Introduction to Object Oriented Paradigm, Procedural Paradigm

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. For example, a person is an object which has certain properties such as height, gender, age, etc. It also has certain methods such as move, talk, and so on. Detailed features have been discussed in the later part of the lecture.

Difference between Procedural Programming and Object Oriented Programming

Procedural Programming	**Object Oriented Programming**
In procedural programming, program is divided into small parts called ***functions***.	In object oriented programming, program is divided into small parts called ***objects***.
Procedural programming follows ***top down approach***.	Object oriented programming follows ***bottom up approach***.
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private, public, protected etc.
Adding new data and function is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way for hiding data so it is ***less secure***.	Object oriented programming provides data hiding so it is ***more secure***.
In procedural programming, overloading is not possible.	Overloading is possible in object oriented programming.
In procedural programming, function is more important than data.	In object oriented programming, data is more important than function.
Procedural programming is based on ***unreal world***.	Object oriented programming is based on ***real world***.
Examples: C, FORTRAN, Pascal, Basic etc.	Examples: C++, Java, Python, C# etc.

The important features of Object Oriented programming are:

- Inheritance
- Polymorphism
- Data Hiding
- Encapsulation
- Overloading
- Reusability

Let us see a brief overview of these important features of Object Oriented programming

But before that it is important to know some new terminologies used in Object Oriented programming namely

- Objects
- Classes

Objects:

Object is an instance of a class.

Classes:

These contain data and functions bundled together under a unit. In other words class is a collection of similar objects. When we define a class it just creates template or Skelton. So no memory is created when class is created. Memory is occupied only by object.

Example:

```
Class classname
 {
 Data
 Functions
 };
 main ( )
 {
 classname objectname1,objectname2,..;
 }
```

In other words classes acts as data types for objects.

Member functions:

The functions defined inside the class as above are called member functions.

Data Hiding:

This concept is the main heart of an Object oriented programming. The data is hidden inside the class by declaring it as private inside the class. When data or functions are defined as private it can be accessed only by the class in which it is defined. When data or functions are defined as public then it can be accessed anywhere outside the class. Object Oriented programming gives importance to protecting data which in any system. This is done by declaring data as private and

making it accessible only to the class in which it is defined. This concept is called data hiding. But one can keep member functions as public.

So above class structure becomes

Example:

```
Class classname
 {
 private:
 datatype data;

 public:
 Member functions
 };
 main ( )
 {
 classname objectname1,objectname2,..;
 }
```

Encapsulation:

The technical term for combining data and functions together as a bundle is encapsulation.

Inheritance:

Inheritance as the name suggests is the concept of inheriting or deriving properties of an existing class to get new class or classes. In other words we may have common features or characteristics that may be needed by number of classes. So those features can be placed in a common tree class called base class and the other classes which have these charaterisics can take the tree class and define only the new things that they have on their own in their classes. These classes are called derived class. The main advantage of using this concept of inheritance in Object oriented programming is it helps in reducing the code size since the common characteristic is placed separately called as base class and it is just referred in the derived class. This provide the users the important usage of terminology called as reusability

Reusability:

This usage is achieved by the above explained terminology called as inheritance. Reusability is nothing but re- usage of structure without changing the existing one but adding new features or characteristics to it. It is very much needed for any programmers in different situations. Reusability gives the following advantages to users

It helps in reducing the code size since classes can be just derived from existing one and one need to add only the new features and it helps users to save their time.

For instance if there is a class defined to draw different graphical figures say there is a user who want to draw graphical figure and also add the features of adding color to the graphical figure. In this scenario instead of defining a class to draw a graphical figure and coloring it what the user can do is make use of the existing class for drawing graphical figure by deriving the class and add new feature to the derived class namely add the feature of adding colors to the graphical figure.

Polymorphism and Overloading:

Poly refers many. So Polymorphism as the name suggests is a certain item appearing in different forms or ways. That is making a function or operator to act in different forms depending on the place they are present is called Polymorphism.

The structure of C++ program is divided into four different sections:

(1) Header File Section
(2) Class Declaration section
(3) Member Function definition section
(4) Main function section

(1) Header File Section:

This section contains various header files.
You can include various header files in to your program using this section.

