C++ Inheritance

Inheritance is a mechanism of acquiring the features and behaviors of a class by another class. The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class. Inheritance implements the IS-A relationship.

Different Types of Inheritance
Depending on the way the class is derived or how many base classes a class inherits, we have the following types of inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

1) Single Inheritance

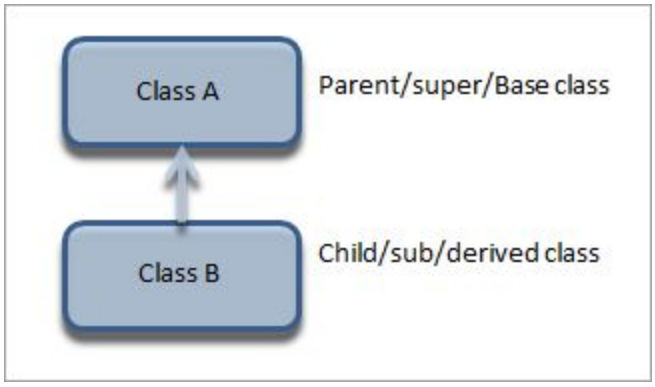
In single inheritance, a class derives from one base class only. This means that there is only one subclass that is derived from one superclass.

Single inheritance is usually declared as follows:

```
class subclassname : accessspecifier superclassname {


        //class specific code;


};
```



Given below is a complete Example of Single Inheritance.

```
#include <iostream>

#include <string>

using namespace std;

class Animal

{

   string name="";

   public:

   int tail=1;

   int legs=4;
```

```
};

class Dog : public Animal

{

   public:

   void voiceAction()

   {

      cout<<"Barks!!!";

   }

};

int main()

{

   Dog dog;
```

```
    cout<<"Dog has "<<dog.legs<<" legs"<<endl;

    cout<<"Dog has "<<dog.tail<<" tail"<<endl;

    cout<<"Dog ";

    dog.voiceAction();

}
```

Output:

Dog has 4 legs

Dog has 1 tail

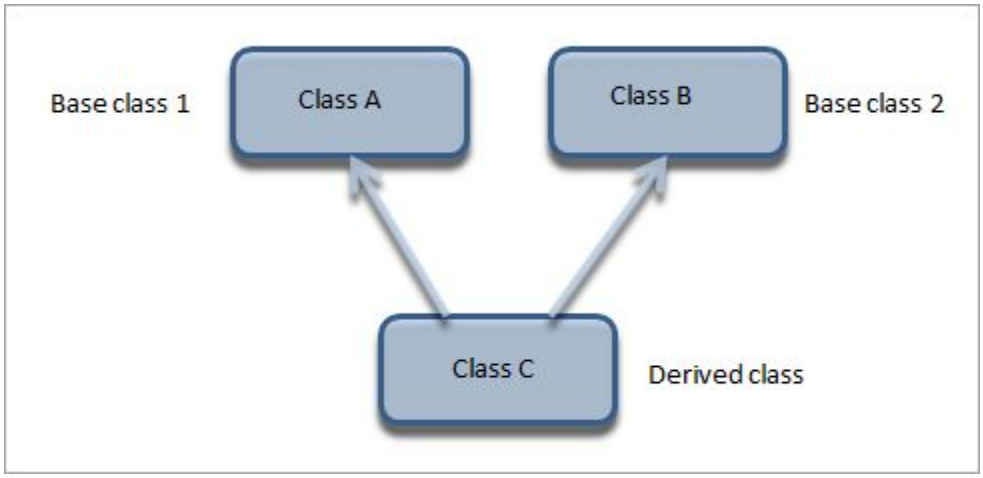
Dog Barks!!!

We have a class Animal as a base class from which we have derived a subclass dog. Class dog inherits all the members of Animal class and can be extended to include its own properties, as seen from the output.

Single inheritance is the simplest form of inheritance.

#2) Multiple Inheritance

Multiple Inheritance is pictorially represented below.



