# UNIT-3

# C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

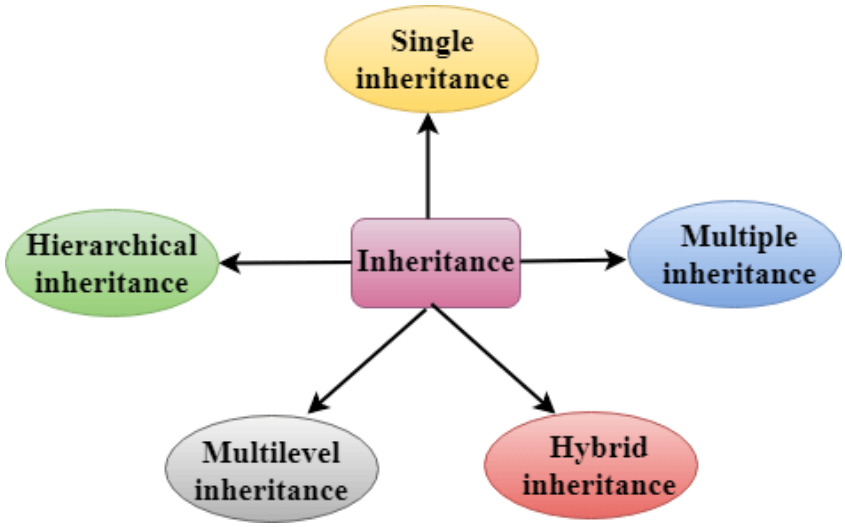
## Advantage of C++ Inheritance

**Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

## Types Of Inheritance

**C++ supports five types of inheritance:**

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




## Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name : visibility-mode base_class_name
{
   // body of the derived class.
}
```

**Where,**

**derived_class_name:** It is the name of the derived class.

**visibility mode:** The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

**base_class_name:** It is the name of the base class.

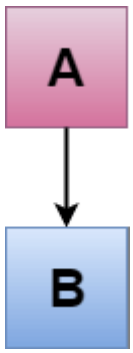
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

# 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
```

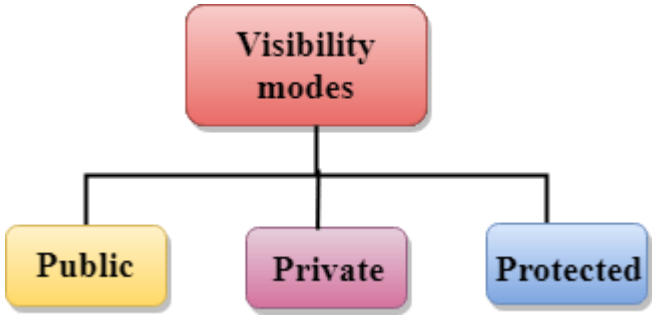
In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

# How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

**Visibility modes can be classified into three categories:**




- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

# 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.

# C++ 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++.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
  };
  class Dog: public Animal
  {
     public:
   void bark(){
  cout<<"Barking..."<<endl;
   }
  };
  class BabyDog: public Dog
  {
     public:
   void weep() {
```

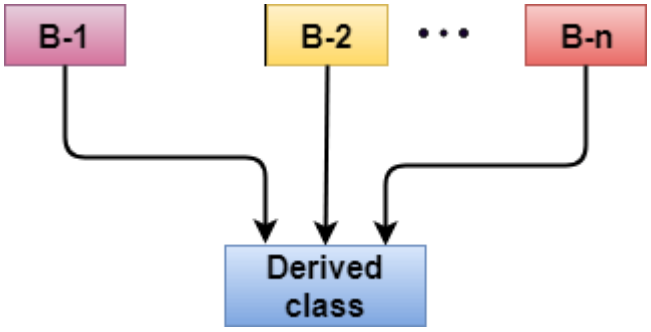
```
    cout<<"Weeping...";
     }
   };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

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.