For example:
include <iostream.h >

Header file contains declaration and definition of various built in functions as well as object. In order to use this built in functions or object we need to include particular header file in our program.

(2) Class Declaration Section:

This section contains declaration of class.
You can declare class and then declare data members and member functions inside that class.

For example:
class Demo
{

```
int a, b;
public:
void input();
void output();
}
```

You can also inherit one class from another existing class in this section.

(3) Member Function Definition Section:

o This section is optional in the structure of C++ program.
o Because you can define member functions inside the class or outside the class. If all the member functions are defined inside the class then there is no need of this section.
o This section is used only when you want to define member function outside the class.
o This section contains definition of the member functions that are declared inside the class.

For example:

```
void Demo:: input ()
{
cout << "Enter Value of A:";
cin >> a;
cout << "Enter Value of B:";
cin >> b;
}
```

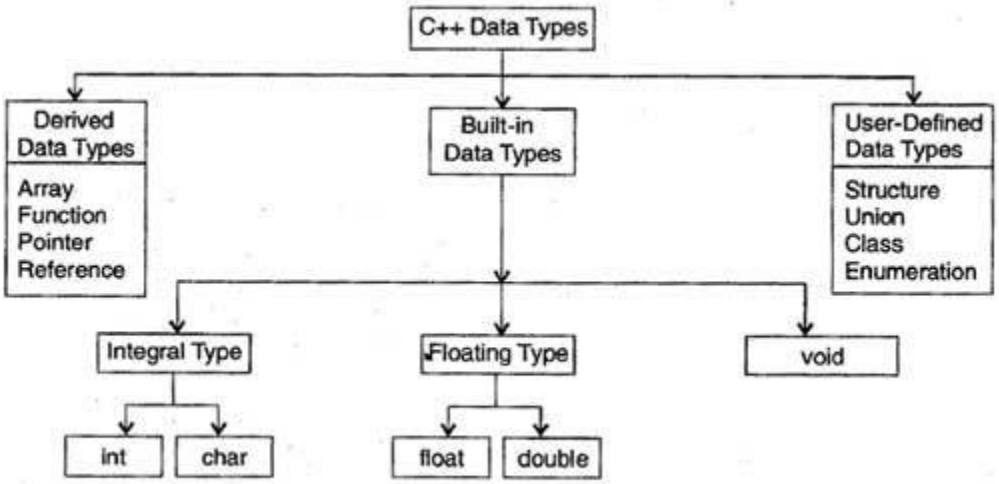
(4) Main Function Section:

In this section you can create an object of the class and then using this object you can call various functions defined inside the class as per your requirement.

For example:

```
Void main ()
{
Demo d1;
d1.input ();
d1.output ();
}
```

A data type determines the type and the operations that can be performed on the data. C++ provides various data types and each data type is represented differently within the computer's memory. The various data types provided by C++ are *built-in data types*, *derived data types* and *user-defined data types* as shown in Figure.



Various Data Types in C++

What is variable declaration and variable initialization?

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of linking of the program.

Initializing a variable means assigning an initial value to that variable.

Some valid declarations are shown here –

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

Here The line **int i, j, k;** both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression.
Some examples are −

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
```

```
char x = 'x';           // the variable x has the value 'x'.
```

extern keyword is used to declare a variable at any place.

Expressions

There are two kinds of expressions in C++ −

lvalue	Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.
rvalue	The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

For example

```
int a = 20;         // a is lvalue and 20 is rvalue
```

But the following is not a valid statement and would generate compile-time error –

