Experiment No.01

Title : Study of classes and objects

Objectives:

1.Understanding features of Object Oriented Programming.

2.Understanding of Classes & Objects.

Theory:

C++ History:

C++ is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs. C++ runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. This tutorial adopts a simple and practical approach to describe the concepts of C++.

C++ Features:

1. Object :

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

2. Class:

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

3.Abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail

4. Encapsulation

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

5. Inheritance

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

6.Polymorphism

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

7.Overloading

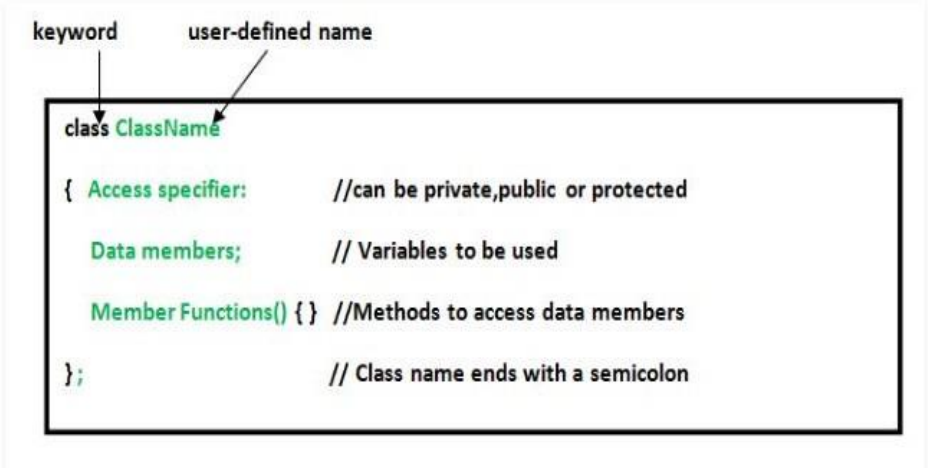
The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

❖ C++ Class:

The building block of C++ that leads to Object Oriented programming is a **Class**. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4*

wheels, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.




```
class Test
{
private:
int data1; float data2;

public:
```

```
void insertIntegerData(int d)
{
data1 = d;
cout << "Number: " << data1;
}

float insertFloatData()
{
cout << "\nEnter data: ";

cin >> data2; return data2;
}
};
```

❖ **C++ objects:**

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

Class Name Object name;

Accessing data members and member functions:

The data members and member functions of class can be accessed using the dot('.') operator with the object.

Case Study:

Write a C++ program for calculating student percentage.

Class Structure:

Conclusion:

Experiment No. 02

Title : Study of constructor and Destructor.

Objectives:

1. Understanding of constructor and Destructor.
2. Utilization constructor and Destructor in classes.

Theory:

❖ **Constructors:**

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after storage is allocated to the object.

```
class A
{
int x;
public:
// constructor
A()
{
      // object initialization
}
```

Constructors are of three types:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

Default Constructors

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Parameterized Constructors

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

```
class Cube
{
public:
int side;
Cube(int x)
{
        side=x;
}
};
```

Copy Constructors

These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object.

❖ **Destructors**

Destructor is a special class function which destroys the object as soon as the scope of object ends. The Destructor is called automatically by the compiler when the object goes out of scope.

The syntax for Destructor is same as that for the constructor, the class name is used for the name of Destructor, with a **tilde** ~ sign as prefix to it.

```
class A
{
public:
// defining destructor for class
~A()
{
        // statement
}
};
```

Case Study:

Write a C++ program for calculating employee yearly salary and Income tax.

Class Structure:

Conclusion:

Experiment No. 03

Title : Study of implementation friend function & Friend class in C++.

Objectives:

1. Study of friend function.
2. Implementation friend function in C++.

Theory:

1.Friend function:

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

Sample Example:

```
class Box
{

  double width;

  public:
    friend void printWidth( Box box );
    void setWidth( double wid );
};

// Member function definition
void Box::setWidth( double wid )
```

```
{
   width = wid;
}

// Note: printWidth() is not a member function of any class.

void printWidth( Box box )
{
   /* Because printWidth() is a friend of Box, it can
   directly access any member of this class */
   cout << "Width of box : " << box.width <<endl;
}
```