# Ambiquity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class A
{
   public:
   void display()
   {
      std::cout << "Class A" << std::endl;
   }
};
class B
{
   public:
   void display()
   {
      std::cout << "Class B" << std::endl;
   }
};
class C : public A, public B
{
   void view()
   {
      display();
   }
};
int main()
{
   C c;
   c.display();
   return 0;
}
```

Output:

```
error: reference to 'display' is ambiguous
        display();
```

- The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
   void view()
   {
      A :: display();        // Calling the display() function of class A.
```

```
        B :: display();         // Calling the display() function of class B.

    }
};
```

An ambiguity can also occur in single inheritance.

Consider the following situation:

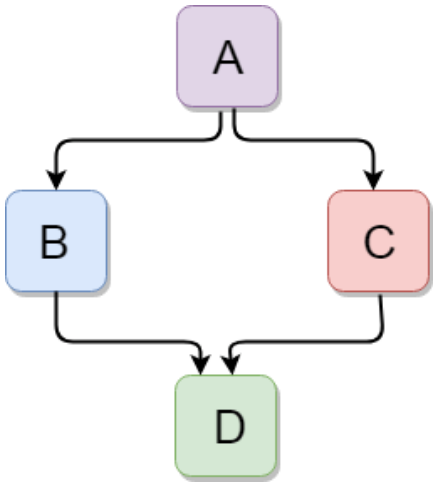
```
class A
{
   public:
void display()
{
   cout<<?Class A?;
}
} ;
class B
{
  public:
 void display()
{
 cout<<?Class B?;
}
} ;
```

In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
int main()
{
    B b;
   b.display();              // Calling the display() function of B class.
   b.B :: display();       // Calling the display() function defined in B class.
}
```

# C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.




Let's see a simple example:

```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
     std::cout << "Enter the value of 'a' : " << std::endl;
     cin>>a;
   }
};

class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
      std::cout << "Enter the value of 'b' : " << std::endl;
     cin>>b;
   }
};
class C
{
   protected:
   int c;
   public:
   void get_c()
```

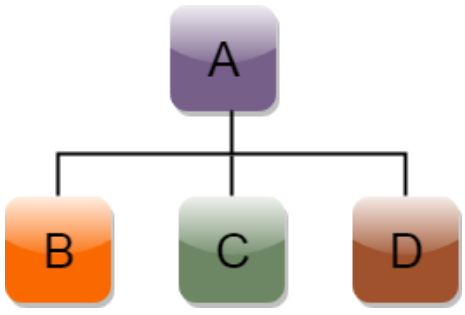
```
    {
        std::cout << "Enter the value of c is : " << std::endl;
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
        get_a();
        get_b();
        get_c();
        std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
    }
};
int main()
{
    D d;
    d.mul();
    return 0;
}
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

# C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



**Syntax of Hierarchical inheritance:**

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Shape                // Declaration of base class.
{
   public:
   int a;
   int b;
   void get_data(int n,int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape  // inheriting Shape class
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
   }
};
class Triangle : public Shape    // inheriting Shape class
{
   public:
```

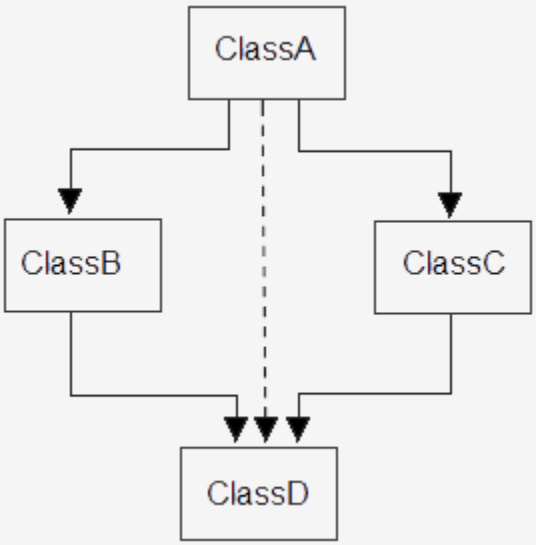
```
    int triangle_area()
    {
       float result = 0.5*a*b;
       return result;
    }
};
int main()
{
    Rectangle r;
    Triangle t;
    int length,breadth,base,height;
    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
    cin>>length>>breadth;
    r.get_data(length,breadth);
    int m = r.rect_area();
    std::cout << "Area of the rectangle is : " <<m<< std::endl;
    std::cout << "Enter the base and height of the triangle: " << std::endl;
    cin>>base>>height;
    t.get_data(base,height);
    float n = t.triangle_area();
    std::cout <<"Area of the triangle is : "  << n<<std::endl;
    return 0;
}
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