```
10 = 20;
```

Operators in C++

An operator is a symbol that tells the compiler to perform specific **mathematical** or **logical** manipulations. C++ is rich in built-in operators and provide the following types of operators

1	Arithmetic Operators
2	Relational Operators
3	Logical Operators
4	Bitwise Operators
5	Assignment Operators
6	Misc Operators

1. Arithmetic Operators

Assume variable A holds 10 and variable B holds 20, then −

OPERATOR	USAGE	EXAMPLE

+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

2. Relational Operators

Assume variable A holds 10 and variable B holds 20, then –

OPERATOR	USAGE	EXAMPLE
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

3. Logical Operators

Assume variable A holds 1 and variable B holds 0, then –

OPERATOR	USAGE	EXAMPLE
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.

||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

4. Bitwise Operator

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows –

p	**Q**	**p & q**	**p | q**	**p ^ q**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

OPERATOR	**USAGE**	**EXAMPLE**
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100

|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (Read as tilde)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

5. Assignment Operator

OPERATOR	USAGE	EXAMPLE
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C

		<< 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

6. Misc Operators

OPERATOR	USAGE
Sizeof	sizeof operator returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
Condition ? X : Y	Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
,	Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
. (dot) and -> (arrow)	Member operators are used to reference individual members of classes, structures, and unions.
Cast	Casting operators convert one data type to another. For example, int(2.2000) would return 2.
&	Pointer operator & returns the address of a variable. For example &a; will give actual address of the variable.
*	Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

OPERATOR PRECEDENCE

Operator precedence determines the order of evaluation of terms in an expression based on the operator use.

For example, the multiplication operator has higher precedence than the addition operator For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* &	Right to left

	sizeof	
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma		Left to right

Operator in the same row in the above table are of same precedence and are evaluated according to their associativity.

if…else statement

An 'if' statement consists of a boolean expression followed by one or more statements. An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.

Scope Of Variable:-

A scope is a region of the program and broadly speaking there are three places, where variables can be declared −

- Inside a function or a block which is called local variables,
- In the definition of function parameters which is called formal parameters.
- Outside of all functions which is called global variables.

We will learn what is a function and it's parameter in subsequent chapters. Here let us explain what are local and global variables.

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables −

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

int main () {
  // Local variable declaration:
  int a, b;
  int c;

  // actual initialization
  a = 10;
  b = 20;
  c = a + b;

  cout << c;

  return 0;
}
```

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables −

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

// Global variable declaration:
int g;

int main () {
  // Local variable declaration:
  int a, b;

  // actual initialization
  a = 10;
  b = 20;
```

```
  g = a + b;

  cout << g;

  return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example −

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

// Global variable declaration:
int g = 20;

int main () {
  // Local variable declaration:
  int g = 10;

  cout << g;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

10

Default Arguments:-

C++ compiler allows the programmer to assign default values in function prototype. • When the function is called with less parameter or without parameter the default values are used for operation.

```
#include<iostream.h>
 void main()
  {
      int sum(int a,int b=10;int c=15,int d=20);
      int a=2;int b=3; int c=4; int d=5;
      cout<<sum(a,b,c,d);
      cout<<sum(a,b,c);
      cout<<sum(a,b);
      cout<<sum(a);
      cout<<sum(b,c,d);
}
```

```
int sum(int j,int k,int l,int m)
  {
      return(j+k+l+m);
   }
```

Inline Function:-

- C++ provides a mechanism called inline function . When a function is declared as inline the compiler copies the code of the function in calling function i.e function body is inserted in place of function call during compilation.
- Passing of control between coller and collee function is avoided.

Program - Write a program to find square of a number.

```
#include <iostream.h>
Inline float square (float j);
 {
      return (j*j);
 }
 void main()
  {
          int p,q;
          cout<<"Enter a number:";
          cin>>p;
          q=square(p);
          cout<<q;
}
```

Dynamic Allocation New and Delete Operator:-

In Dynamic memory allocation, memory is allocated during the execution of the program.

- The new operator not only creates the object but also allocates memory.
- It allocates correct amount of memory from the heap that is also called as a free store.
- The delete operator not only destroys the object but also releases allocated memory.
- The object created and memory allocated by using new operator should be deleted by delete operator . Otherwise such mismatch operations may corrupt the heap or may crash the system.
- The compiler should have routines to handle such errors.
- The object created by new operator remains in memory until it is released by delete operator.

- Don't destroy the pointer repetitively. The statement delete x does not destroy the pointer x but destroys the object associated with it.
- If object is created but not deleted it occupies unnecessary memory. So it is a good habit to destroy the object & release memory.
- The use of C functions malloc( ),calloc( ),realloc( ) are not fit to be used with object oriented programming.