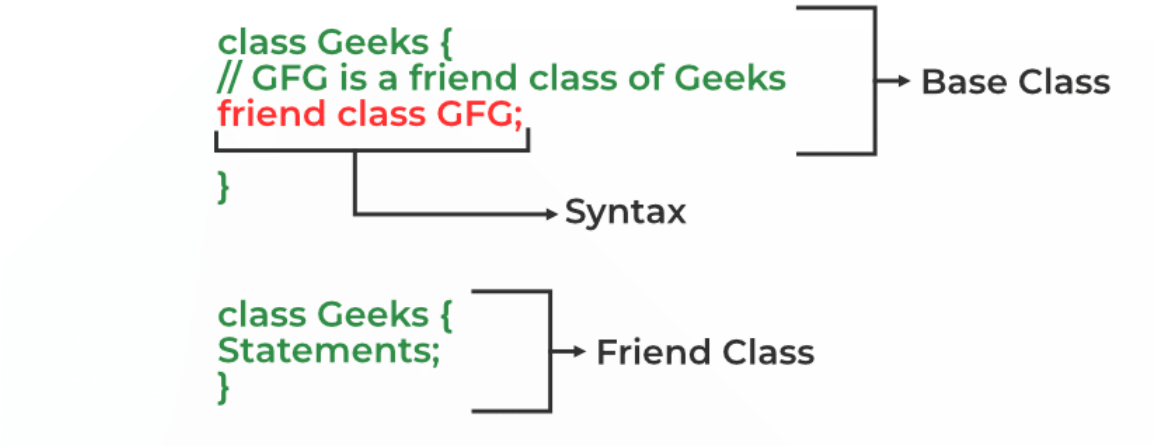
2. Friend Class:

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 Linked List 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



Example

```
#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;
}
```

Output

The value of Private Variable = 10
The value of Protected Variable = 99

Case Study:

Write a C++ program for money transfer in two bank account using friend function.

Class Structure:

Conclusion:

Experiment No. 04

Title : Study of Inline Function, static data member & member functions.

Objectives:

1. Understanding of static keyword.
2. Utilization of static data member.

Theory:

1. Inline Function in C++:

Inline function is one of the important features of C++. C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

```
Inline return-type function-name (parameters)
{
  // function code
}
```

Example:

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
  return s*s*s;
}
int main()
{
  cout << "The cube of 3 is: " << cube(3) << "\n";
  return 0;
}
```

Output: The cube of 3 is: 27

2. Static Keyword:

Static data members are class members that are declared using static keywords. A static member has certain special characteristics. These are:

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is initialized before any object of this class is being created, even before main starts.
- It is visible only within the class, but its lifetime is the entire program

Syntax
Static data type data_member_name;

Example:

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

class A
{
public:
   A() { cout << "A's Constructor Called " << endl;  }
};

class B
{
   static A a;
public:
   B() { cout << "B's Constructor Called " << endl; }
};

int main()
{
   B b;
   return 0;
}
```

Output:
B's Constructor Called

The above program calls only B's constructor, it doesn't call A's constructor. The reason for this is simple, static members are only declared in a class declaration, not defined. They must be explicitly defined outside the class using the scope resolution operator.

3. Member Functions:
There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution **::** operator along with class name and function name.

Example:

```
#include <bits/stdc++.h>
using namespace std;
class Geeks
{
   public:
   string geekname;
   int id;
    // printname is not defined inside class definition
   void printname();
```

```
    // printid is defined inside class definition
   void printid()
   {
      cout <<"Geek id is: "<<id;
   }
};
 // Definition of printname using scope resolution operator ::
void Geeks::printname()
{
   cout <<"Geekname is: "<<geekname;
}
int main()
{
   Geeks obj1;
   obj1.geekname = "xyz";
   obj1.id=15;
    // call printname()
   obj1.printname();
   cout << endl;
     // call printid()
   obj1.printid();
   return 0;
}
```

Output:

Geekname is: xyz

Geek id is: 15

Case Study:

Write a C++ program for calculating total vehicle passed and collected amount individual as well as over all summary data.

Class Structure:

Conclusion:

Experiment No. 05

Title : Study of Array, Array of Objects, Pointer to Object, THIS pointer, Dynamic allocation operators (New & Delete) in C++

Objectives:

1. Understanding of array ,array of objects, this pointer.
2. Implementation of array of objects , new & delete operator.

Theory:

❖ Array

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [arraySize];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement −

double balance[10];

Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

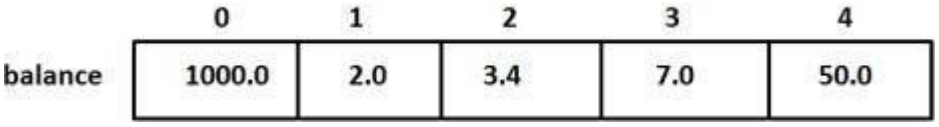
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above −