Multiple inheritance is a type of inheritance in which a class derives from more than one classes. As shown in the above diagram, class C is a subclass that has class A and class B as its parent.

In a real-life scenario, a child inherits from its father and mother. This can be considered as an example of multiple inheritance.

We present the below program to demonstrate Multiple Inheritance.

```
#include <iostream>

using namespace std;

//multiple inheritance example

class student_marks {

protected:

int rollNo, marks1, marks2;

public:

void get() {

cout << "Enter the Roll No.: "; cin >> rollNo;

cout << "Enter the two highest marks: "; cin >> marks1 >> marks2;

   }

};

class cocurricular_marks {
```

```
protected:

int comarks;

public:

void getsm() {

cout << "Enter the mark for CoCurricular Activities: "; cin >>
comarks;

   }

};



//Result is a combination of subject_marks and cocurricular
activities marks

class Result : public student_marks, public cocurricular_marks {

   int total_marks, avg_marks;

   public:

   void display()
```

```
   {

      total_marks = (marks1 + marks2 + comarks);

      avg_marks = total_marks / 3;

      cout << "\nRoll No: " << rollNo << "\nTotal marks: " <<
total_marks;

      cout << "\nAverage marks: " << avg_marks;

   }

};

int main()

{

   Result res;

res.get(); //read subject marks

res.getsm(); //read cocurricular activities marks

res.display(); //display the total marks and average marks
```

```
}
```

Output:

Enter the Roll No.: 25

Enter the two highest marks: 40 50

Enter the mark for CoCurricular Activities: 30

Roll No: 25

Total marks: 120

Average marks: 40

In the above example, we have three classes i.e. student_marks, cocurricular_marks, and Result. The class student_marks reads the subject mark for the student. The class cocurricular_marks reads the student's marks in co-curricular activities.

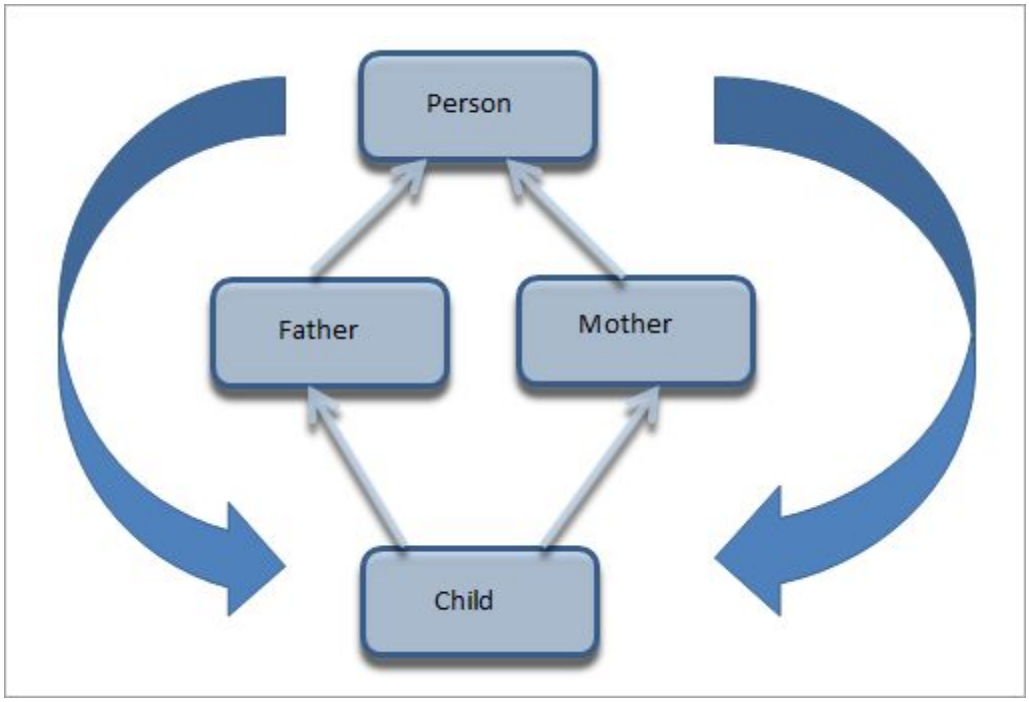
The Result class calculates the total_marks for the student along with the average marks.

In this model, Result class is derived from student_marks and cocurricular_marks as we calculate Result from the subject as well as co-curricular activities marks.

This exhibits multiple inheritances.

Diamond problem

Diamond Problem is pictorially represented below:



Here, we have a child class inheriting two classes Father and Mother. These two classes, in turn, inherit the class Person.

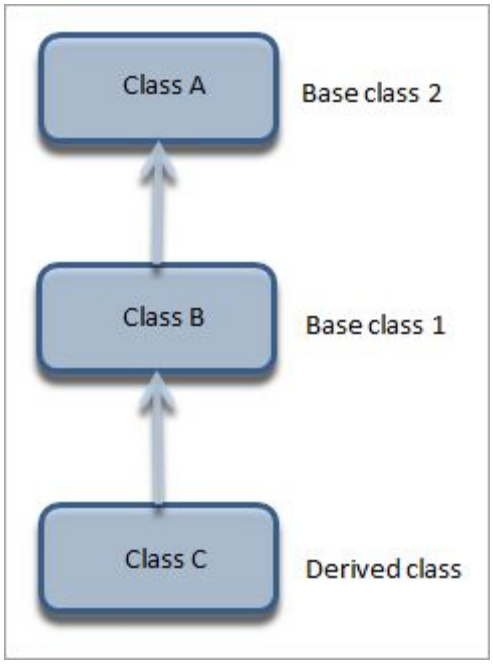
As shown in the figure, class Child inherits the traits of class Person twice i.e. once from Father and the second time from Mother. This gives rise to ambiguity as the compiler fails to understand which way to go.

Since this scenario arises when we have a diamond-shaped inheritance, this problem is famously called "**The Diamond Problem**".

The Diamond problem implemented in C++ results in ambiguity error at compilation. We can resolve this problem by making the root base class as virtual. We will learn more about the "virtual" keyword in our upcoming tutorial on polymorphism.

#3) Multilevel Inheritance

Multilevel inheritance is represented below.



In multilevel inheritance, a class is derived from another derived class. This inheritance can have as many levels as long as our implementation doesn't go wayward. In the above diagram, class C is derived from Class B. Class B is in turn derived from class A.

Let us see an example of Multilevel Inheritance.

```
#include <iostream>

#include <string>

using namespace std;

class Animal

{

   string name="";

   public:

   int tail=1;

   int legs=4;




};

class Dog : public Animal

{
```

```
   public:

   void voiceAction()

   {

      cout<<"Barks!!!";

   }

};

class Puppy:public Dog{

   public:

   void weeping()

   {

      cout<<"Weeps!!";

   }

};
```

```
int main()

{

   Puppy puppy;

cout<<"Puppy has "<<puppy.legs<<" legs"<<endl;

cout<<"Puppy has "<<puppy.tail<<" tail"<<endl;

cout<<"Puppy ";

puppy.voiceAction();

cout<<" Puppy ";

puppy.weeping();

}
```

Output:

Puppy has 4 legs

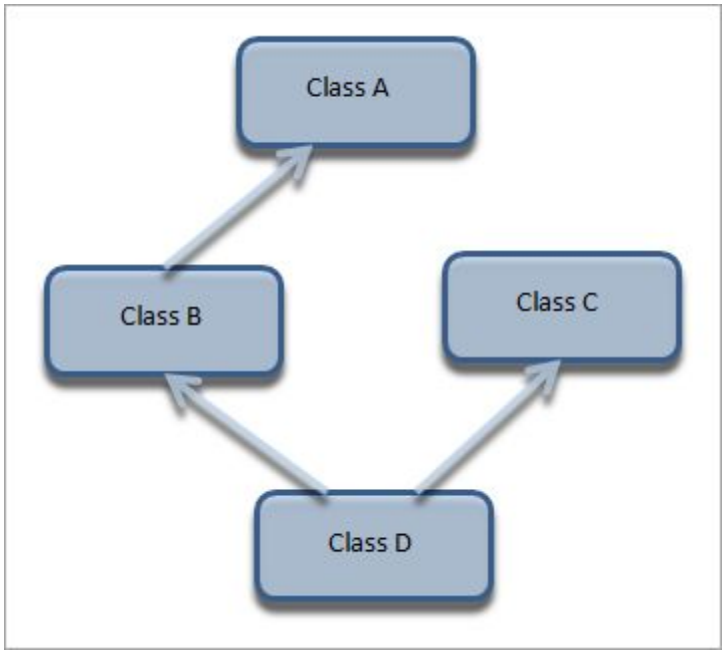
Puppy has 1 tail

Puppy Barks!!! Puppy Weeps!!

Here we modified the example for Single inheritance such that there is a new class Puppy which inherits from the class Dog that in turn inherits from class Animal. We see that the class Puppy acquires and uses the properties and methods of both the classes above it.

#4) Hybrid Inheritance

Hybrid inheritance is depicted below.



Hybrid inheritance is usually a combination of more than one type of inheritance. In the above representation, we have multiple inheritance (B, C, and D) and multilevel inheritance (A, B and D) to get a hybrid inheritance.

Let us see an example of Hybrid Inheritance.

```
#include <iostream>

#include <string>

using namespace std;

//Hybrid inheritance = multilevel + multilpe

class student{ //First base Class

                int id;

                string name;

                public:

                void getstudent(){

                                cout << "Enter student Id and
student name"; cin >> id >> name;

   }

};

class marks: public student{ //derived from student
```

```
                protected:

                int marks_math,marks_phy,marks_chem;

                public:

                void getmarks(){

                                cout << "Enter 3 subject marks:";
cin >>marks_math>>marks_phy>>marks_chem;

   }

};

class sports{

              protected:

              int spmarks;

              public:

              void getsports(){

                            cout << "Enter sports marks:";
```

```
cin >> spmarks;

   }

};

class result : public marks, public sports{//Derived class by
multiple inheritance//

                int total_marks;

                float avg_marks;

                public :

               void display(){

total_marks=marks_math+marks_phy+marks_chem;

                            avg_marks=total_marks/3.0;

                           cout << "Total marks =" << total_marks
<< endl;
```

```
                            cout << "Average marks =" << avg_marks
<< endl;

                            cout << "Average + Sports marks =" <<
avg_marks+spmarks;

   }

};

  int main(){

                result res;//object//

                res.getstudent();

                res.getmarks();

                res.getsports();

                res.display();

               return 0;

}
```

Output:

Enter student Id and student name 25 Ved

Enter 3 subject marks:89 88 87

Enter sports marks:40

Total marks =264

Average marks =88

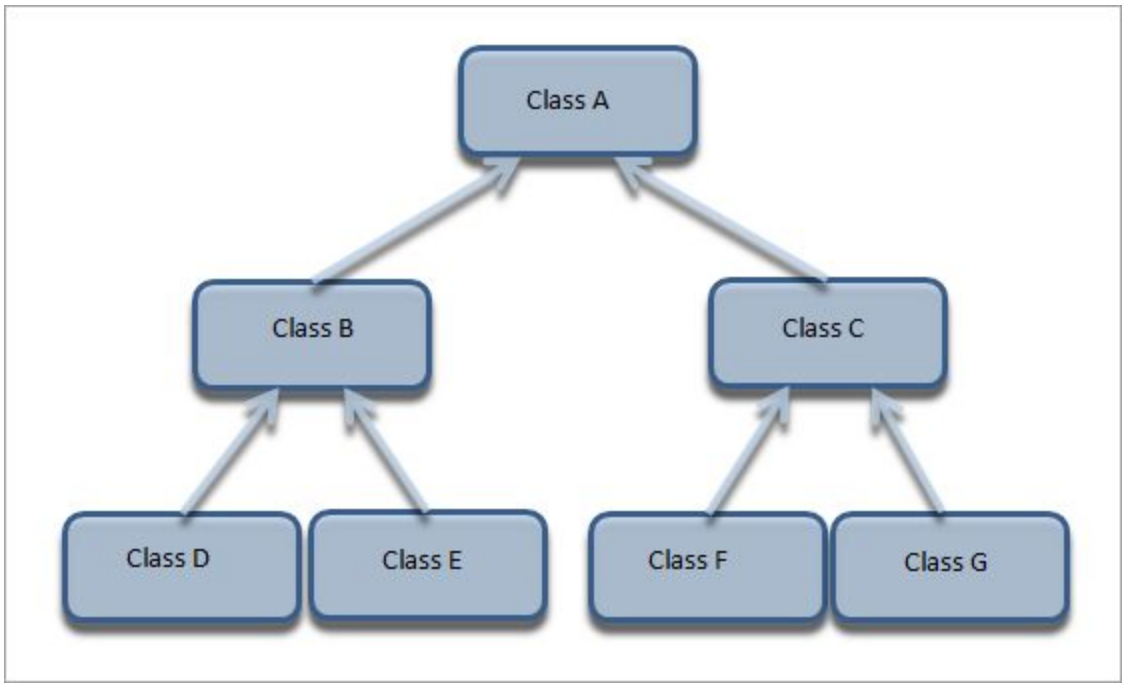
Average + Sports marks =128

Here we have four classes i.e. Student, Marks, Sports, and Result. Marks are derived from the student class. The class Result derives from Marks and Sports as we calculate the result from the subject marks as well as sports marks.

The output is generated by creating an object of class Result that has acquired the properties of all the three classes.

Note that in hybrid inheritance as well, the implementation may result in “Diamond Problem” which can be resolved using “virtual” keyword as mentioned previously.

5) Hierarchical Inheritance



In hierarchical inheritance, more than one class inherits from a single base class as shown in the representation above. This gives it a structure of a hierarchy.

Given below is the Example demonstrating Hierarchical Inheritance.

```
#include <iostream>

using namespace std;

//hierarchical inheritance example

class Shape                 // shape class -> base class

{

public:

int x,y;

void get_data(int n,int m) {

      x= n;

      y = m;

   }

};

class Rectangle : public Shape // inherit Shape class

{
```

```
public:

int area_rect() {

int area = x*y;

return area;

   }

};

class Triangle : public Shape // inherit Shape class

{

public:

int triangle_area() {

float area = 0.5*x*y;

return area;

   }

};
```

```
class Square : public Shape // inherit Shape class

{

public:

int square_area() {

float area = 4*x;

return area;

   }

};

int main()

{ Rectangle r;

   Triangle t;

   Square s;

   int length,breadth,base,height,side;

   //area of a Rectangle
```

```
   std::cout << "Enter the length and breadth of a rectangle: ";
cin>>length>>breadth;


   r.get_data(length,breadth);


   int rect_area = r.area_rect();


   std::cout << "Area of the rectangle = " <<rect_area<< std::endl;


   //area of a triangle


   std::cout << "Enter the base and height of the triangle: ";
cin>>base>>height;


   t.get_data(base,height);


   float tri_area = t.triangle_area();


   std::cout <<"Area of the triangle = " << tri_area<<std::endl;


   //area of a Square


   std::cout << "Enter the length of one side of the square: ";
cin>>side;


   s.get_data(side,side);


   int sq_area = s.square_area();
```

```
    std::cout <<"Area of the square = " << sq_area<<std::endl;

    return 0;

}
```

Output:

Enter the length and breadth of a rectangle: 10 5

Area of the rectangle = 50

Enter the base and height of the triangle: 4 8

Area of the triangle = 16

Enter the length of one side of the square: 5

Area of the square = 20

The above example is a classic example of class Shape. We have a base class Shape and three classes i.e. rectangle, triangle, and square are derived from it.

We have a method to read data in the Shape class while each derived class has its own method to calculate area. In the main function, we read data for each object and then calculate the area.

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

Order of 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.

```
// 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;
}
```

Output:

```
Inside base class
Inside sub class
```

Example 2:

```
// C++ program to show the order of constructor calls
// in Multiple Inheritance

#include <iostream>
using namespace std;

// first base class
class Parent1
{

	public:

	// first base class's Constructor
	Parent1()
	{
		cout << "Inside first base class" << endl;
	}
};

// second base class
class Parent2
{
	public:

	// second base class's Constructor
	Parent2()
```

```
	{
		cout << "Inside second base class" << endl;
	}
};

// child class inherits Parent1 and Parent2
class Child : public Parent1, public Parent2
{
	public:

	// child class's Constructor
	Child()
	{
		cout << "Inside child class" << endl;
	}
};

// main function
int main() {

	// creating object of class Child
	Child obj1;
	return 0;
}
```

Output:

```
Inside first base class
Inside second base class
Inside child class
```

Example 3:

```
// C++ program to show how to call parameterised Constructor
// of base class when derived class's Constructor is called

#include <iostream>
using namespace std;

// base class
class Parent
{
```

```
        public:

        // base class's parameterised constructor
        Parent(int i)
        { int x =i;
                cout << "Inside base class's parameterised constructor" << endl;
        }
};


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

        // sub class's parameterised constructor
        Child(int j): Parent(j)
        {
                cout << "Inside sub class's parameterised constructor" << endl;
        }
};

// main function
int main() {

        // creating object of class Child
        Child obj1(10);
        return 0;
}
```

Output:

```
Inside base class's parameterised constructor
Inside sub class's parameterised constructor
```

Containership in C++

We can create an object of one class into another and that object will be a member of the class. This type of relationship between classes is known as **containership** or **has_a** relationship as one class contain the object of another class. And the class which contains the object and members of another class in this kind of relationship is called a **container class**.

The object that is part of another object is called as contained object, whereas object that contains another object as its part or attribute is called container object.

Difference between containership and inheritance

Containership

-> When features of existing class are wanted inside your new class, but, not its interface

for eg->

1)computer system has a hard disk

2)car has an Engine, chassis, steering wheels.

Inheritance

-> When you want to force the new type to be the same type as the base class.

for eg->

1)computer system is an electronic device

2)Car is a vehicle

Employees can be of Different types as can be seen above. It can be a developer, an HR manager, a sales executive, and so on. Each one of them belongs to Different problem domain but the basic Characteristics of an employee are common to all.

Syntax for Containership:

```
// Class that is to be contained
class first {
    .
    .
};

// Container class
class second {

    // creating object of first
    first f;
    .
    .
};
```

Important Points:

- Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.
- To call the parameterised constructor of base class inside the parameterised consructor of sub class, we have to mention it explicitly.
- The parameterised constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterised constructor of sub class.

Example 1:// CPP program to illustrate concept of Containership

```
#include <iostream>
using namespace std;

class first {
public:
```

```
        void showf()
        {
                cout << "Hello from first class\n";
        }
};

// Container class
class second {
        // creating object of first
        first f;

public:
        // constructor
        second()
        {
                // calling function of first class
                f.showf();
        }
};

int main()
{
        // creating object of second
        second s;
}
```

Output:

```
Hello from first class
```

```
#include <iostream>

using namespace std;



class first {

public:

   first()

   {

       cout << "Hello from first class\n";
```

```
    }
};

// Container class
class second {
    // creating object of first
    first f;

public:
    // constructor
    second()
    {
        cout << "Hello from second class\n";
    }
};
int main()
{
    // creating object of second
    second s;
}
```

Output:

```
Hello from first class
Hello from second class
```

Example

```
#include<iostream>
using namespace std;


class cDate
{
    int mDay,mMonth,mYear;
public:
    cDate()
    {
        mDay = 10;
        mMonth = 11;
        mYear = 1999;
    }
    cDate(int d,int m ,int y)
    {
        mDay = d;
        mMonth = m;
        mYear = y;
    }
    void display()
    {
        cout << "day" << mDay << endl;
```

```
        cout <<"Month" << mMonth << endl;
        cout << "Year" << mYear << endl;
    }
};
// Container class
class cEmployee
{
protected:
    int mId;
    int mBasicSal;
    // Contained Object
    cDate mBdate;
public:
    cEmployee()
    {
        mId = 1;
        mBasicSal = 10000;
        mBdate = cDate();
    }
    cEmployee(int, int, int, int, int);
    void display();
};
```

```
cEmployee :: cEmployee(int i, int sal, int d, int m, int y)
{
    mId = i;
    mBasicSal = sal;
    mBdate = cDate(d,m,y);
}
void cEmployee::display()
{
    cout << "Id : " << mId << endl;
    cout << "Salary :" <<mBasicSal << endl;
    mBdate.display();
}



int main()
{
    // Default constructor call
    cEmployee e1;
    e1.display();
    // Parameterized constructor called
    cEmployee e2(2,20000,11,11,1999);
```

```
    e2.display();
    return 0;
}
```

output

```
Id : 1
Salary :10000
day 10
Month 11
Year 1999
Id : 2
Salary :20000
day 11
Month 11
Year 1999
```

Virtual Function in C++

Normal member function accessed with pointers:

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

```
// CPP program to illustrate
// concept of Virtual Functions

#include <iostream>
using namespace std;

class base {
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }

    void show()
    {
        cout << "show base class" << endl;
    }
};

class derived : public base {
public:
    void print()
    {
        cout << "print derived class" << endl;
    }

    void show()
    {
        cout << "show derived class" << endl;
    }
};

int main()
{
    base* bptr;
    derived d;
    bptr = &d;

    // virtual function, binded at runtime
```

```
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Output:

```
print derived class
show base class
```

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer ‘bptr’ contains the address of object ‘d’ of derived class.

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is *print derived class* as pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time(output is *show base class* as pointer is of base type).

NOTE: If we have created a virtual function in the base class and it is being overridden in the derived class then we don’t need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

Working of virtual functions(concept of VTABLE and VPTR)

If a class contains a virtual function then compiler itself does two things:

1. If object of that class is created then a **virtual pointer(VPTR)** is inserted as a data member of the class to point to VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
2. Irrespective of object is created or not, **a static array of function pointer called VTABLE** where each cell contains the address of each virtual function contained in that class.

```
// CPP program to illustrate
// working of Virtual Functions
#include <iostream>
using namespace std;

class base {
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};

int main()
{
    base* p;
    derived obj1;
    p = &obj1;

    // Early binding because fun1() is non-virtual
    // in base
    p->fun_1();

    // Late binding (RTP)
    p->fun_2();

    // Late binding (RTP)
    p->fun_3();

    // Late binding (RTP)
    p->fun_4();

    // Early binding but this function call is
    // illegal(produces error) becasue pointer
```

```
    // is of base type and function is of
    // derived class
    // p->fun_4(5);
}
```

Output:

```
base-1
derived-2
base-3
Base-4
```

Nested Classes in C++

A nested class is a class that is declared in another class. The nested class is also a member variable of the enclosing class and has the same access rights as the other members. However, the member functions of the enclosing class have no special access to the members of a nested class.

A program that demonstrates nested classes in C++ is as follows.

```
#include<iostream>
using namespace std;
class A {
   public:
   class B {
      private:
      int num;
      public:
      void getdata(int n) {
         num = n;
      }
      void putdata() {
         cout<<"The number is "<<num;
      }
   };
};
int main() {
   cout<<"Nested classes in C++"<< endl;
   A :: B obj;
   obj.getdata(9);
   obj.putdata();
   return 0;

}
```

Output

```
Nested classes in C++
The number is 9
```

In the above program, class B is defined inside the class A so it is a nested class. The class B contains a private variable num and two public functions getdata() and putdata(). The function getdata() takes the data and the function putdata() displays the data. This is given as follows.