Example:-

```
#include<iostream.h>
#include<conio.h>
 void main( )
  {
     clrscr( );
     int *p;
     p=new int[3];
     cout<<"Enter 3 integers:";
     cin>>*p>>*(p+1)>>*(p+2);
      for(i=0;i<3;i++)
       cout<<*(p+i)<<unsigned(p+i);
      delete[]p;
  }
```

Classes and Object

Class is the blue print for creating object and it indicates how the data and functions are used when the class is instantiated or in other words when object is created.

Sample Program using Class

```
class a{
        int a1,a2;
        public :
                void input(void){
                        cout << "data";
                        cin >> a1 >> a2;
                }
                void show (void){
                        cout << "The data are:";
                        cout << a1 << a2;
                }
};
void main(){
        a ob1;//object of class a
        ob1.input();
        ob1.show();
        a ob2;//object of class a
        ob2.input();
```

```
        ob2.show();
}
```

Output

```
dataThe data are:11dataThe data are:22
```

Constructor:-

A constructor is a method member having the following properties:-

- The name of the method member must be same as the class name.
- It must not have any return type.
- The method members which are known as constructor can be overloaded.
- The constructor can have default arguments.
- The constructor although they are method members,cannot be called arbitrarily.
- They are invoked at the time of creation of objects.

Example:-

```
Class a
      {
        int a1,a2;
      public:
          a(int p,int q)
            {
              a1=q;
              a2=27;
             }

      void show( )
        {
          Cout<<a1<<a2;
        }
};
void main()
    {
        a ob(2,3);
        ob.show();
    }
```

Destructor:-

When an object is no longer needed it can be destroyed.Destructor is a member function having the character ~(tilde) followed by the name of its class and brackets(i.e, ~classname ( )).It is invoked automatically to reclaim all the resources allocated to the object when the object goes out of scope and it is no longer needed.

Example:-

```
#include <iostream.h>
  class Test
    {
        public:
            Test( );      //Constructor
                ~Test( );       //Destructor
    };
Test :: Test( )
      {
        cout<<"Constructor called"<<endl;
      }
Test ::~Test ( )
    {
       cout<<"Destructor called"<<endl;
    }

void main(){
      Test x;
      cout << "terminating main()" << endl;
}//object x goes out of scope, destructor is called
```

Output

```
constructor of class Test called
terminating main()
terminating of class Test called
```

Access Specifiers

1. private
2. public
3. protected

These keywords are called access specifiers. All the members that follow a keyword belong to that data type. If no keyword is specified then the members are assumed to have private privilege.

Example

```
class a{
        int a6;
        private :
                int a1;
        public :
                int a2;
        private :
                int a3;
        protected :
                int a4;
```

```
        public :
                int a5;
};
void main(){
        a ob;
        ob.a6=1;//cannot be accessed as default data members are private
        ob.a1=2;//cannot be accessed as private data member
        ob.a2=3; //accessible as it is a public data member
        ob.a3=4;//cannot be accessed as private data member
        ob.a4=5;// cannot be accessed as protected data member
        ob.a5=6;//accessible as it is a public data member

}
```

Access Specifier	Accessible To	
	Own class Members	Object of a Class
private:	Yes	No
protected:	Yes	No
public:	Yes	Yes

Defining Member Functions

1. Inside the Class
2. Outside the Class

1. Inside the Class

Example :

```
class a{
        int a1,a2;
        int larger(int p, int q){
                if(p>q){
                        return p;
                }else{
                        return q;
                }
        }
        public :
                void getdata(){
                        cin>> a1>>a2;
                }
                void show(){
```

```
                        cout<<"The larger one is "<<larger(a1,a2);
                }
};
void main(){
        a ob1;
        ob1.getdata();
        ob1.show();
}
```

Any member function within the scope of a class can call another member function.

2. Outside the Class

A member function can be defined outside the class by attaching a membership identity label which tells the compiler which class the function belongs to.