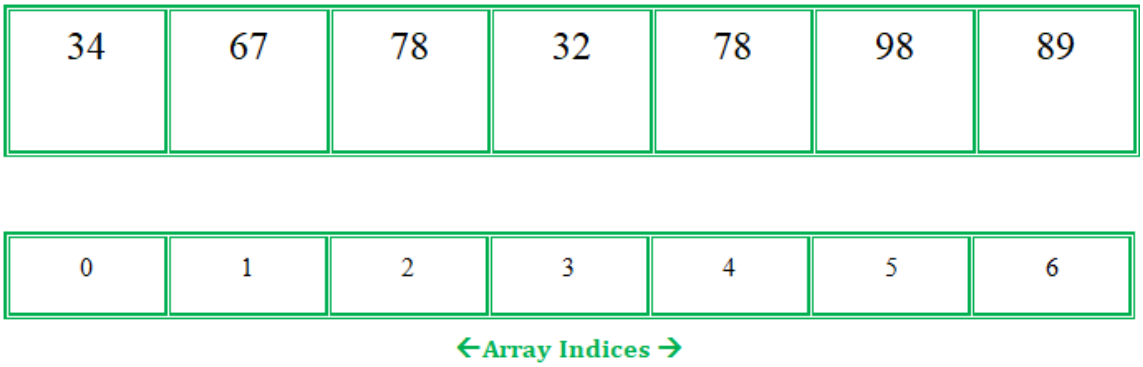
–

Sr.No	Concept & Description
1	Multi-dimensional arrays C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	Pointer to an array You can generate a pointer to the first element of an array by simply specifying the array name, without any index.
3	Passing arrays to functions You can pass to the function a pointer to an array by specifying the array's name without an index.
4	Return array from functions C++ allows a function to return an array.

❖ **Array of objects:**

An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They can be used to store the collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C/C++ can store derived data types such as structures, pointers, etc.

Given below is the picture representation of an array.



When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:
ClassName ObjectName[number of objects];

The Array of Objects stores *objects*. An array of a class type is also known as an array of objects.

❖ Using Pointers with Objects

For accessing normal data members we use the dot . operator with object and -> qith pointer to object. But when we have a pointer to data member, we have to dereference that pointer to get what its pointing to, hence it becomes,

```
Object.*pointerToMember
```

Copy
and with pointer to object, it can be accessed by writing,

```
ObjectPointer->*pointerToMember
```

Copy
Lets take an example, to understand the complete concept.

```
class Data

{

   public:

   int a;

   void print()

{
```

```
        cout <<"a is "<< a;

}

};



int main()

{

    Data d,*dp;

ect.*pointerToMember
```

Copy
and with pointer to object, it can be accessed by writing,

```
ObjectPointer->*pointerToMember
```

Copy
Lets take an example, to understand the complete concept.

```
class Data

{

    public:

    int a;

    void print()

{

        cout <<"a is "<< a;

}

};
```

```
int main()

{

   Data d,*dp;

   dp =&d;// pointer to object



   int Data::*ptr=&Data::a;// pointer to data member 'a'



   d.*ptr=10;

   d.print();



   dp->*ptr=20;

   dp->print();

}
```

Copy

```
a is 10

a is 20
```

The syntax is very tough, hence they are only used under special circumstances.

❖ **"THIS" pointer in C++ :**

To understand 'this' pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**

- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

❖ **Dynamic allocation operators (New & Delete)**

What is new?

The new is a memory allocation operator, which is used to allocate the memory at the runtime. The memory initialized by the new operator is allocated in a heap. It returns the starting address of the memory, which gets assigned to the variable. The functionality of the new operator in C++ is similar to the malloc() function, which was used in the C programming language. C++ is compatible with the malloc() function also, but the new operator is mostly used because of its advantages.

Syntax of new operator

1. type variable = **new** type(parameter_list);

In the above syntax

type: It defines the datatype of the variable for which the memory is allocated by the new operator.

variable: It is the name of the variable that points to the memory.

parameter_list: It is the list of values that are initialized to a variable.

The new operator does not use the sizeof() operator to allocate the memory. It also does not use the resize as the new operator allocates sufficient memory for an object. It is a construct that calls the constructor at the time of declaration to initialize an object. As we know that the new operator allocates the memory in a heap; if the memory is not available in a heap and the new operator tries to allocate the memory, then the exception is thrown. If our code is not able to handle the exception, then the program will be terminated abnormally.

Delete operator

It is an operator used in C++ programming language, and it is used to de-allocate the memory dynamically. This operator is mainly used either for those pointers which are allocated using a new operator or NULL pointer.

Syntax

1. **delete** pointer_name

For example, if we allocate the memory to the pointer using the new operator, and now we want to delete it. To delete the pointer, we use the following statement:

1. **delete** p;

To delete the array, we use the statement as given below:

1. **delete** [] p;

Some important points related to delete operator are:

- It is either used to delete the array or non-array objects which are allocated by using the new keyword.
- To delete the array or non-array object, we use delete[] and delete operator, respectively.
- The new keyword allocated the memory in a heap; therefore, we can say that the delete operator always de-allocates the memory from the heap
- It does not destroy the pointer, but the value or the memory block, which is pointed by the pointer is destroyed.

Case Study:

Write a C++ program for implementing array.

Class Structure:

Conclusion:

Experiment No. 06

Title : Study of implementation Function & Operator overloading in C++.

Objectives:

1. Study of operator & function overloading.
2. Implementation function & operator overloading in C++.

Theory:

Types of overloading in C++ are:

- Function overloading
- Operator overloading

1. Function Overloading in C++

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 feature in C++.