## C++ Virtual Base Class

Virtual base class is used in situation where a derived have multiple copies of base class. Consider the following figure:




**Example without using virtual base class**

```
#include<iostream.h>
#include<conio.h>

class ClassA
{
	public:
	int a;
};

class ClassB : public ClassA
{
	public:
	int b;
};
class ClassC : public ClassA
{
	public:
	int c;
};

class ClassD : public ClassB, public ClassC
{
	public:
	int d;
};

void main()
{
```

```
        ClassD obj;

        obj.a = 10;       //Statement 1, Error occur
        obj.a = 100;      //Statement 2, Error occur

        obj.b = 20;
        obj.c = 30;
        obj.d = 40;

        cout<< "\n A : "<< obj.a;
        cout<< "\n B : "<< obj.b;
        cout<< "\n C : "<< obj.c;
        cout<< "\n D : "<< obj.d;

    }
```

Output :

```
A from ClassB  : 10
A from ClassC  : 100
B : 20
C : 30
D : 40
```

In the above example, both **ClassB** & **ClassC** inherit **ClassA**, they both have single copy of **ClassA**. However **ClassD** inherit both **ClassB** & **ClassC**, therefore **ClassD** have two copies of **ClassA**, one from **ClassB** and another from **ClassC**.

Statement 1 and 2 in above example will generate error, bco'z compiler can't differentiate between two copies of **ClassA** in **ClassD**.

To remove multiple copies of **ClassA** from **ClassD**, we must inherit **ClassA** in **ClassB** and **ClassC** as **virtual** class.

**Example using virtual base class**

```
#include<iostream.h>
#include<conio.h>

class ClassA
{
    public:
    int a;
};

class ClassB : virtual public ClassA
{
    public:
    int b;
};
```

```
class ClassC : virtual public ClassA
{
    public:
    int c;
};

class ClassD : public ClassB, public ClassC
{
    public:
    int d;
};

void main()
{
            ClassD obj;

            obj.a = 10;      //Statement 1
            obj.a = 100;     //Statement 2

            obj.b = 20;
            obj.c = 30;
            obj.d = 40;

            cout<< "\n A : "<< obj.a;
            cout<< "\n B : "<< obj.b;
            cout<< "\n C : "<< obj.c;
            cout<< "\n D : "<< obj.d;

      }
```

Output :

```
    A : 100
    B : 20
    C : 30
    D : 40
```

According to the above example, **ClassD** have only one copy of **ClassA** and statement 4 will overwrite the value of **a**, given in statement 3.

