Introduction to Procedure Oriented Programming

As we know, prior to object-oriented programming (OOP), programs were written using procedural languages. Procedural languages stress functions. The bigger problems are broken down into smaller sub-problems and written as functions. Procedural languages did not pay attention to data. As a result, the possibility of not addressing the problem in an effective manner was high. Also, as data was almost neglected, data security was easily compromised.

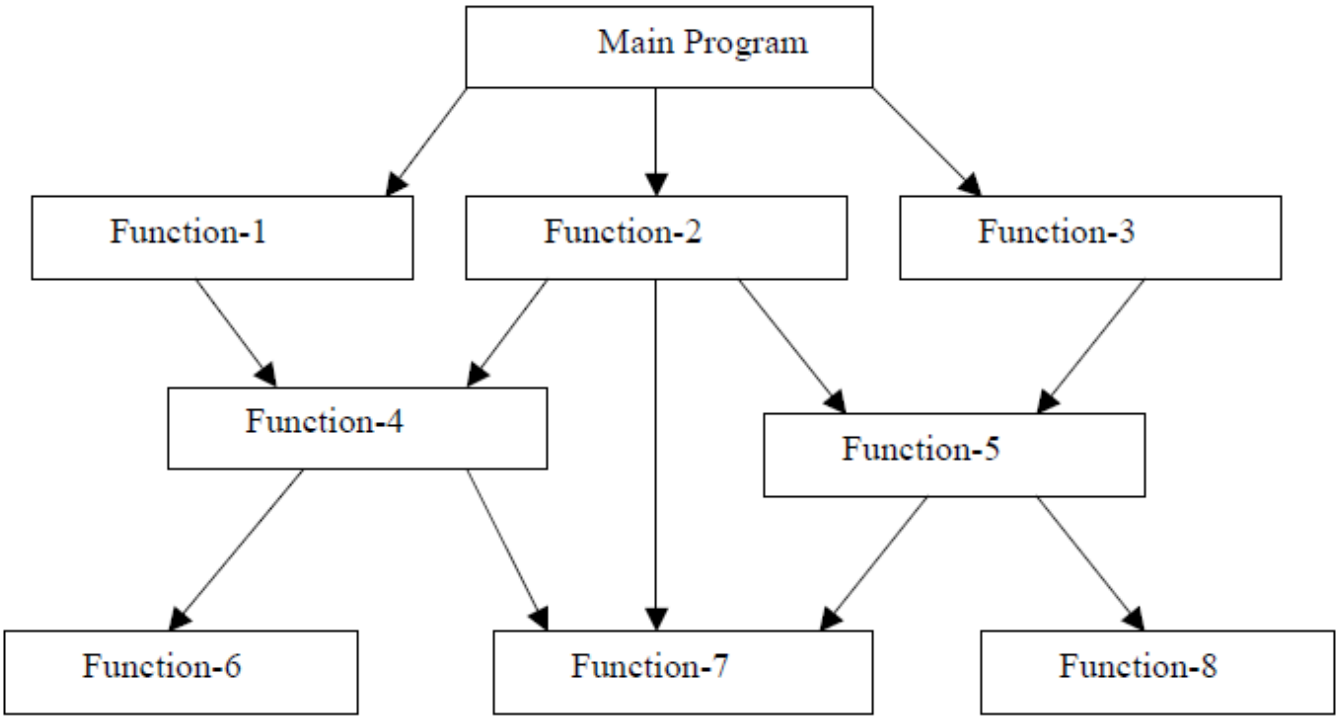


Fig. 1.2 Typical structure of procedural oriented programs

Advantages of POP

1. It emphasis on algorithm (doing this).
2. Large programs are divided into smaller programs known as functions.
3. Function can communicate by global variable.
4. Data move freely from one function to another function.
5. Functions change the value of data at any time from any place. (Functions transform data from one form to another.)
6. It uses top-down programming approach.

Disadvantages of POP

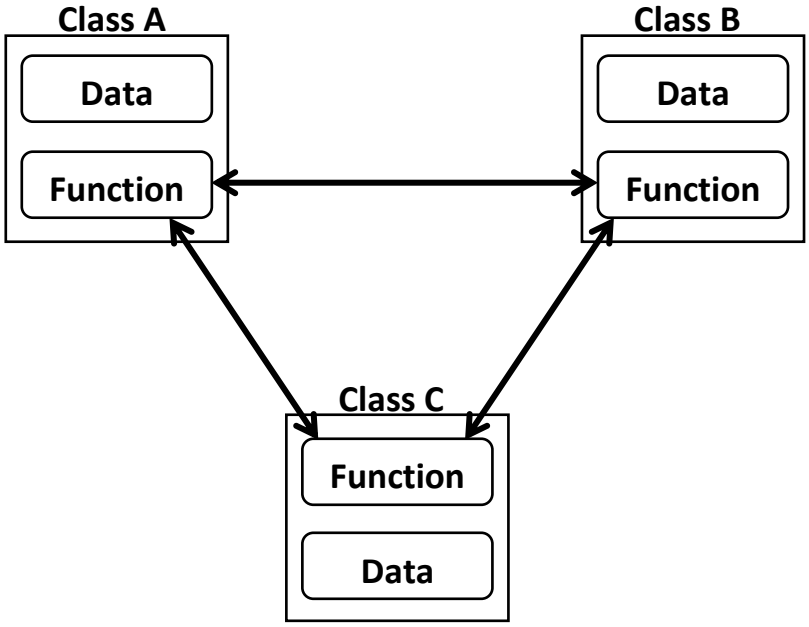
1. Procedural languages are difficult to relate with the real world objects.
2. Procedural codes are very difficult to maintain, if the code grows larger.
3. Procedural languages does not have automatic memory management as like in Java. Hence, it makes the programmer to concern more about the memory management of the program.
4. The data, which is used in procedural languages are exposed to the whole program. So, there is no security for the data.

All these drawbacks of procedural programming were overcome by object-oriented programming.

Object-Oriented Programming In C++

Object-oriented programming revolves around data. The main programming unit of OOP is the object. An object is a representation of a real-time entity and consists of data and methods or functions that operate on data. This way, data, and functions are closely bound and data security is ensured.

In OOP, everything is represented as an object and when programs are executed, the objects interact with each other by passing messages. An object need not know the implementation details of another object for communicating.



Features of OOPs

1. Classes and Objects

A class, on the other hand, is a blueprint of the object. Conversely, an object can be defined as an instance of a class. A class contains a skeleton of the object and does not take any space in the memory.

An object is a basic unit in object-oriented programing. An object contains data and methods or functions that operate on that data. Objects take up space in memory.

2. Abstraction

Abstraction is the process of hiding irrelevant information from the user. For Example, when we are driving the car, first we start the engine by inserting a key. We are not aware of the process that goes on in the background for starting the engine.

Using abstraction in programming, we can hide unnecessary details from the user. By using abstraction in our application, the end user is not affected even if we change the internal implementation.

3. Encapsulation

Encapsulation is the process using which data and the methods or functions operating on them are bundled together. By doing this, data is not easily accessible to the outside world. In OOP we achieve encapsulation by making data members as private and having public functions to access these data members.

4. Inheritance

Using inheritance object of one class can inherit or acquire the properties of the object of another class. Inheritance provides reusability of code.

As such we can design a new class by acquiring the properties and functionality of another class and in this process, we need not modify the functionality of the parent class. We only add new functionality to the class.

5. Polymorphism

Polymorphism means one things in multiple form. Polymorphism is an important feature of OOP and is usually implemented as operator overloading or function overloading. Operator overloading is a process in which an operator behaves differently in different situations. Similarly, in function overloading, the same function behaves differently in different situations.

6. Dynamic Binding

OOP supports dynamic binding in which function call is resolved at runtime. This means that the code to be executed as a result of a function call is decided at runtime. Virtual functions are an example of dynamic binding.

7. Message Passing

In OOP, objects communicate with each other using messages. When objects communicate, information is passed back and forth between the objects. A message generally consists of the object name, method name and actual data that is to be sent to another object.

Why C++ is Partial OOP language?

C++ language was designed with the main intention of using object-oriented features to C language. Although C++ language supports the features of OOP like Classes, objects, inheritance, encapsulation, abstraction, and polymorphism, there are few reasons because of which C++ is classified as a partial object-oriented programming language.

We present a few of these reasons below:

1) Creation of class/objects is Optional

In C++, the main function is mandatory and is always outside the class. Hence, we can have only one main function in the program and can do without classes and objects.

This is the first violation of Pure OOP language where everything is represented as an object.

2) Use of Global Variables

C++ has a concept of global variables that are declared outside the program and can be accessed by any other entity of the program. This violates encapsulation. Though C++ supports encapsulation with respect to classes and objects, it doesn't take care of it in case of global variables.

3) Presence of a Friend Function

C++ supports a friend class or function that can be used to access private and protected members of other classes by making them a friend. This is yet another feature of C++ that violates OOP paradigm.

To conclude, although C++ supports all the OOP features mentioned above, it also provides features that can act as a workaround for these features, so that we can do without them. This makes C++ a partial Object-oriented programming language.

Advantages of OOPs:

1. Reusability
2. Modularity
3. Flexibility
4. Maintainability
5. Data and Information Hiding

Classes and Objects

Classes:

- The classes are the most important feature of C++ that leads to Object Oriented programming.
- Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that class.
- The variables inside class definition are called as data members and the functions are called member functions.
- Class name must start with an uppercase letter. If class name is made of more than one word, then first letter of each word must be in uppercase.

Syntax to Defining a class

```
class ClassName
{
        Access Specifier: Data Member
        Access Specifier: Member Function
};
```

Objects

- Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.
- Each object has different data variables.
- Objects are initialized using special class functions called **Constructors**. We will study about constructors later.
- And whenever the object is out of its scope, another special class member function called **Destructor** is called, to release the memory reserved by the object. C++ doesn't have Automatic Garbage Collector like in JAVA, in C++ Destructor performs this task.

Syntax to Declaring Object

ClassName objectname;

Example 1: If Data member is Public.

```
#include<iostream>
using namespace std;
class Box
{
    public: int length, width, height;
};
int main()
{
    Box b1;
    b1.length=13;
    b1.width=15;
    b1.height=16;
    cout<<endl<<"Box Dimension:"<<b1.length<<"x"<<b1.width<<"x"<<b1.height;
    cout<<endl<<"Box Volume:"<<b1.length*b1.width*b1.height;
}
```

Example 2: If Data Member is Private.

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

```
class Box
{
    private: int length, width, height;
};
int main()
{
    Box b1;
    b1.length=13;
    b1.width=15;
    b1.height=16;
    cout<<endl<<"Box Dimension:"<<b1.length<<"x"<<b1.width<<"x"<<b1.height;
    cout<<endl<<"Box Volume:"<<b1.length*b1.width*b1.height;
}
```

Above program gives error because we cannot access private member outside the class. It can be access only by its own member function.

Defining Member Function:

There are two way defining a function.