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)
- Parameters should have a different number

  add(int a, int b)

  add(int a, int b, int c)

2.Operator overloading:

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate

definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Overload able /Non-overload able Operators:

Following is the list of operators, which can be overloaded –

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded –

::	.*	.	?:

```
class Box {
  private:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box

  public:
     // Overload + operator to add two Box objects.
     Box operator+(const Box& b) {
        Box box;
        box.length = this->length + b.length;
        box.breadth = this->breadth + b.breadth;
        box.height = this->height + b.height;
        return box;
     }
};
```

Case Study:

Write a C++ program for addition and subtraction of date using function & operator overloading.

Class Structure:

Conclusion:

Experiment No. 07

Title : Study of Inheritance.

Objectives:

Understanding of Inheritance and its types.
Implementation inheritance in program.

Theory:

Inheritance:

Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

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

Base and Derived Classes:

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

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

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

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

```
        void setHeight(int h) {
                height = h;
        }
};


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

Access Control and Inheritance:

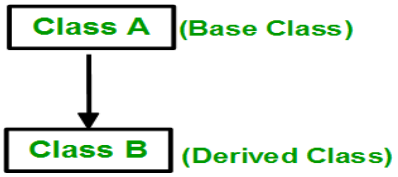
The different access types according to - who can access them in the following way −

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

❖ **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.

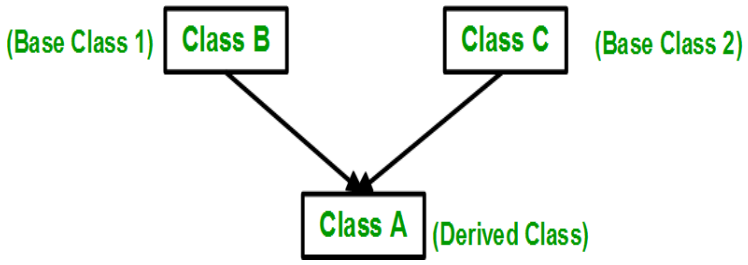
Example:




```
// base class

class Vehicle {
        public:
                Vehicle()
                {
```

```
        }
};

// sub class derived from two base classes class
Car: public Vehicle
{

};
```

2.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:

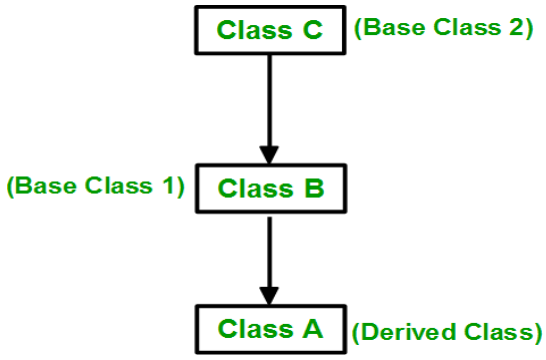
```
class subclass_name : access_mode base_class1,
access_mode base_class2, ....
{
  //body of subclass
};
```

Example:

```
// 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
{

};
```

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

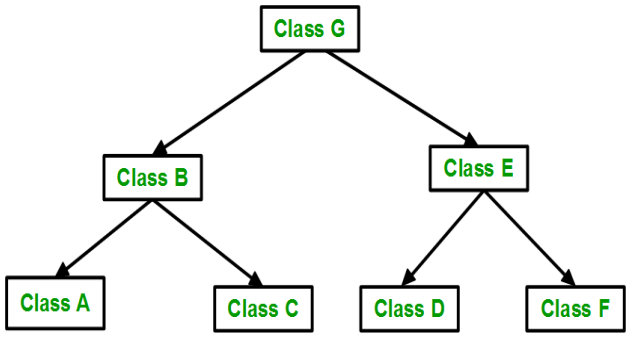


Example:

```
// base class
class Vehicle
{
        public:
                Vehicle()
                {
                        cout << "This is a Vehicle" <<
endl;
                }
};
```

```
class fourWheeler: public Vehicle
{
        public:
                four Wheeler()
                        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;
                }
};
```

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.

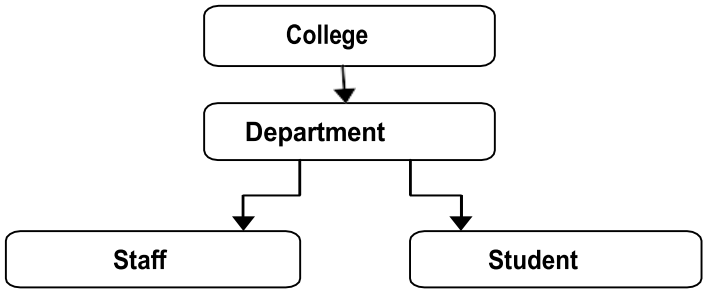


Example:

```
// base class class Vehicle
                    {
                        public:
                            Vehicle()
                            {
                                cout << "This is a Vehicle" << endl;}
                };// second sub
class Bus: public Vehicle
                    {