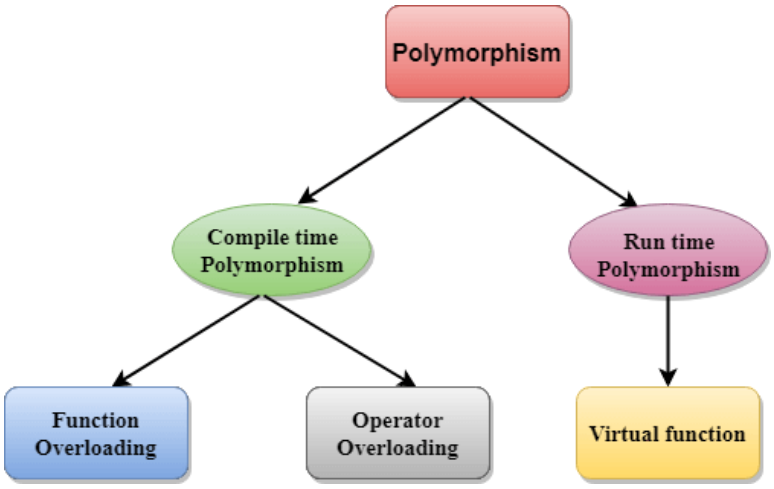
# C++ Polymorphism

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.

## Real Life Example Of Polymorphism

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.

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




- **Compile time polymorphism**: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
 class A                          //  base class declaration.
{
    int a;
    public:
    void display()
    {
        cout<< "Class A ";
    }
};
class B : public A                  //  derived class declaration.
{
   int b;
   public:
  void display()
 {
      cout<<"Class B";
 }
};
```

In the above case, the prototype of display() function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

- **Run time polymorphism**: Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

## Differences b/w compile time and run time polymorphism.

| Compile time polymorphism | Run time polymorphism |
|---|---|
| The function to be invoked is known at the compile time. | The function to be invoked is known at the run time. |
| It is also known as overloading, early binding and static binding. | It is also known as overriding, Dynamic binding and late binding. |
| Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters. | Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters. |
| It is achieved by function overloading and operator overloading. | It is achieved by virtual functions and pointers. |
| It provides fast execution as it is known at the compile time. | It provides slow execution as it is known at the run time. |
| It is less flexible as mainly all the things execute at the compile time. | It is more flexible as all the things execute at the run time. |

## C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

// an example without the virtual keyword.

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
class Dog: public Animal
{
```

```
 public:
 void eat()
   {          cout<<"Eating bread...";
   }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

**Output:**

```
Eating bread...
```

# C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

// an example with virtual keyword.

```
#include <iostream>
using namespace std;
class Shape {                                  //  base class
   public:
virtual void draw(){                          // virtual function
cout<<"drawing..."<<endl;
   }
};
class Rectangle: public Shape                 //  inheriting Shape class.
{
 public:
 void draw()
  {
     cout<<"drawing rectangle..."<<endl;
   }
};
class Circle: public Shape                    //  inheriting Shape class.

{
 public:
 void draw()
  {
    cout<<"drawing circle..."<<endl;
```

```
  }
};
int main(void) {
   Shape *s;                    //  base class pointer.
   Shape sh;                    // base class object.
      Rectangle rec;
       Circle cir;
     s=&sh;
    s->draw();
      s=&rec;
    s->draw();
   s=?
   s->draw();
}
```

**Output:**

```
drawing...
drawing rectangle...
drawing circle...
```

# Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
#include <iostream>
using namespace std;
class Animal {                          //  base class declaration.
   public:
   string color = "Black";
};
class Dog: public Animal                // inheriting Animal class.
{
 public:
   string color = "Grey";
};
int main(void) {
    Animal d= Dog();
   cout<<d.color;
}
```

**Output:**

```
Black
```

# C++ Overloading (Function and Operator)

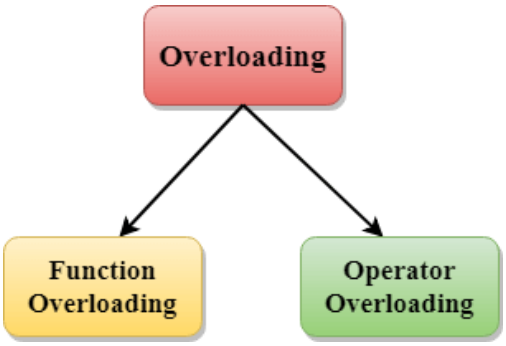
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

