{
```

```
   protected:
    int a;
    public:
    void get_a(int n)
    {
       a = n;
    }
};

class B
{
    protected:
    int b;
    public:
    void get_b(int n)
    {
       b = n;
    }
};
class C : public A,public B
{
   public:
    void display()
    {
       std::cout << "The value of a is : " <<a<< std::endl;
       std::cout << "The value of b is : " <<b<< std::endl;
       cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

Virtual Base Class

The virtual base class is used when a derived class has multiple copies of the base class.

Example Code

```
#include <iostream>
using namespace std;
class B {
   public: int b;
};

class D1 : public B {
   public: int d1;
};

class D2 : public B {
   public: int d2;
};

class D3 : public D1, public D2 {
   public: int d3;
};

int main() {
   D3 obj;

   obj.b = 40; //Statement 1, error will occur
   obj.b = 30; //statement 2, error will occur
   obj.d1 = 60;
   obj.d2 = 70;
   obj.d3 = 80;

   cout<< "\n B : "<< obj.b
   cout<< "\n D1 : "<< obj.d1;
   cout<< "\n D2: "<< obj.d2;
   cout<< "\n D3: "<< obj.d3;
}
```

In the above example, both D1 & D2 inherit B, they both have a single copy of B. However, D3 inherit both D1 & D2, therefore D3 have two copies of B, one from D1 and another from D2.

Statement 1 and 2 in above example will generate error, as compiler can't differentiate between two copies of b in D3.

To remove multiple copies of B from D3, we must inherit B in D1 and D3 as virtual class.

So, above example using virtual base class will be −

Example Code

```
#include<iostream>
using namespace std;
class B {
   public: int b;
};

class D1 : virtual public B {
   public: int d1;
};

class D2 : virtual public B {
   public: int d2;
};

class D3 : public D1, public D2 {
   public: int d3;
};

int main() {
   D3 obj;

   obj.b = 40; // statement 3
   obj.b = 30; // statement 4

   obj.d1 = 60;
   obj.d2 = 70;
   obj.d3 = 80;

   cout<< "\n B : "<< obj.b;
   cout<< "\n D1 : "<< obj.d1;
   cout<< "\n D2 : "<< obj.d2;
   cout<< "\n D3 : "<< obj.d3;
}
```

Output

```
B : 30
D1 : 60
D2 : 70
D3 : 80
```

Now, D3 have only one copy of B and statement 4 will overwrite the value of b, given in statement 3.

Abstract Class and Pure Virtual Function in C++

Abstract Class is a class which contains at least one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Pure Virtual Functions in C++

Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0. Here is the syntax for a pure virtual function,

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

Example of Abstract Class in C++

, Below we have a simple example where we have defined an abstract class

```
//Abstract base class
class Base
{
    public:
    virtual void show() = 0;     // Pure Virtual Function
};

class Derived:public Base
{
    public:
    void show()
    {
```

```
        cout << "Implementation of Virtual Function in Derived class\n";
    }
};

int main()
{
    Base obj;    //Compile Time Error
    Base *b;
    Derived d;
    b = &d;
    b->show();
}
```

```
Implementation of Virtual Function in Derived class
```

In the above example Base class is abstract, with pure virtual show() function, hence we cannot create object of base class.

Why can't we create Object of an Abstract Class?

When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE(studied in last topic), but doesn't put any address in that slot. Hence the VTABLE will be incomplete.
As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an errror message whenever you try to do so.

Pure Virtual definitions

- Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class.
- Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.

```
// Abstract base class
class Base
{
    public:
    virtual void show() = 0;     //Pure Virtual Function
};

void Base :: show()     //Pure Virtual definition
{
    cout << "Pure Virtual definition\n";
}
```

```
class Derived:public Base
{
    public:
    void show()
    {
        cout << "Implementation of Virtual Function in Derived class\n";
    }
};

int main()
{
    Base *b;
    Derived d;
    b = &d;
    b->show();
}
```

```
Implementation of Virtual Function in Derived class
```

Constructor/ Destructor Call in C++

Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class. **If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke**, i.e the order of invocation is that the base class's default constructor will be invoked first and then the derived class's default constructor will be invoked.

Why the base class's constructor is called on creating an object of derived class?
To understand this you will have to recall your knowledge on inheritance. What happens when a class is inherited from other? The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only. So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class's constructor as the definition of these members exists in base class only. This is why the constructor of **base class is called first to initialize all the inherited members.**

// C++ program to show the order of constructor call
// in single inheritance

```
#include <iostream>
using namespace std;
// base class
class Parent
{
	public:
	// base class constructor
	Parent()
	{
		cout << "Inside base class" << endl;
	}
};

// sub class
class Child : public Parent
{
	public:

	//sub class constructor
	Child()
	{
		cout << "Inside sub class" << endl;
	}
};

// main function
int main() {

	// creating object of sub class
	Child obj;

	return 0;
}
```