                    };
```

Case Study:

Write a C++ program for following hierarchy.



Class Structure:

Conclusion:

Experiment No.08

Title : Study of implementation virtual function & virtual class,early & late binding

Objectives:

1.Understanding of virtual function & virtual class

2.Study of early & late binding.

Theory:

❖ Virtual Function

A virtual function is a member function which is declared within a base class and is re-defined (overridden) 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.

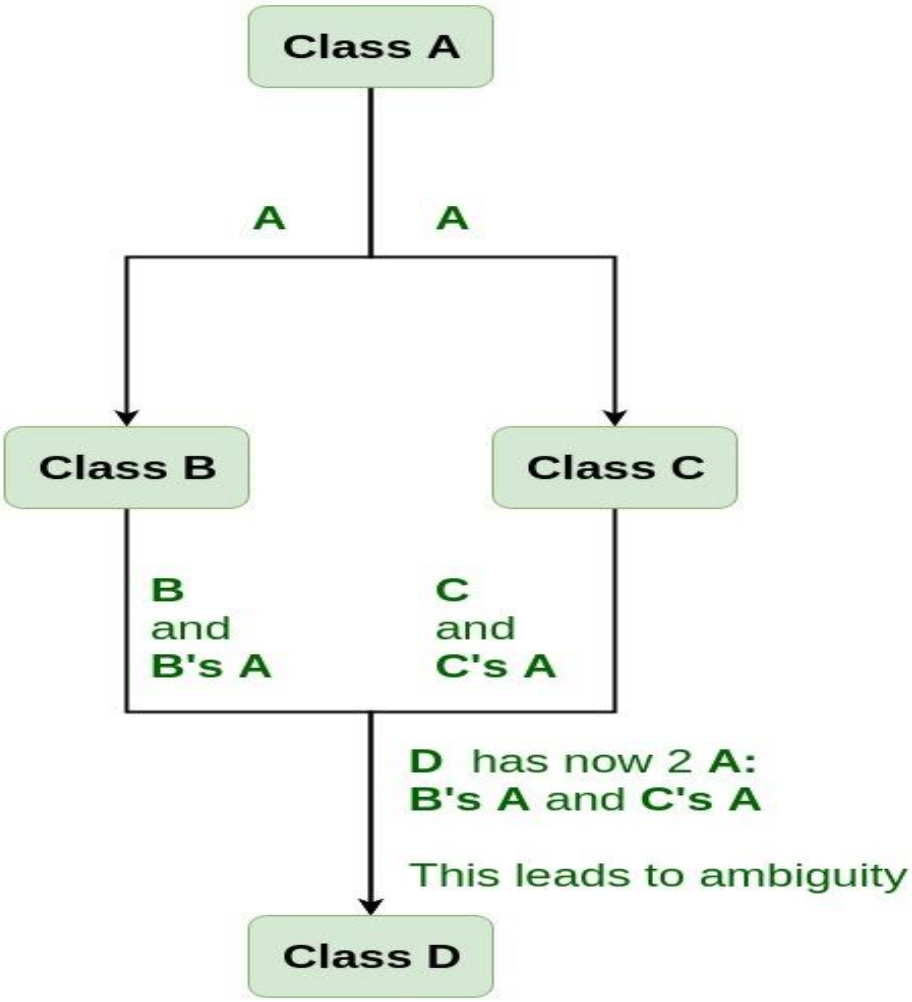
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function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

❖ Virtual 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:

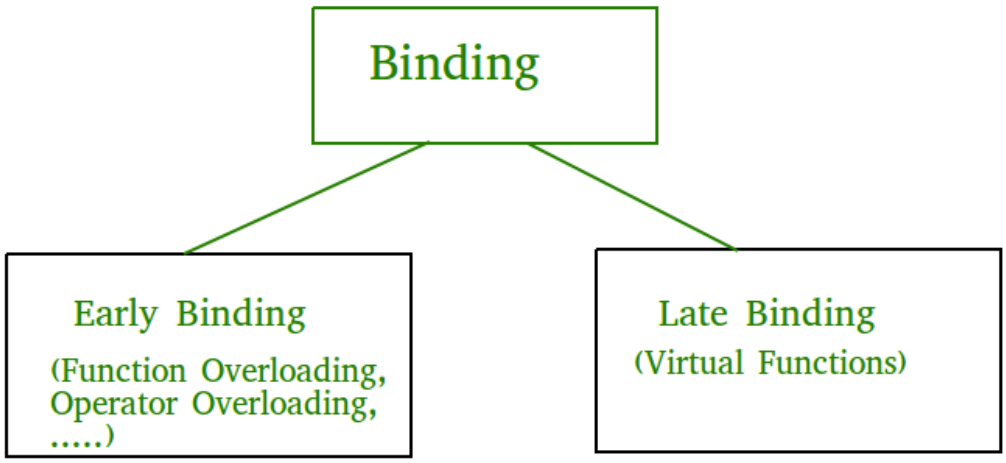
Consider the situation where we have one class A . This class A is inherited by two other classes B and C. Both these class are inherited into another in a new class D as shown in figure below.



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.

❖ Early Binding & Late Binding:

Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



Early Binding (compile-time time polymorphism) As the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.

Late Binding : (Run time polymorphism) In this, the compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition.This can be achieved by declaring a virtual function.

Case Study:

Write a C++ program using Virtual Function.

Class Structure:

Conclusion:

Experiment No. 09

Title : Study of implementation of Generic Function & Classes

Objectives:

1.Understanding of Generic Function & Classes.
2.Implementation of Generic Function & Classes in C++.

Theory :

❖ **Generic Functions:**

Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces. For example, classes like an array, map, etc, which can be used using generics very efficiently. We can use them for any type.

The method of Generic Programming is implemented to increase the efficiency of the code. Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates the need to create different algorithms if the data type is an integer, string or a character.

The advantages of Generic Programming are

1.Code Re usability

2.Avoid Function Overloading

3.Once written it can be used for multiple times and cases.

Generics can be implemented in C++ using Templates. Template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

Generic Functions using Template:

We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().

Class Template

Like function templates, we can also use templates with the class to make it compatible with more than one data type. In C++ programming, you might have used the vector to create a dynamic array, and you can notice that it works fine with every data type you pass inside the <>, ex- vector<int>. This is just because of the class template. It can be useful for classes like Linked List, binary tree, Stack, Queue, Array, etc.

Syntax:

```
template <class T>
class class Name {
 // Class Definition.
 // We can use T as a type inside this class.
};
```

Case Study:

Write a C++ program for Generic Function & Classes.

Class Structure:

Conclusion:

Experiment No. 10

Title : Study of implementation of STL in C++

Objectives:

1.Understanding of STL in C++.

2.Implementation of STL in C++.

Theory :

❖ **The Standard Template Library (STL):**

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. Working knowledge of template classes is a prerequisite for working with STL.

The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs. The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting, and manipulating data.

One of the key benefits of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This means that you can write an algorithm once, and then use it with different types of data without having to write separate code for each type.

The STL also provides a way to write efficient code. Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.

❖ **Some of the key components of the STL include:**

1.Containers: The STL provides a range of containers, such as vector, list, map, set, and stack, which can be used to store and manipulate data.

2.Algorithms: The STL provides a range of algorithms, such as sort, find, and binary_search, which can be used to manipulate data stored in containers.

3.Iterators: Iterators are objects that provide a way to traverse the elements of a container. The STL provides a range of iterators, such as forward_iterator, bidirectional_iterator, and random_access_iterator, that can be used with different types of containers.

4.Function Objects: Function objects, also known as functors, are objects that can be used as function arguments to algorithms. They provide a way to pass a function to an algorithm, allowing you to customize its behavior.

5.Adapters: Adapters are components that modify the behavior of other components in the STL. For example, the reverse_iterator adapter can be used to reverse the order of elements in a container.

By using the STL, you can simplify your code, reduce the likelihood of errors, and improve the performance of your programs.

❖ **STL has 4 components:**

1.Algorithms
2.Containers
3.Functions
4.Iterators

1. Algorithms

The header algorithm defines a collection of functions specially designed to be used on a range of elements. They act on containers and provide means for various operations for the contents of the

containers.

2. Containers

Containers or container classes store objects and data. There are in total seven standards “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adapters.

3. Functions

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functions. Functions allow the working of the associated function to be customized with the help of parameters to be passed. Must Read – Functions

4. Iterators

As the name suggests, iterators are used for working on a sequence of values. They are the major feature that allows generality in STL. Must Read – Iterators

- **Advantages of the C++ Standard Template Library (STL):**

Re usability: One of the key advantages of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This can lead to more efficient and maintainable code.
Efficient algorithms: Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.
Improved code readability: The STL provides a consistent and well-documented way of working with data, which can make your code easier to understand and maintain.
Large community of users: The STL is widely used, which means that there is a large community of developers who can provide support and resources, such as tutorials and forums.

- **Disadvantages of the C++ Standard Template Library (STL**):

Learning curve: The STL can be difficult to learn, especially for beginners, due to its complex syntax and use of advanced features like iterators and function objects.
Lack of control: When using the STL, you have to rely on the implementation provided by the library, which can limit your control over certain aspects of your code.
Performance: In some cases, using the STL can result in slower execution times compared to custom code, especially when dealing with small amounts of data.

Case Study:

Write a C++ program using STL in C++

Class Structure:

Conclusion:

Experiment No. 11

Title : Study of implementation Exception Handling C++.

Objectives:

1. Study of Exception Handling.
2. Implementation of Exception Handling in C++.

Theory:

❖ **Exception Handling:**

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch** and **throw**.

- **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows –

```
try
{
  // protected code
}

catch( ExceptionName e1 )
{
  // catch block
```

```
}

catch( ExceptionName e2 )
{
  // catch block
}

catch( ExceptionName eN )
{
  // catch block
}
```

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

Throwing Exceptions:

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs –

```
double division(int a, int b)
{
     if( b == 0 )
     {
                throw "Division by zero condition!";
     }
  return (a/b);
}
```

Catching Exceptions:

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
  // protected code
```

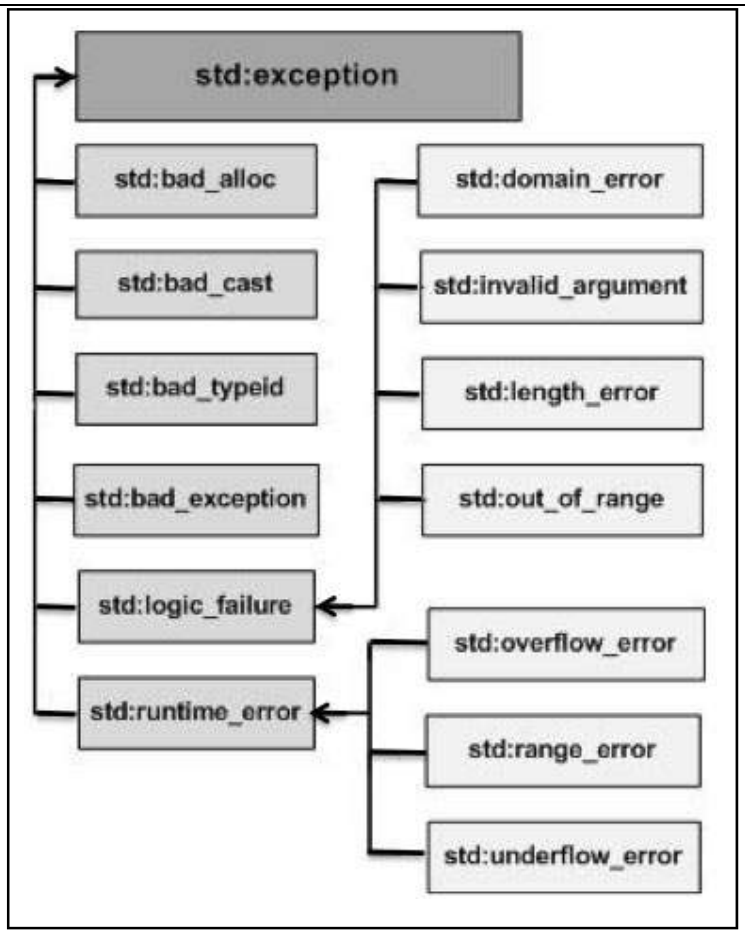
```
}
catch( ExceptionName e )
{
  // code to handle ExceptionName exception
}
```

Sample Example:

```
double division(int a, int b)
{
    if( b == 0 )
    {
            throw "Division by zero condition!";
    }
  return (a/b);
}


int main ()
{
     int x = 50;
    int y = 0;
    double z = 0;

    try
    {
            z = division(x, y);
            cout << z << endl;
    }
    catch (const char* msg)
    {
            cout << msg << endl;
    }

  return 0;
}
```

C++ Standard Exceptions:

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below −



Here is the small description of each exception mentioned in the above hierarchy −

Sr.No	Exception & Description
1	**std::exception** An exception and parent class of all the standard C++ exceptions.
2	**std::bad_alloc** This can be thrown by **new**.
3	**std::bad_cast** This can be thrown by **dynamic_cast**.
4	**std::bad_exception** This is useful device to handle unexpected exceptions in a C++ program.

5	**std::bad_typeid** This can be thrown by **typeid**.
6	**std::logic_error** An exception that theoretically can be detected by reading the code.
7	**std::domain_error** This is an exception thrown when a mathematically invalid domain is used.
8	**std::invalid_argument** This is thrown due to invalid arguments.
9	**std::length_error** This is thrown when a too big std::string is created.
10	**std::out_of_range** This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().
11	**std::runtime_error** An exception that theoretically cannot be detected by reading the code.
12	**std::overflow_error** This is thrown if a mathematical overflow occurs.
13	**std::range_error** This is occurred when you try to store a value which is out of range.
14	**std::underflow_error** This is thrown if a mathematical underflow occurs.

Case Study:

Write a C++ program for implementing new my_exception class for array related exception.

Class Structure:

Conclusion:

Experiment No. 12

Title : Study of implementation of File Handling.

Objectives:

1.Understanding of File Handling.
2.Implementation of File Handling in C++.

Theory :

❖ File Handling

File handling is used to store data permanently in a computer. Using file handling we can store our data in secondary memory (Hard disk).
How to achieve the File Handling
For achieving file handling we need to follow the following steps:-
STEP 1-Naming a file
STEP 2-Opening a file
STEP 3-Writing data into the file
STEP 4-Reading data from the file
STEP 5-Closing a file.