1. Inside Class Definition
2. Outside Class Definition

1. Inside Class Definition

```
#include<iostream>
using namespace std;
class Box
{
        private: int length, width, height;
        public:         void setLength(int l)
                        {
                                length=l;
                        }
                        void setWidth(int w)
                        {
                                width=w;
                        }
                        void setHeight(int h)
                        {
                                height=h;
                        }
                        void showDimension()
                        {
                                cout<<length<<"x"<<width<<"x"<<height;
                        }
                        int getVolume()
                        {
                                return length*width*height;
                        }
};
```

```
int main()
{
	Box b1;
	b1.setLength(13);
	b1.setWidth(14);
	b1.setHeight(15);
	cout<<endl<<"Box Dimension:";
	b1.showDimension();
	cout<<endl<<"Box Volume:"<<b1.getVolume();
}
```

2. Outside Class Definition:

As the name suggests, here the functions are defined outside the class however they are declared inside the class. Functions should be declared inside the class to bound it to the class and indicate it as it's member but they can be defined outside of the class. To define a function outside of a class, scope resolution operator:: is used.

```
#include<iostream>
using namespace std;
class Box
{
	public:
			double length, breadth, height;
			double getVolume();

};
double Box::getVolume()
{
	return length*breadth*height;
}
int main()
{
	Box b1;
	b1.length=5.5;
	b1.breadth=6.5;
	b1.height=7.5;
	cout<<"Box volume:"<<b1.getVolume();
}
```

Access Specifiers

The access modifier of C++ allows us to determine which class members are accessible to other classes and functions, and which are not. There are 3 types of Access Specifiers as

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

Public: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes

and functions 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: The above 2 programs are the example of public access specifier.

Private: The class members declared as *private* can be accessed only by the member 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 are allowed to access the private data members of a class.

Example Program:

```
#include<iostream>
using namespace std;
class Box
{
        private:
                        double length;
                        double breadth;
                        double height;
        public:
                        double findvolume()
                        {
                                return length*breadth*height;
                        }
};
int main()
{
        Box b1;
        b1.length=5.5;
        b1.breadth=7.5;
        b1.height=4.5;
        cout<<"\n Box b1 Volume:"<<b1.findvolume();

}
```

The above program gives error because data member length, breadth and height declare by private access specifier which is access by only its member function. So the correct program as to access private member.

Program

```
#include<iostream>
using namespace std;
class Box
{
        private:
                        double len;
                        double breadth;
                        double height;

        public:
                        void setLength(double l)
                        {
```

```
            len=l;
        }
        void setBreadth(double b)
        {
            breadth=b;
        }
        void setHeight(double h)
        {
            height=h;
        }
        double findvolume()
        {
            return len*breadth*height;
        }
};
int main()
{
    Box b1;
    b1.setLength(5.5);
    b1.setBreadth(7.5);
    b1.setHeight(4.5);
    cout<<"\n Box b1 Volume:"<<b1.findvolume();

}
```

Protected: Protected access modifier is similar to private access modifier in the sense that it can't be accessed outside of its class unless with the help of friend class, the difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

Constructor in C++

C++ constructors are *special* member functions which are created when the object is created or defined and its task is to initialize the object of its class. It is called **constructor** because it constructs the values of data members of the class.

A constructor has the same name as the class and it doesn't have any return type. It is invoked whenever an object of its associated class is created.

Syntax

Types of Constructor

- Default Constructor
- Parameterized Constructor
- Copy Constructor

Default Constructor:

If no constructor is defined in the class then the compiler automatically creates one for the program. This constructor which is created by the compiler when there is no user defined constructor and which doesn't take any parameters is called **default constructor**.

Parameterized constructor

To put it up simply, the constructor that can take arguments are called parameterized constructor. In practical programs, we often need to initialize the various data elements of the different object with different values when they are created. This can be achieved by passing the arguments to the constructor functions when the object is created.

Note: Remember that constructor is always defined and declared in public section of the class and we can't refer to their addresses.

Copy Constructor

Generally in a constructor the object of its own class can't be passed as a value parameter. But the classes own object can be passed as a reference parameter. Such constructor having reference to the object of its own class is known as copy constructor.

Moreover, it creates a new object as a copy of an existing object. For the classes which do not have a copy constructor defined by the user, compiler itself creates a copy constructor for each class known as default copy constructor.

Note: In copy constructor passing an argument by value is not possible.

Program

```
#include<iostream>
using namespace std;
class Box
{
        private: int length, width, height;
        public:
                        //Default Constructor
                        Box()
                        {
                                length=width=height=5;
                        }
                        //parameterised constructor
                        Box(int l, int w, int h)
                        {
                                length=l;
                                width=w;
                                height=h;
                        }
                        //copy constructor
                        Box(Box &b)
                        {
                                length=b.length;
                                width=b.width;
                                height=b.height;
                        }
                        void showDimension()
                        {
                                cout<<length<<"x"<<width<<"x"<<height;
                        }
                        int getVolume()
```

```
                {
                        return length*width*height;
                }

};
int main()
{
        Box b1; //default constructor
        Box b2(5,6,7); // parameterised constructor
        Box b3(b2); // copy constructor

        cout<<endl<<"Box b1:";
        cout<<endl<<"Box b1 Dimesion:";
        b1.showDimension();
        cout<<endl<<"Box Volume"<<b1.getVolume();

        cout<<endl<<"Box b2:";
        cout<<endl<<"Box b2 Dimesion:";
        b2.showDimension();
        cout<<endl<<"Box Volume"<<b2.getVolume();

        cout<<endl<<"Box b3:";
        cout<<endl<<"Box b3 Dimesion:";
        b3.showDimension();
        cout<<endl<<"Box Volume"<<b3.getVolume();
}
```

Destructor in C++:

A destructor is a special member function that works just opposite to constructor, unlike constructors that are used for initializing an object, destructors destroy (or delete) the object.

Syntax of Destructor

```
~class_name()
{
  //Some code
}
```

When does the destructor get called?

A destructor is automatically called when:

1) The program finished execution.
2) When a scope (the { } parenthesis) containing local variable ends.
3) When you call the delete operator.

```
#include<iostream>
using namespace std;
class Box
{
        private: int length, width, height;
        public:
                        //Default Constructor
```

```
        Box()
        {
                length=width=height=5;
        }
        //destructor
        ~Box()
        {
                cout<<endl<<"Object Destroyed";
        }
        void showDimension()
        {
                cout<<length<<"x"<<width<<"x"<<height;
        }
        int getVolume()
        {
                return length*width*height;
        }

};
int main()
{
        Box b1; //default constructor

        cout<<endl<<"Box b1:";
        cout<<endl<<"Box b1 Dimesion:";
        b1.showDimension();
        cout<<endl<<"Box Volume"<<b1.getVolume();
}
```

this pointer

The **this** pointer holds the address of current object, in simple words you can say that this pointer points to the current object of the class. Let's take an example to understand this concept.

Example

Here you can see that we have two data members num and ch. In member function setMyValues() we have two local variables having same name as data members name. In such case if you want to assign the local variable value to the data members then you won't be able to do until unless you use this pointer, because the compiler won't know that you are referring to object's data members unless you use this pointer. This is one of the example where you must use **this** pointer.

Example Program

```
#include <iostream>
using namespace std;
class Demo
{
        private:
                int num;
                char ch;
        public:
                void setMyValues(int num, char ch)
```

```
        {
            this->num = num;
            this->ch = ch;
        }
        void displayMyValues()
        {
             cout<<num<<endl;
            cout<<ch;
        }
};
int main()
{
    Demo d1;
    d1.setMyValues(100, 'A');
    d1.displayMyValues();
}
```

INHERITANCE IN C++

What is Inheritance?

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

Base Class: The class whose features are inherited is called the super class or the base class or the parent class.

Derived Class: The class that inherits the features is called the sub class or the derived class or the child class.

Advantages of Inheritance

1. code reusability and readability
2. The base class once defined and once it is compiled, it need not be reworked.
3. Saves time and effort as the main code need not be written again.

Defining Derived Class:

A derived class can be defined by specifying its relationship with the base class in addition to its own details.

Syntax

```
class derived_class_name: visibility mode base_class_name
{
    -----------
    -----------
    -----------
};
```

The colon (:) indicates that the derived_class_name is derived from the base_class_name. The visibility mode is optional and if present may be either public or private. The default visibility mode

is private. Visibility mode specifies whether the feature of the base class are privately derived or publicly derived.

Access Control in C++

It is not possible for a derived class to directly access private members of its base class. However, it gets a complete object of base class which contains the private members. The derived class can also access private members of the base class through its public and protected members.

A derived class does not inherit

1. Constructors and destructors of the base class
2. Friend classes and friend functions of the base class
3. Overloaded operators of the base class

Modes of Inheritance

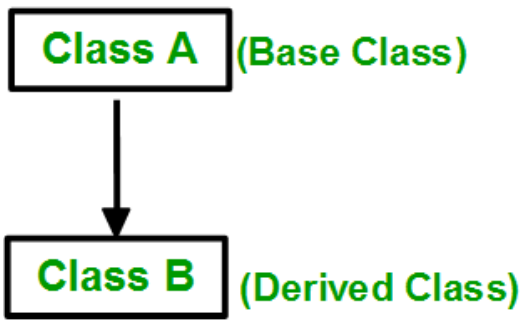
1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base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

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

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

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



Single Inheritance: Public

```
#include<iostream>
using namespace std;
class base
{
        private: int a;
        public: int b;
                        void set_ab(int x, int y)
                        {
                                a=x;
                                b=y;
                        }
                        int get_a()
                        {
                                return a;
                        }
                        void show_a()
                        {
                                cout<<"a="<<a<<endl;
                        }
};
class derived: public base
{
        private:
                        int c;
        public:
                        void mult()
                        {
                                c=b*get_a();
                        }
                        void display()
                        {
                                cout<<"a="<<get_a()<<endl;
                                cout<<"b="<<b<<endl;
                                cout<<"c="<<c<<endl;
                        }
};
int main()
{
        derived d1;
        d1.set_ab(10,20);
        d1.mult();
        d1.show_a();
        d1.display();

        d1.b=5;
```

```
        d1.mult();
        d1.display();
}
```

Single Inheritance: Private

```
#include<iostream>
using namespace std;
class base
{
        private:
                int a;
        public:
                int b;
                void set_ab()
                {
                        cout<<"\n Enter Values a and b:";
                        cin>>a>>b;
                }
                int get_a()
                {
                        return a;
                }
                void show_a()
                {
                        cout<<"a="<<a<<endl;
                }
};
class derived: private base
{
        private:
                int c;
        public:
                void mult()
                {
                        set_ab();
                        c=b*get_a();
                }
                void display()
                {
                        show_a();
                        cout<<"b="<<b<<endl;
                        cout<<"c="<<c<<endl;
                }
};
int main()
{
        derived d1;
        //d1.set_ab(); //Error can not access private member 'a'.
```

```
        d1.mult();
        //d1.show_a();
        d1.display();

        //d1.b=5;
        d1.mult();
        d1.display();
}
```

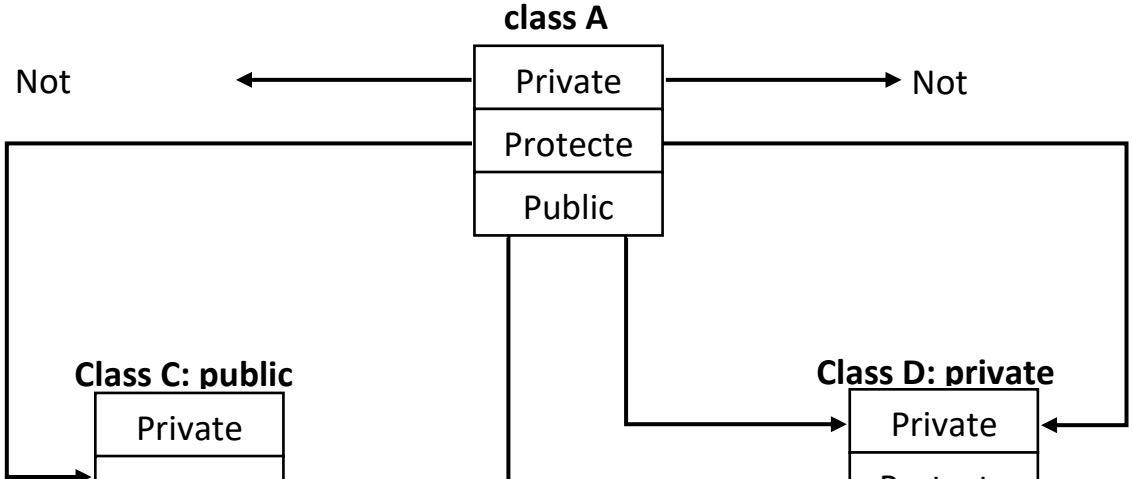
Making a private member inheritable:

We have just seen how to increase the capabilities of an existing class without modifying it. We have also seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. What we do if the private data needs to the private member by making it public. This would make it accessible to all the other function of the program, thus taking away the advantage of data hiding.

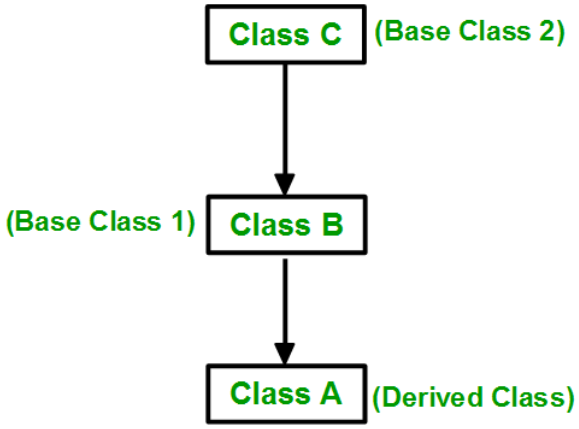
C++ provides a third visibility modifier protected which serve a limited purpose in inheritance. A member declared as protected is accessible by the member function within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes. A class can now use all the three visibility modes as

```
class alpha
{
        private:                //optional
                -------         //only visible to member function within its class
                -------
        Protected:
                -------         //visible to member functions of its own class and derived
class
                -------
        Public:
                -------         //visible to all functions in the program
                -------
}
```

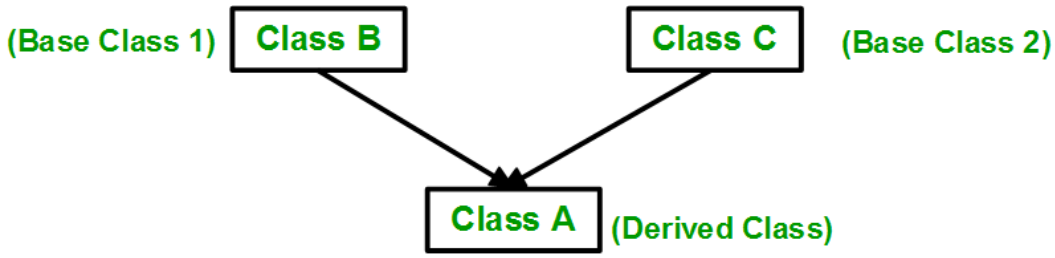
When a protected member is inherited in public mode, it becomes protected in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A protected member, inherited in the private mode derivation, becomes private in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since private member cannot be inherited).



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



3. **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**.



Example Program

```
#include<iostream>
using namespace std;
class base1
{
        protected:
                        int x;
        public:
                        void get_x(int a)
                        {
                                x=a;
                        }
};
class base2
{
        protected:
```

```
                int y;
        public:
                void get_y(int b)
                {
                        y=b;
                }
};
class derived: public base1, public base2
{
        protected:
                int z;
        public:
                void display()
                {
                        z=x+y;
                        cout<<endl<<"X="<<x;
                        cout<<endl<<"Y="<<y;
                        cout<<endl<<"Z="<<z;
                }
};
int main()
{
        derived d1;
        d1.get_x(5);
        d1.get_y(10);
        d1.display();
}
```

Ambiguity Resolution in Inheritance

Sometime, we may face a problem in using multiple inheritance, when a function with the same name appears in more than one base class. Consider the following two classes as

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

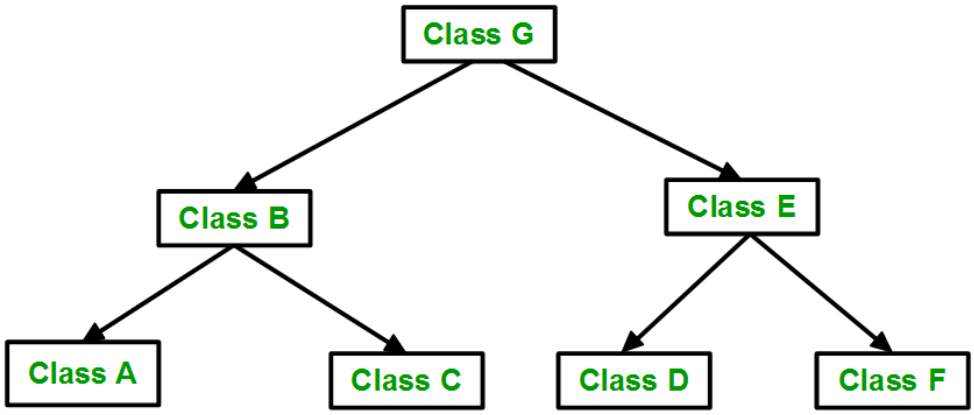
Which display() function used by the derived class when we inherit these two classes?

We can solve this problem by defining a named instance within the derived class, using the class scope resolution operator(::) with the function as

```
#include<iostream>
```

```
using namespace std;
class A
{
        public:
                void display()
                {
                        cout<<endl<<"class A";
                }
};
class B: public A
{
        public:
                void display()
                {
                        cout<<endl<<"class B";
                }
};
int main()
{
        B b;
        b.display();
        b.A::display();
        b.B::display();
}
```

4. **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.



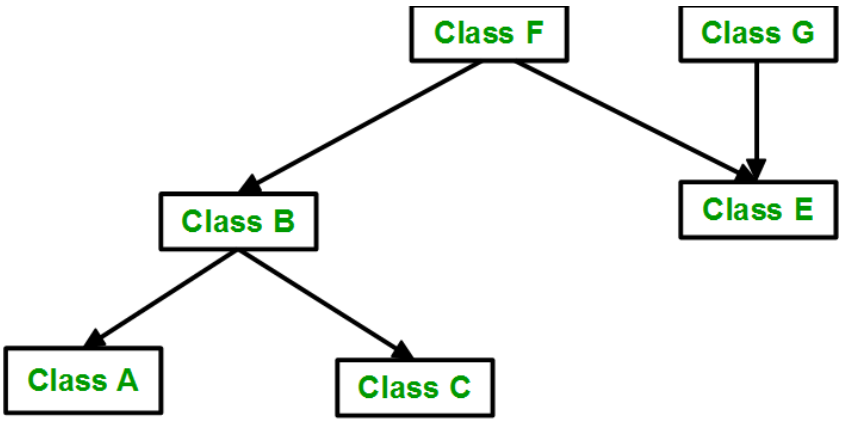
Example Program:

```
#include <iostream>
using namespace std;
class A
{
   public:
                        int x, y;
                        void getdata()
                        {
                           cout<<endl<<"Enter value of x and y:";
                           cin>>x>>y;
                        }
};
```

```
class B : public A
{
   public:
                        void product()
                        {
                           cout<<endl<<"Product= "<<x*y;
                        }
};
class C : public A
{
   public:
                        void sum()
                        {
                      cout <<endl<<"Sum="<<x+y;
                        }
};
int main()
{
   B obj1;
   C obj2;
   obj1.getdata();
   obj1.product();
   obj2.getdata();
   obj2.sum();

}
```

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



Example Program:

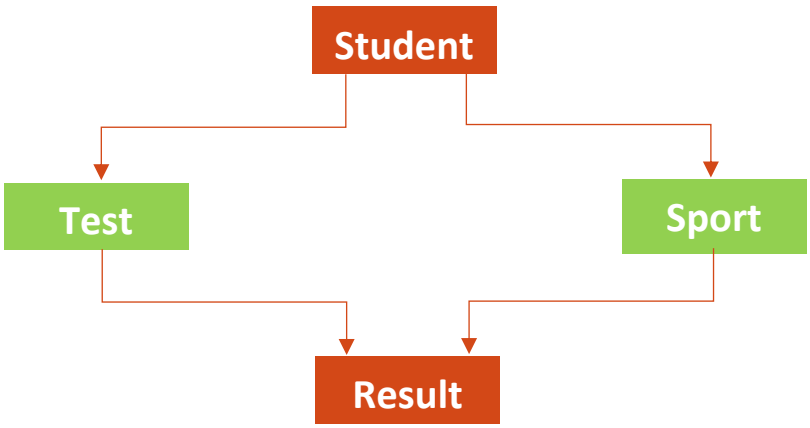
```
#include<iostream>
using namespace std;
class Student
{
        protected:
                        int rno;
                        string sname;
        public:
```

```
                        void getData()
                        {
                                cout<<endl<<"Enter Roll No:";
                                cin>>rno;
                                cout<<endl<<"Enter Student Name:";
                                cin>>sname;
                        }
                        void showData()
                        {
                                cout<<endl<<"Roll No:"<<rno;
                                cout<<endl<<"Student Name:"<<sname;
                        }
};
class Test: public Student
{
        protected:
                        int m1,m2,m3;
        public:
                        void getMarks()
                        {
                                cout<<endl<<"Enter marks of 3 subject:";
                                cin>>m1>>m2>>m3;
                        }
                        void showMarks()
                        {
                                cout<<endl<<"M-I:"<<m1;
                                cout<<endl<<"M-II:"<<m2;
                                cout<<endl<<"M-III:"<<m3;
                        }
};
class Sport
{
        protected:
                        char grade;
        public:
                        void getGrade()
                        {
                                cout<<endl<<"Enter Grade:";
                                cin>>grade;
                        }
                        void showGrade()
                        {
                                cout<<endl<<"Grade:"<<grade;
                        }
};
class Result:public Test, public Sport
{
        private:
                        int total;
```

```
                float ptage;
        public:
                void displayResult()
                {
                        showData();
                        showMarks();
                        showGrade();
                        total=m1+m2+m3;
                        ptage=(float)total/3;
                        cout<<endl<<"Total:"<<total;
                        cout<<endl<<"Percentage:"<<ptage;
                }
};
int main()
{
        Result r1;
        r1.getData();
        r1.getMarks();
        r1.getGrade();
        r1.displayResult();
}
```

Virtual Base Class:

An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class. C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.



Example program:

```
#include<iostream>
using namespace std;
class Student
{
        protected:
                        int rno;
                        string sname;
        public:
                        void getData()
```

```
                        {
                                cout<<endl<<"Enter Roll No:";
                                cin>>rno;
                                cout<<endl<<"Enter Student Name:";
                                cin>>sname;
                        }
                        void showData()
                        {
                                cout<<endl<<"Roll No:"<<rno;
                                cout<<endl<<"Student Name:"<<sname;
                        }
};
class Test:virtual public Student
{
        protected:
                        int m1, m2, m3;
        public:
                        void getMarks()
                        {
                                cout<<endl<<"Enter marks of 3 subject:";
                                cin>>m1>>m2>>m3;
                        }
                        void showMarks()
                        {
                                cout<<endl<<"M-I:"<<m1;
                                cout<<endl<<"M-II:"<<m2;
                                cout<<endl<<"M-III:"<<m3;
                        }
};
class Sport:virtual public Student
{
        protected:
                        char grade;
        public:
                        void getGrade()
                        {
                                cout<<"\n Enter Grade:";
                                cin>>grade;
                        }
                        void showGrade()
                        {
                                cout<<endl<<"Grade:"<<grade;
                        }
};
class Result:public Test, public Sport
{
        private:
                int total;
                float ptage;
```

```
        public:
                        void displayResult()
                        {
                                showData();
                                showMarks();
                                showGrade();
                                total=m1+m2+m3;
                                ptage=(float)total/3.0;
                                cout<<endl<<"Total:"<<total;
                                cout<<endl<<"Percentage:"<<ptage;
                        }
};

int main()
{
        Result r1;
        r1.getData();
        r1.getMarks();
        r1.getGrade();
        r1.displayResult();
}
```

Constructor in Derived Class:

A constructor plays a vital role in initializing an object. An important note, while using constructors during inheritance, is that, as long as a base class constructor does not take any arguments, the derived class need not have a constructor function. However, if a base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructor. Remember, while applying inheritance, we usually create objects using derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base class contains constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base class is constructed in the same order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructor will be executed in the order of inheritance.

The derived class takes the responsibility of supplying the initial values to its base class. The constructor of the derived class receives the entire list of required values as its argument and passes them on to the base constructor in the order in which they are declared in the derived class. A base class constructor is called and executed before executing the statements in the body of the derived class.

The header line of the derived-constructor function contains two parts separated by a colon (:). The first part provides the declaration of the arguments that are passed to the derived class constructor and the second part lists the function calls to the base class.

Example Program:

```
// program to show how constructors are invoked in derived class
#include <iostream>
```

```
using namespace std;
class alpha
{
   private:
                int x;
   public:
            alpha(int i)
            {
               x = i;
               cout<<endl<<"alpha initialized";
            }
            void show_x()
            {
               cout<<endl<<"x="<<x;
            }
};
class beta
{
   private:
                float y;
   public:
            beta(float j)
            {
               y = j;
               cout<<endl<<"beta initialized";
            }
            void show_y()
            {
               cout<<endl<<"y = "<<y;
            }
};
class gamma : public beta, public alpha
{
   private:
                int n,m;
   public:
            gamma(int a, float b, int c, int d):alpha(a), beta(b)
            {
               m = c;
               n = d;
               cout<<endl<<"gamma initialized";
            }
            void show_mn()
            {
               cout<<endl<<"m = "<<m;
               cout<<endl<<"n = "<<n;
            }
};
```

```
int main()
{
   gamma g(5, 7.65, 30, 100);
   g.show_x();
   g.show_y();
   g.show_mn();
}
```

Friend Function:

Friend functions of the class are granted permission to access private and protected members of the class in C++. They are defined globally outside the class' scope. Friend functions are not member functions of the class. So, what exactly is the friend function?

A friend function is a function that is declared outside a class, but is capable of accessing the private and protected members of class. There could be situations in programming wherein we want two classes to share their members. These members may be data members, class functions or function templates. In such cases, we make the desired function, a friend to both these classes which will allow to access private and protected data members of the class.

Generally, non-member functions cannot access the private members of a particular class. Once declared as a friend function, the function is able to access the private and the protected members of these classes.

Syntax

```
class className {
   ... .. ...
   friend returntype functionName(arguments);
   ... .. ...
}
```

Characteristics of Friend Function in C++

1. The function is not in the 'scope' of the class to which it has been declared a friend.
2. Friend functionality is not restricted to only one class
3. Friend functions can be a member of a class or a function that is declared outside the scope of class.
4. It cannot be invoked using the object as it is not in the scope of that class.
5. We can invoke it like any normal function of the class.
6. Friend functions have objects as arguments.
7. It cannot access the member names directly and has to use dot membership operator and use an object name with the member name.
8. We can declare it either in the 'public' or the 'private' part.

Example Program

```
#include<iostream>
using namespace std;
class Sample
{
        private:
```

```
                int a, b;
        public:
                void setValue()
                {
                        a=2;
                        b=5;
                }
                friend float mean(Sample s);
};
float mean(Sample s)
{
        return float(s.a+s.b)/2.0;
}
int main()
{
        Sample x;
        x.setValue();
        cout<<endl<<"Mean Value:"<<mean(x);
}
```

Friend Function with Two Classes:

```
#include<iostream>
using namespace std;
class ABC;
class XYZ
{
        private:
                int x;
        public:
                void setValue(int i)
                {
                        x=i;
                }
                friend void max(XYZ, ABC);
};
class ABC
{
        private:
                int a;
        public:
                void setValue(int i)
                {
                        a=i;
                }
                friend void max(XYZ, ABC);
};
void max(XYZ m, ABC n)
{
```

```
        if(m.x>=n.a)
                cout<<endl<<"Maximum:"<<m.x;
        else
                cout<<endl<<"Maximum:"<<n.a;
}
int main()
{
        ABC abc;
        abc.setValue(7);
        XYZ xyz;
        xyz.setValue(13);
        max(xyz, abc);
}
```

Polymorphism in C++

What is Polymorphism in C++?
In C++, polymorphism causes a member function to behave differently based on the object that calls/invokes it. Polymorphism is a Greek word that means to have many forms. It occurs when you have a hierarchy of classes related through inheritance.

Example

1) The “+” operator in c++ can perform two specific functions at two different scenarios i.e when the “+” operator is used in numbers, it performs addition.
   int a = 6;
   int b = 6;
   int sum = a + b;

   Output: sum =12

   And the same “+” operator is used in the string, it performs concatenation.
   string firstName = "ShreeSoft";
   string lastName = "Informatics";

   string name = firstName + lastName;

   output: name = "ShreeSoft Informatics "

Types of Polymorphism:

C++ supports two types of polymorphism:

1. Compile-time polymorphism
2. Runtime polymorphism.

1) Compile time Polymorphism

This type of polymorphism is also referred to as **static binding** or **early binding**. It takes place during compilation. We use **function overloading** and **operator overloading** to achieve compile-time polymorphism.

1. Function Overloading

Function overloading means one function can perform many tasks. In C++, a single function is used to perform many tasks with the same name and different types of arguments. In the function overloading function will call at the time of program compilation. It is an example of compile-time polymorphism.

Readability of the program increases by function overloading. It is achieved by using the same name for the same action.

2. **Operator Overloading**

Operator overloading means defining additional tasks to operators without changing its actual meaning. We do this by using operator function. The purpose of operator overloading is to provide a special meaning to the user-defined data types. The advantage of Operators overloading is to perform different operations on the same operand.

Example Program

```
#include <iostream>
using namespace std;
class Base
{
   private:
                        string s;
    public:
                Base(){}
                Base(string str)
                {
                        s=str;
                }
                void operator+(Base a)
                {
                        string m = s + a.s;
                        cout<<endl<<"Concatenated String:"<<m;
                }
};
int main()
{
   Base a1("Welcome ");
   Base a2("ShreeSoft Informatics");
   a1+a2;
   return 0;
}
```

2) Run time Polymorphism

In a Runtime polymorphism, functions are called at the time the program execution. Hence, it is known as late binding or dynamic binding.

Function overriding is a part of runtime polymorphism. In function overriding, more than one method has the same name with different types of the parameter list.

It is achieved by using virtual functions and pointers. It provides slow execution as it is known at the run time. Thus, It is more flexible as all the things executed at the run time.

1. **Function Overriding**

In function overriding, we give the new definition to base class function in the derived class. At that time, we can say the base function has been overridden. It can be only

possible in the 'derived class'. In function overriding, we have two definitions of the same function, one in the superclass and one in the derived class. The decision about which function definition requires calling happens at **runtime**. That is the reason we call it 'Runtime polymorphism'.
Example Program

```
#include <iostream>
using namespace std;
class A
{
        public:
                        void disp()
                        {
                        cout<<"Super Class Function"<<endl;
                        }
};
class B: public A
{
        public:
                        void disp()
                        {
                        cout<<"Sub Class Function";
                        }
};
int main()
{
 A obj;
 obj.disp();
 B obj2;
 obj2.disp();
 return 0;
}
```

2. Virtual Function

A virtual function is declared by keyword virtual. The return type of virtual function may be int, float, void.

A virtual function is a member function in the base class. We can redefine it in a derived class. It is part of run time polymorphism. The declaration of the virtual function must be in the base class by using the keyword virtual. A virtual function is not static.

The virtual function helps to tell the compiler to perform dynamic binding or late binding on the function.

If it is necessary to use a single pointer to refer to all the different classes' objects. This is because we will have to create a pointer to the base class that refers to all the derived objects.

But, when the base class pointer contains the derived class address, the object always executes the base class function. For resolving this problem, we use the virtual function.

When we declare a virtual function, the compiler determines which function to invoke at runtime.

Example Program without use of virtual Function

```
#include <iostream>
using namespace std;
class Add
{
  int x=5, y=20;
   public:
   void display()  //overridden function
   {
      cout << "Value of x is : " << x+y<<endl;
   }
};

class Substract: public Add
{
   int y = 10,z=30;
   public:
   void display()  //overridden function
   {
      cout << "Value of y is : " <<y-z<<endl;
   }
};
int main()
{
   Add *m;          //base class pointer .it can only access the base class members
   Substract s;     // making object of derived class
   m = &s;
  m->display();      // Accessing the function by using base class  pointer
   return 0;
}
```

Example Program with use of virtual Function

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

class Add
{
public:
   virtual void print ()
   { int a=20, b=30;
    cout<< " base class Action is:"<<a+b <<endl;
```

```
        }

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

    class Sub: public Add
    {
    public:
      void print () //print () is already virtual function in derived class, we could also declared as virtual void print () explicitly
      {
            int x=20,y=10;
            cout<< " derived class Action:"<<x-y <<endl;
    }
            void show ()
            {
                  cout<< "show derived class" <<endl;
            }
    };

//main function
int main()
{
   Add *aptr;
   Sub s;
   aptr = &s;

   //virtual function, binded at runtime (Runtime polymorphism)
   aptr->print();

   // Non-virtual function, binded at compile time
   aptr->show();

   return 0;
}
```