# Types of overloading in C++ are:

- Function overloading
- Operator overloading




## C++ 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.

## C++ Function Overloading 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
```

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

// Program of function overloading with different types of arguments.

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);


int mul(int a,int b)
{
    return a*b;
}
float mul(double x, int y)
{
    return x*y;
}
int main()
{
    int r1 = mul(6,7);
    float r2 = mul(0.2,3);
    std::cout << "r1 is : " <<r1<< std::endl;
    std::cout <<"r2 is : "  <<r2<< std::endl;
    return 0;
}
```

**Output:**

```
r1 is : 42
r2 is : 0.6
```

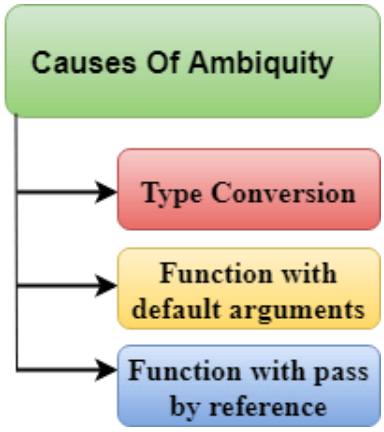
# Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

**Causes of Function Overloading:**

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.




- Type Conversion:

**Let's see a simple example.**

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{
   std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
   fun(12);
   fun(1.2);
   return 0;
}
```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

- Function with Default Arguments

**Let's see a simple example.**

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
   fun(12);

   return 0;
}
```

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

Let's see a simple example.

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
```

```
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

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 &).

## C++ 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 of Operator Overloading

```
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 atleast 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.

## C++ Operators Overloading 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
```

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

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

   int x;
     public:
     A(){}
   A(int i)
   {
      x=i;
   }
   void operator+(A);
   void display();
};

void A :: operator+(A a)
{

   int m = x+a.x;
   cout<<"The result of the addition of two objects is : "<<m;

}
int main()
{
   A a1(5);
   A a2(4);
   a1+a2;
   return 0;
}
```

**Output:**

```
The result of the addition of two objects is : 9
```

# C++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

## C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

```
#include <iostream>
```

```
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
class Dog: public Animal
{
 public:
 void eat()
   {
     cout<<"Eating bread...";
   }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

Output:

```
Eating bread...
```

# C++ 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.

```
#include <iostream>
using namespace std;
class A
{
  int x=5;
   public:
   void display()
   {
      std::cout << "Value of x is : " << x<<std::endl;
   }
};
class B: public A
{
   int y = 10;
   public:
   void display()
   {
      std::cout << "Value of y is : " <<y<< std::endl;
   }
};
int main()
{
   A *a;
   B b;
   a = &b;
  a->display();
   return 0;
}
```

**Output:**

```
Value of x is : 5
```

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class.

Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

## C++ virtual function 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;
```

**Let's see a simple 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::end
l;
   }
};
int main()
{
   Base *bptr;
   //Base b;
   Derived d;
   bptr = &d;
   bptr->show();
   return 0;
}
```

**Output:**

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

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

## Virtual destructor

Deleting a derived class object using a pointer to a base class, the base class should be defined with a virtual destructor.