Streams in C++ :-

We give input to the executing program and the execution program gives back the output. The sequence of bytes given as input to the executing program and the sequence of bytes that comes as output from the executing program are called stream. In other words, streams are nothing but the flow of data in a sequence.

The input and output operation between the executing program and the devices like keyboard and monitor are known as "console I/O operation". The input and output operation between the executing program and files are known as "disk I/O operation".

Classes for File stream operations :-

The I/O system of C++ contains a set of classes which define the file handling methods. These include ifstream, ofstream and fstream classes. These classes are derived from fstream and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files.

1. ios:-
ios stands for input output stream.
This class is the base class for other classes in this class hierarchy.
This class contains the necessary facilities that are used by all the other derived classes for input and output operations.

2. istream:-
istream stands for input stream.
This class is derived from the class 'ios'.
This class handle input stream.
The extraction operator(>>) is overloaded in this class to handle input streams from

files to the program execution.
This class declares input functions such as get(), getline() and read().

3. ostream:-
ostream stands for output stream.
This class is derived from the class 'ios'.
This class handle output stream.
The insertion operator(<<) is overloaded in this class to handle output streams to files from the program execution.
This class declares output functions such as put() and write().

4. streambuf:-
This class contains a pointer which points to the buffer which is used to manage the input and output streams.

5. fstreambase:-
This class provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream class.
This class contains open() and close() function.

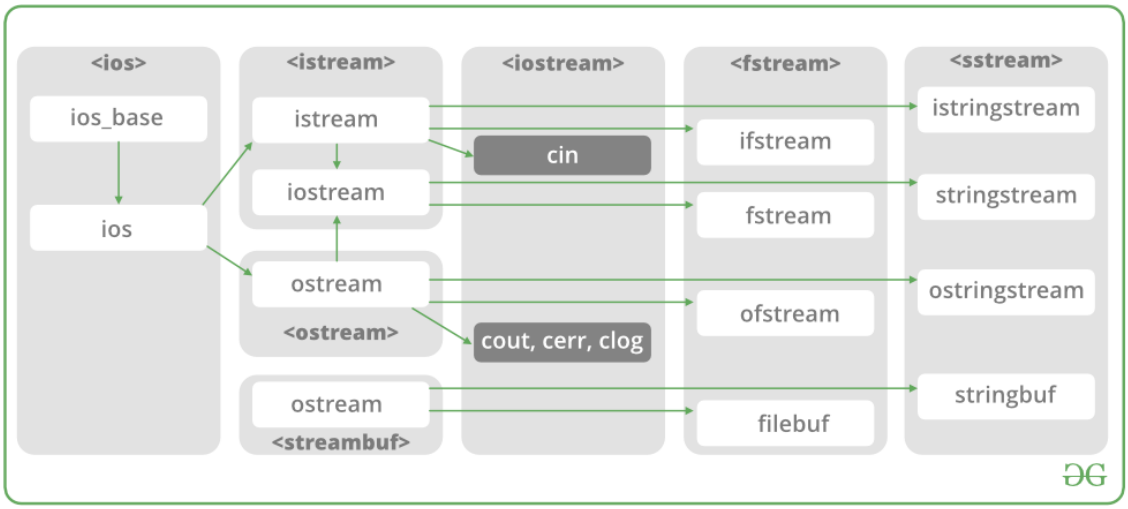
6. ifstream:-
This class provides input operations.
It contains open() function with default input mode.
Inherits the functions get(), getline(), read(), seekg() and tellg() functions from the istream.

7. ofstream:-
This class provides output operations.
It contains open() function with default output mode.
Inherits the functions put(), write(), seekp() and tellp() functions from the ostream.

8. fstream:-
This class provides support for simultaneous input and output operations.
Inherits all the functions from istream and ostream classes through iostream.

9. filebuf:-
Its purpose is to set the file buffers to read and write.
We can also use file buffer member function to determine the length of the file.

In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream available in fstream headerfile.
ofstream: Stream class to write on files
ifstream: Stream class to read from files
fstream: Stream class to both read and write from/to files.



Now the first step to open the particular file for read or write operation. We can open file by
1. passing file name in constructor at the time of object creation
2. using the open method

For e.g.

Open File by using constructor
ifstream (const char* filename, ios_base::openmode mode = ios_base::in);
ifstream fin(filename, openmode) by default openmode = ios::in
ifstream fin("filename");

Open File by using open method
Calling of default constructor
ifstream fin;
fin.open(filename, openmode)
fin.open("filename");

Modes

Member Constant	Stands For	Access
in *	input	File open for reading: the internal stream buffer supports input operations.
out	output	File open for writing: the internal stream buffer supports output operations.

binary	binary	Operations are performed in binary mode rather than text.
ate	at end	The output position starts at the end of the file.
app	append	All output operations happen at the end of the file, appending to its existing contents.
trunc	truncate	Any contents that existed in the file before it is open are discarded.

Case Study:

Write a C++ program for File handling.

Class Structure:

Conclusion: