**Unit – III/ Inheritance:**

**Access Specifiers**

Access modifiers are used to implement an important feature of Object-Oriented Programming known as [**Data Hiding**](https://practice.geeksforgeeks.org/problems/what-is-data-hiding). Consider a real-life example:

The Indian secret informatic system having 10 senior members have some top secret regarding national security. So we can think that 10 people as class data members or member functions who can directly access secret information from each other but anyone can’t access this information other than these 10 members i.e. outside people can’t access information directly without having any privileges. This is what data hiding is.  
Access Modifiers or Access Specifiers in a [class](https://www.geeksforgeeks.org/c-classes-and-objects/) are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Let us now look at each one these access modifiers in details:

1. **Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.  
   **Example:**

|  |
| --- |
| // C++ program to demonstrate public  // access modifier    #include<iostream>  using namespace std;    // class definition  class Circle  {      public:          double radius;            double  compute\_area()          {              return 3.14\*radius\*radius;          }    };    // main function  int main()  {      Circle obj;        // accessing public datamember outside class      obj.radius = 5.5;        cout << "Radius is: " << obj.radius << "\n";      cout << "Area is: " << obj.compute\_area();      return 0;  } |

**Output:**

Radius is: 5.5

Area is: 94.985

In the above program the data member *radius* is public so we are allowed to access it outside the class.

1. **Private**: The class members declared as *private* can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the [friend functions](https://www.geeksforgeeks.org/friend-class-function-cpp/) are allowed to access the private data members of a class.  
   **Example:**

|  |
| --- |
| // C++ program to demonstrate private  // access modifier    #include<iostream>  using namespace std;    class Circle  {      // private data member      private:          double radius;        // public member function      public:          double  compute\_area()          {   // member function can access private              // data member radius              return 3.14\*radius\*radius;          }    };    // main function  int main()  {      // creating object of the class      Circle obj;        // trying to access private data member      // directly outside the class      obj.radius = 1.5;        cout << "Area is:" << obj.compute\_area();      return 0;  } |

The output of above program will be a compile time error because we are not allowed to access the private data members of a class directly outside the class.  
**Output**:

In function 'int main()':

11:16: error: 'double Circle::radius' is private

double radius;

^

31:9: error: within this context

obj.radius = 1.5;

^

However, we can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

|  |
| --- |
| // C++ program to demonstrate private  // access modifier    #include<iostream>  using namespace std;    class Circle  {      // private data member      private:          double radius;        // public member function      public:          void compute\_area(double r)          {   // member function can access private              // data member radius              radius = r;                double area = 3.14\*radius\*radius;                cout << "Radius is: " << radius << endl;              cout << "Area is: " << area;          }    };    // main function  int main()  {      // creating object of the class      Circle obj;        // trying to access private data member      // directly outside the class      obj.compute\_area(1.5);          return 0;  } |

**Output**:

Radius is: 1.5

Area is: 7.065

1. **Protected**: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.  
   **Example:**

|  |
| --- |
| // C++ program to demonstrate  // protected access modifier  #include <bits/stdc++.h>  using namespace std;    // base class  class Parent  {      // protected data members      protected:      int id\_protected;    };    // sub class or derived class  class Child : public Parent  {          public:      void setId(int id)      {            // Child class is able to access the inherited          // protected data members of base class            id\_protected = id;        }        void displayId()      {          cout << "id\_protected is: " << id\_protected << endl;      }  };    // main function  int main() {        Child obj1;        // member function of the derived class can      // access the protected data members of the base class        obj1.setId(81);      obj1.displayId();      return 0;  } |

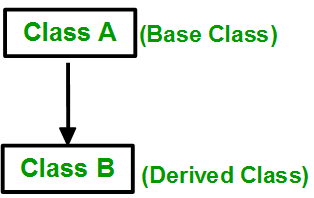
**Output**:

id\_protected is: 81

**Types of Inheritance**

**Types of Inheritance in C++**

1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.