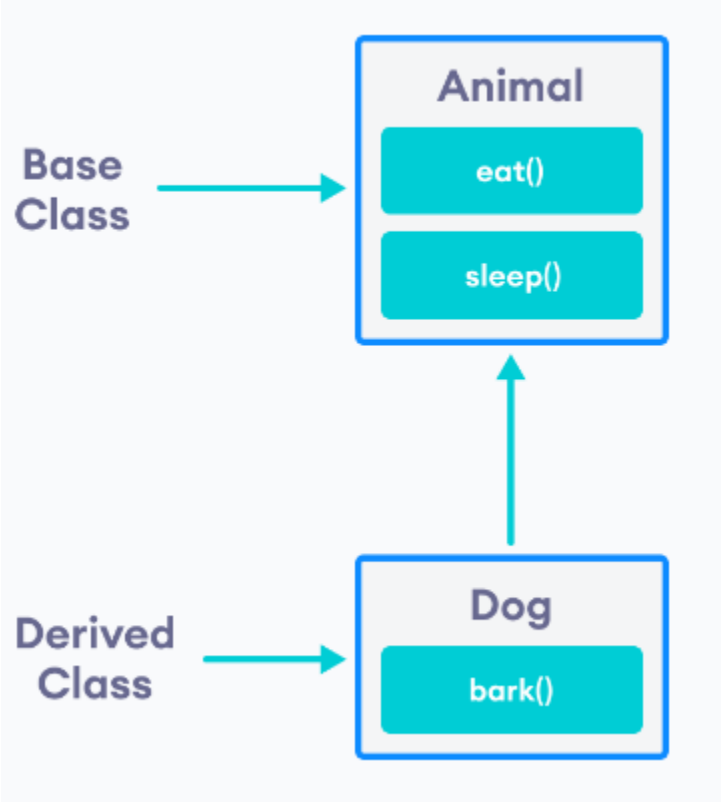
## Example Code

```
#include<iostream>
using namespace std;
class b {
   public:
      b() {
         cout<<"Constructing base \n";
```

```
      }
      virtual ~b() {
         cout<<"Destructing base \n";
      }
};
class d: public b {
   public:
      d() {
         cout<<"Constructing derived \n";
      }
      ~d() {
         cout<<"Destructing derived \n";
      }
};
int main(void) {
   d *derived = new d();
   b *bptr = derived;
   delete bptr;
   return 0;
}
```

## Output

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```




is-a relationship

Inheritance is an **is-a relationship**. We use inheritance only if an **is-a relationship** is present between the two classes.
Here are some examples:

- A car is a vehicle.
- Orange is a fruit.
- A surgeon is a doctor.
- A dog is an animal.

# Purpose of Inheritance in C++

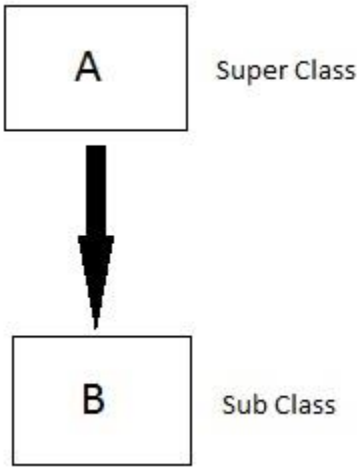
1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

# Types of Inheritance in C++

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

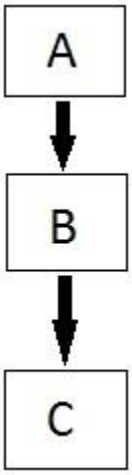
## Single Inheritance in C++

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



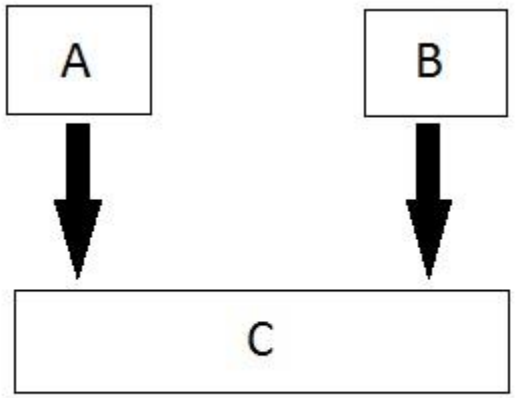

## Multilevel Inheritance in C++

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



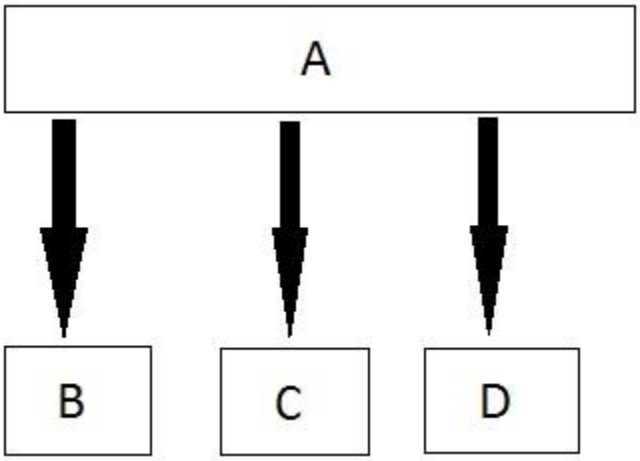
# Multiple Inheritance in C++

In this type of inheritance a single derived class may inherit from two or more than two base classes.



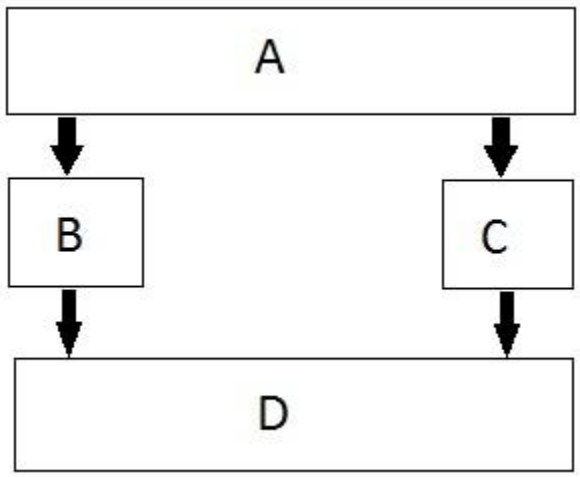

# Hierarchical Inheritance in C++

In this type of inheritance, multiple derived classes inherits from a single base class.



# Hybrid (Virtual) Inheritance in C++

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.




```
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

```
OUTPUT:
Tuut, tuut!
Ford Mustang
```

# Multilevel Inheritance

A class can also be derived from one class, which is already derived from another class.

In the following example, `MyGrandChild` is derived from class `MyChild` (which is derived from `MyClass`).

## Example

```
// Base class (parent)
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class." ;
    }
};

// Derived class (child)
class MyChild: public MyClass {
};

// Derived class (grandchild)
class MyGrandChild: public MyChild {
};

int main() {
  MyGrandChild myObj;
  myObj.myFunction();
  return 0;
}
```

# Multiple Inheritance

A class can also be derived from more than one base class, using a **comma-separated list:**

## Example

```
// Base class
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in parent class." ;
    }
};

// Another base class
```

```
class MyOtherClass {
  public:
    void myOtherFunction() {
      cout << "Some content in another class." ;
    }
};

// Derived class
class MyChildClass: public MyClass, public MyOtherClass {
};

int main() {
  MyChildClass myObj;
  myObj.myFunction();
  myObj.myOtherFunction();
  return 0;
}
```

# Access Specifiers

You learned from the Access Specifiers chapter that there are three specifiers available in C++. Until now, we have only used `public` (members of a class are accessible from outside the class) and `private` (members can only be accessed within the class). The third specifier, `protected`, is similar to `private`, but it can also be accessed in the **inherited** class:

## Example

```
// Base class
class Employee {
  protected: // Protected access specifier
    int salary;
};

// Derived class
class Programmer: public Employee {
  public:
    int bonus;
    void setSalary(int s) {
      salary = s;
    }
    int getSalary() {
      return salary;
    }
};

int main() {
  Programmer myObj;
  myObj.setSalary(50000);
```

```
  myObj.bonus = 15000;
  cout << "Salary: " << myObj.getSalary() << "\n";
  cout << "Bonus: " << myObj.bonus << "\n";
  return 0;
}
```