  
**Syntax**:

class subclass\_name : access\_mode base\_class

{

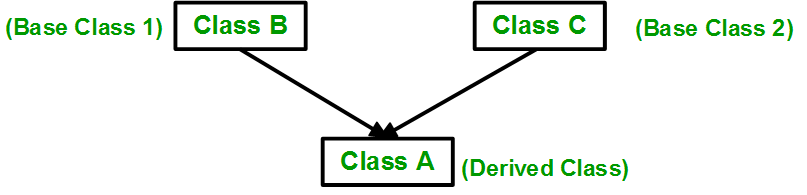
//body of subclass

};

|  |
| --- |
| // C++ program to explain  // Single inheritance  #include <iostream>  using namespace std;    // base class  class Vehicle {    public:      Vehicle()      {        cout << "This is a Vehicle" << endl;      }  };    // sub class derived from two base classes  class Car: public Vehicle{    };    // main function  int main()  {      // creating object of sub class will      // invoke the constructor of base classes      Car obj;      return 0;  } |

Output:

This is a vehicle

1. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.  
   **Syntax**:

class subclass\_name : access\_mode base\_class1, access\_mode base\_class2, ....

{

//body of subclass

};

Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

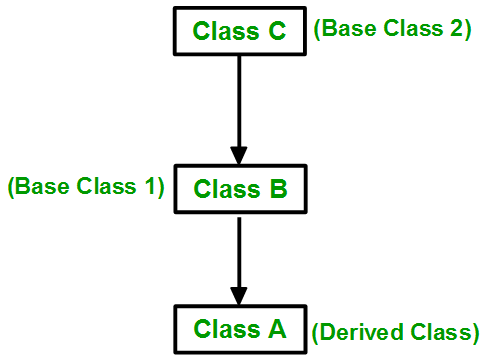
|  |
| --- |
| // C++ program to explain  // multiple inheritance  #include <iostream>  using namespace std;    // first base class  class Vehicle {    public:      Vehicle()      {        cout << "This is a Vehicle" << endl;      }  };    // second base class  class FourWheeler {    public:      FourWheeler()      {        cout << "This is a 4 wheeler Vehicle" << endl;      }  };    // sub class derived from two base classes  class Car: public Vehicle, public FourWheeler {    };    // main function  int main()  {      // creating object of sub class will      // invoke the constructor of base classes      Car obj;      return 0;  } |

Output:

This is a Vehicle

This is a 4 wheeler Vehicle

1. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.



|  |
| --- |
| // C++ program to implement  // Multilevel Inheritance  #include <iostream>  using namespace std;    // base class  class Vehicle  {    public:      Vehicle()      {        cout << "This is a Vehicle" << endl;      }  };  class fourWheeler: public Vehicle  {  public:      fourWheeler()      {        cout<<"Objects with 4 wheels are vehicles"<<endl;      }  };  // sub class derived from two base classes  class Car: public fourWheeler{     public:       car()       {         cout<<"Car has 4 Wheels"<<endl;       }  };    // main function  int main()  {      //creating object of sub class will      //invoke the constructor of base classes      Car obj;      return 0;  } |

output:

This is a Vehicle

Objects with 4 wheels are vehicles

Car has 4 Wheels

1. **Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



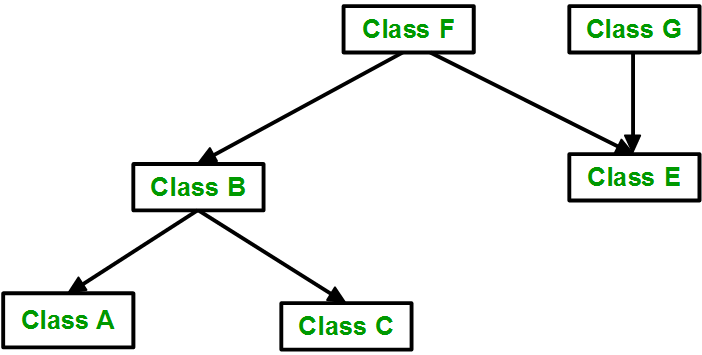
|  |
| --- |
| // C++ program to implement  // Hierarchical Inheritance  #include <iostream>  using namespace std;    // base class  class Vehicle  {    public:      Vehicle()      {        cout << "This is a Vehicle" << endl;      }  };      // first sub class  class Car: public Vehicle  {    };    // second sub class  class Bus: public Vehicle  {    };    // main function  int main()  {      // creating object of sub class will      // invoke the constructor of base class      Car obj1;      Bus obj2;      return 0;  } |

**Output:**

This is a Vehicle

This is a Vehicle

1. **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 inheritance:



|  |
| --- |
| // C++ program for Hybrid Inheritance    #include <iostream>  using namespace std;    // base class  class Vehicle  {    public:      Vehicle()      {        cout << "This is a Vehicle" << endl;      }  };    //base class  class Fare  {      public:      Fare()      {          cout<<"Fare of Vehicle\n";      }  };    // first sub class  class Car: public Vehicle  {    };    // second sub class  class Bus: public Vehicle, public Fare  {    };    // main function  int main()  {      // creating object of sub class will      // invoke the constructor of base class      Bus obj2;      return 0;  } |

**Output:**

This is a Vehicle

Fare of Vehicle

**Ambiguity Resolution in Multiple Inheritance**

### Ambiguity in Multiple Inheritance

The most obvious problem with multiple inheritance occurs during function overriding.

Suppose, two base classes have a same function which is not overridden in derived class.

If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call. For example,

class base1

{

public:

void someFunction()

{...........}

};

class base2

{

void someFunction()

{...........}

};

class derived :public base1,public base2

{

};

int main()

{

derived obj;

obj.someFunction()// Error!

}

This problem can be solved using scope resolution function to specify which function to class either base1 or base2

int main()

{

obj.base1::someFunction( ); // Function of base1 class is called

obj.base2::someFunction(); // Function of base2 class is called.

}

**Constructor Calling (Implicit and Explicit Constructor Call) to Base 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 invokation 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;  } |

Output:

Inside base class

Inside sub class

**Order of constructor call for Multiple Inheritance**

For multiple inheritance order of constructor call is, the base class’s constructors are called in the order of inheritance and then the derived class’s constructor.

|  |
| --- |
| // 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

**Containership and Inheritance**

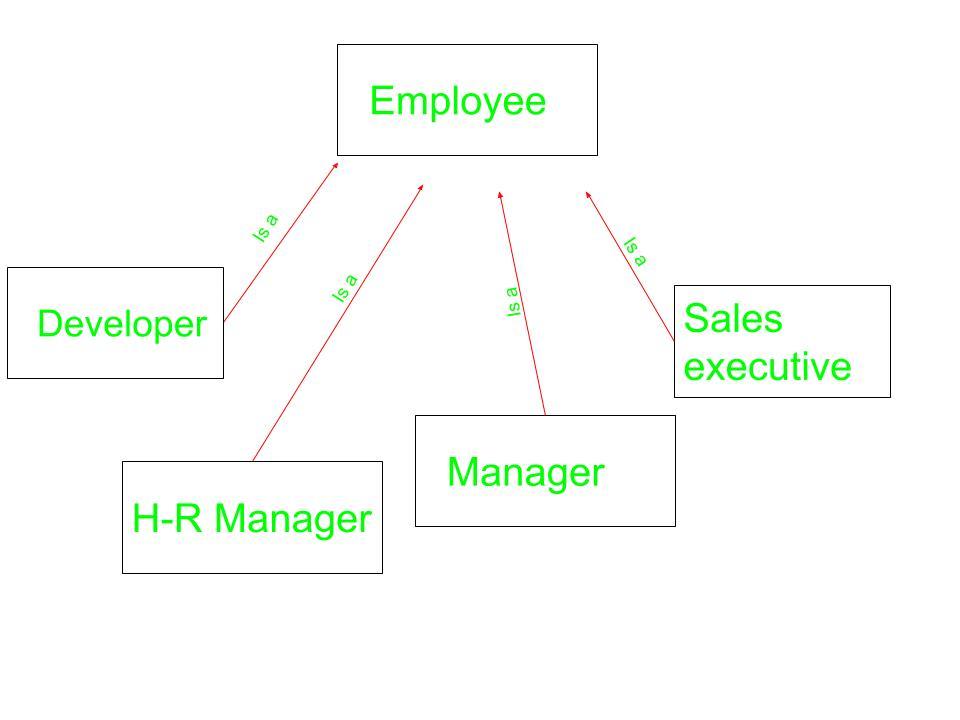
The main difference between **inheritance** and **containership** is that **inheritance** allows using properties and methods of an existing class in the new class while **containership** is another name for composition that describes the ownership between the associated objects.

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 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;

.

.

};

Below examples explain the Containership in C++ in a better way.

**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

**Explanation:**In the class **second** we have an object of class **first**. This is another type of inheritance we are witnessing. This type of inheritance is known as **has\_a** relationship as we say that class **second** has an object of first class **first** as its member. From the object f we call the function of class **first**.

**Example 2:**

|  |
| --- |
| #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

**Explanation:**In this program we have not inherited class **first** into class **second** but as we are having an object of class **first** as a member of class **second**. So when default constructor of class **second** is called, due to presence of object **f** of **first** class in **second**, default constructor of class **first** is called first and then default constructor of class **second** is called .

**Example 3:**

|  |
| --- |
| #include <iostream>  using namespace std;    class first {  private:      int num;    public:      void showf()      {          cout << "Hello from first class\n";          cout << "num = " << num << endl;      }        int& getnum()      {          return num;      }  };    // Container class  class second {      // creating object of first      first f;    public:      // constructor      second()      {          f.getnum() = 20;          f.showf();      }  };    int main()  {      // creating object of second      second s;  } |

**Output:**

Hello from first class

num = 20

**Explanation:**With the help of containership we can only use **public** member/function of the class but not **protected** or **private**. In the **first** class we have returned the reference with the help of **getnum**. Then we show it by a call to **showf**.

**Example 4**

|  |
| --- |
| #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

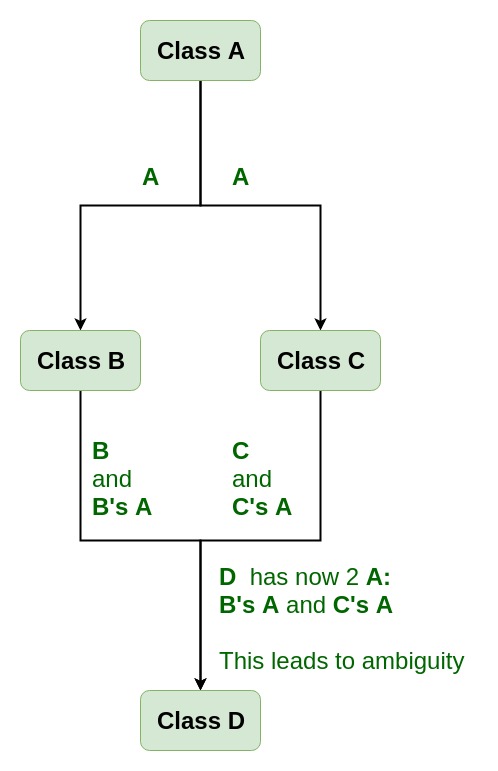
Year 1999

**Virtual Base Class**

Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

**Need for Virtual Base Classes:**

Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

**Example:** To show the need of Virtual Base Class in C++

|  |
| --- |
| #include <iostream>  using namespace std;    class A {  public:      void show()      {          cout << "Hello form A \n";      }  };    class B : public A {  };    class C : public A {  };    class D : public B, public C {  };    int main()  {      D object;      object.show();  } |

**Compile Errors:**

prog.cpp: In function 'int main()':

prog.cpp:29:9: error: request for member 'show' is ambiguous

object.show();

^

prog.cpp:8:8: note: candidates are: void A::show()

void show()

^

prog.cpp:8:8: note: void A::show()

**How to resolve this issue?**  
To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

**Syntax for Virtual Base Classes:**

**Syntax 1:**

class B : virtual public A

{

};

**Syntax 2:**

class C : public virtual A

{

};

**Note:** **virtual** can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class.  
Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

**Example 1**

|  |
| --- |
| #include <iostream>  using namespace std;    class A {  public:      int a;      A() // constructor      {          a = 10;      }  };    class B : public virtual A {  };    class C : public virtual A {  };    class D : public B, public C {  };    int main()  {      D object; // object creation of class d      cout << "a = " << object.a << endl;        return 0;  } |

**Output:**

a = 10

**Explanation :**The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.

**Example 2:**

|  |
| --- |
| #include <iostream>  using namespace std;    class A {  public:      void show()      {          cout << "Hello from A \n";      }  };    class B : public virtual A {  };    class C : public virtual A {  };    class D : public B, public C {  };    int main()  {      D object;      object.show();  } |

**Output:**

Hello from A

**Unit – IV/ Friend and Polymorphism:**

**Friend Function**

**Friend Function** Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:  
a) A method of another class  
b) A global function

|  |
| --- |
| classNode {  private:      intkey;      Node\* next;        /\* Other members of Node Class \*/      friendintLinkedList::search();      // Only search() of linkedList      // can access internal members  }; |

Following are some important points about friend functions and classes:  
**1)** Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

**2)** Friendship is not mutual. If class A is a friend of B, then B doesn’t become a friend of A automatically.

**3)** Friendship is not inherited

**4)** The concept of friends is not there in Java.

**Friend Member Function and Friend Class**

**Friend Class** A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

|  |
| --- |
| classNode {  private:      intkey;      Node\* next;      /\* Other members of Node Class \*/        // Now class  LinkedList can      // access private members of Node      friendclassLinkedList;  }; |

**A friend class** can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a LinkedList class may be allowed to access private members of Node.

We can declare a friend class in C++ by using the friend keyword.

**Syntax:**

***friend class class\_name; // declared in the base class***

***Friend class***

**A simple and complete C++ program to demonstrate friend Class**

#include <iostream>

using namespace std;

class GFG {

private:

int private\_variable;

protected:

int protected\_variable;

public:

GFG()

{

private\_variable = 10;

protected\_variable = 99;

}

// friend class declaration

friend class F;

};

// Here, class F is declared as a

// friend inside class GFG. Therefore,

// F is a friend of class GFG. Class F

// can access the private members of

// class GFG.

class F {

public:

void display(GFG& t)

{

cout << "The value of Private Variable = "

<< t.private\_variable << endl;

cout << "The value of Protected Variable = "

<< t.protected\_variable;

}

};

// Driver code

int main()

{

GFG g;

F fri;

fri.display(g);

return 0;

}

**A simple and complete C++ program to demonstrate friend function of another class**

|  |
| --- |
| #include <iostream>    class B;    class A {  public:      void showB(B&);  };    class B {  private:      int b;    public:      B() { b = 0; }      friend void A::showB(B& x); // Friend function  };    void A::showB(B& x)  {      // Since showB() is friend of B, it can      // access private members of B      std::cout << "B::b = " << x.b;  }    int main()  {      A a;      B x;      a.showB(x);      return 0;  } |

Output:

B::b = 0

**A simple and complete C++ program to demonstrate global friend**

|  |
| --- |
| #include <iostream>    class A {      int a;    public:      A() { a = 0; }        // global friend function      friend void showA(A&);  };    void showA(A& x)  {      // Since showA() is a friend, it can access      // private members of A      std::cout << "A::a=" << x.a;  }    int main()  {      A a;      showA(a);      return 0;  } |

Output:

A::a = 0

**Polymorphism: Function Overloading, Operator Overloading**

**Function overloading** is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading “Function” name should be the same and the arguments should be different. Function overloading can be considered as an example of a [polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/) feature in C++.

If multiple functions having same name but parameters of the functions should be different is known as Function Overloading.  
If we have to perform only one operation and having same name of the functions increases the readability of the program.  
Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the function such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you to understand the behavior of the function because its name differs.

The parameters should follow any one or more than one of the following conditions for Function overloading:

* Parameters should have a different type

*add(int a, int b)  
add(double a, double b)*

**#include <iostream>**

**using namespace std;**

**void add(int a, int b)**

**{**

**cout << "sum = " << (a + b);**

**}**

**void add(double a, double b)**

**{**

**cout << endl << "sum = " << (a + b);**

**}**

**// Driver code**

**int main()**

**{**

**add(10, 2);**

**add(5.3, 6.2);**

**return 0;**

**}**

**Operator Overloading using Friend**

#### Example : Program demonstrating Unary operator overloading using Friend function

#include<iostream>  
using namespace std;  
class UnaryFriend  
{  
     int a=10;  
     int b=20;  
     int c=30;  
     public:  
         void getvalues()  
         {  
              cout<<"Values of A, B & C\n";  
              cout<<a<<"\n"<<b<<"\n"<<c<<"\n"<<endl;  
         }  
         void show()  
         {  
              cout<<a<<"\n"<<b<<"\n"<<c<<"\n"<<endl;  
         }  
         void friend operator-(UnaryFriend &x);      //Pass by reference  
};  
void operator-(UnaryFriend &x)  
{  
     x.a = -x.a;     //Object name must be used as it is a friend function  
     x.b = -x.b;  
     x.c = -x.c;  
}  
int main()  
{  
     UnaryFriend x1;  
     x1.getvalues();  
     cout<<"Before Overloading\n";  
     x1.show();  
     cout<<"After Overloading \n";  
     -x1;  
      x1.show();  
      return 0;  
}

**Output:**  
Values of A, B & C  
10  
20  
30  
  
Before Overloading  
10  
20  
30  
  
After Overloading  
-10  
-20  
-30  
  
In the above program, **operator –** is overloaded using friend function. The **operator()** function is defined as a **Friend function.** The statement **-x1** invokes the **operator()** function. The object **x1** is created of class **UnaryFriend.** The object itself acts as a source and destination object. This can be accomplished by sending reference of an object. The object **x1** is a reference of object **x.** The values of object **x1** are replaced by itself by applying negation.

**Virtual function & Pure Virtual Function**

A virtual function 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](https://www.geeksforgeeks.org/polymorphism-in-c/)
* 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](https://www.geeksforgeeks.org/virtual-destructor/) but it cannot have a virtual constructor.

**Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions**

Consider the following simple program showing run-time behavior of virtual functions.

// 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.

// 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

**Explanation:** Initially, we create a pointer of type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.

Similar concept of **Late and Early Binding**is used as in above example. For fun\_1() function call, base class version of function is called, fun\_2() is overridden in derived class so derived class version is called, fun\_3() is not overridden in derived class and is virtual function so base class version is called, similarly fun\_4() is not overridden so base class version is called.

**NOTE:** fun\_4(int) in derived class is different from virtual function fun\_4() in base class as prototype of both the function is different.

***Pure Virtual Functions***

Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn’t have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A pure virtual function (or abstract function) in C++ is a [virtual function](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/)for which we don’t have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

|  |
| --- |
| // An abstract class  class Test  {      // Data members of class  public:      // Pure Virtual Function      virtual void show() = 0;       /\* Other members \*/  }; |

**A complete example:**  
A pure virtual function is implemented by classes which are derived from a Abstract class. Following is a simple example to demonstrate the same.

|  |
| --- |
| #include<iostream>  using namespace std;    class Base  {     int x;  public:      virtual void fun() = 0;      int getX() { return x; }  };    // This class inherits from Base and implements fun()  class Derived: public Base  {      int y;  public:      void fun() { cout << "fun() called"; }  };    int main(void)  {      Derived d;      d.fun();      return 0;  } |

Output:

fun() called

**Some Interesting Facts:**  
**1)** A class is abstract if it has at least one pure virtual function.  
In the following example, Test is an abstract class because it has a pure virtual function show().

|  |
| --- |
| // pure virtual functions make a class abstract  #include<iostream>  using namespace std;    class Test  {     int x;  public:      virtual void show() = 0;      int getX() { return x; }  };    int main(void)  {      Test t;      return 0;  } |

Output:

Compiler Error: cannot declare variable 't' to be of abstract

type 'Test' because the following virtual functions are pure

within 'Test': note: virtual void Test::show()

**2)** *We can have pointers and references of abstract class type.*  
For example the following program works fine.

|  |
| --- |
| #include<iostream>  using namespace std;    class Base  {  public:      virtual void show() = 0;  };    class Derived: public Base  {  public:      void show() { cout << "In Derived \n"; }  };    int main(void)  {      Base \*bp = new Derived();      bp->show();      return 0;  } |

Output:

In Derived

**3)** *If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.*  
The following example demonstrates the same.

|  |
| --- |
| #include<iostream>  using namespace std;  class Base  {  public:      virtual void show() = 0;  };    class Derived : public Base { };    int main(void)  {    Derived d;    return 0;  } |

Compiler Error: cannot declare variable 'd' to be of abstract type

'Derived' because the following virtual functions are pure within

'Derived': virtual void Base::show()

**4)** *An abstract class can have constructors.*  
For example, the following program compiles and runs fine.

|  |
| --- |
| #include<iostream>  using namespace std;    // An abstract class with constructor  class Base  {  protected:     int x;  public:    virtual void fun() = 0;    Base(int i) { x = i; }  };    class Derived: public Base  {      int y;  public:      Derived(int i, int j):Base(i) { y = j; }      void fun() { cout << "x = " << x << ", y = " << y; }  };    int main(void)  {      Derived d(4, 5);      d.fun();      return 0;  } |

Output:

x = 4, y = 5