Syntax:

```
Return type class name :: function name(){
        Function body
}
```

Example:

```
class a{
        int a1,a2;
        public :
                void getdata();
                void disp();

};
void a :: getdata(){
        cin>>a1>>a2;
}
void a :: disp(){
        cout<<a1<<a2;;
}
void main(){
        a ob1;
        ob1.getdata();
        ob1.disp();
}
```

Friend Function

A non-member function can have access to private members of the class.

It is possible by declaring a non-member function friend to the class, whose private members are to be accessed.

A friend function is usually has objects as arguments.

A friend function can be declared as friend anywhere (public, private, protected) inside the class.
A single function can be friend to many classes.
Example

```
class a{
        int a1,a2;
        public :
                void getdata(){
                        cin>>a1>>a2;
                }
                friend void add(a oa);
};

void add(a ob){
        return (ob.a1+ob.a2);
}

void main(){
        int x;
        a ob2;
        ob2.getdata();
        x=add(ob2);
        cout<<x;
}
```

Assignment
Q. Write a program to exchange values between two classes by using friend function.

this pointer

this pointer is used for referring the address of currently active object. In C we use &(address of operator) to access the address of the variable. In C++ we can access current object address by using this pointer. Whenever a data member and member function arguments are declared with same name to identify the data member this pointer is used.

```
#include<iostream.h>
#include<conio.h>
class test{
        int a,b;
        public:
                void show(){
                        a=10;
                        b=10;
                        cout<<"object address"<<this;
                        cout<<this->a<<endl;
                        cout<<this->b;
```

```
                }
};
void main(){
        test t;
        clrscr();
        t.show();
        getch();

}
```

Static members

1. Static Data Member
2. Static Member Function

1. Static Data Member

If a data member of a class is declared static it follows following characteristics:-

a. The static data members are not associated with any objects.
b. It is initialized to zero when the first object of the class is created.
c. The static data members are allocated memory only once and those static data members are shared by all the objects of the class.
d. The type and scope of each static data member must be defined outside the class definition.

2. Static Member Function

a. Like member variables member functions can also be declared as static.
b. When a function is defined as static it can access only static member variables and static member function of the same class.
c. The non-static members are not available to this function.
d. The static member function declared in the public section can be invoked using its class name without using its object.
e. Only one copy of static member is created in memory for the entire class.
f. It is also possible to invoke static member function using object of the class.

Example

```
class beta{
        private:
                static int c;
        public:
                static void count(){
                        c++;
                }
                static void display(){
```

```
                                count<<c;
                        }
};
int beta :: c = 0;//definition and initialization of data
void main(){
        beta ob;
        beta :: display();
        beta :: count();
        beta :: count();
        beta :: display();

}
```

Constant Member Function

a. The member function of a class can also be declared as constant using const keyword.
b. The constant function cannot modify the data inside the class.
c. The const keyword is written after the function name.
d. If the function tries to change data the compiler will generate error message.

Inheritance

a. This is the technique by which a class can reuse the members of some other class.
b. The members of which class are reused are known as base class
c. The class which reuses is known as the derived class.

Syntax of Derived Class

```
class DerivedClass : [VisibilityMode] BaseClass
{
        //members of derived class
        //and they can access members of the base class
}
```

The visibility mode indicates to the compiler that the inherited members of the parent class will be placed in which area of the child class.
The following are the three possible styles of derivation:

```
class D : public B//public derivation
{
        //members of D
}
```

```
class D : private B//private derivation
{
        //members of D
}
```

```
class D : B//private derivation by default
{
        //members of D
}
```

d. In inheritance the private members cannot be inherited.
e. Public members are inherited.
f. The members which are inherited from the parent class are treated as if they are the own members of child class.

Example

```
class a{
        int a1;
        public :
                int a2;
                void design(){
                        cout<<"**##**";
                }
};
class b : public a
{
        int b1;
        public :
                int b2;
                void show();
};
void b::show()
{
        cout<<a1;//cannot be accessed as private members are not inherited
        cout<<a2;//accessible as public member
        cout<<b1;//accessible as own member
        cout<<b2;//accessible as public member
        design();
}

void main()
{
        b ob;
        ob.a1 = 5;//cannot be accessed as private members are not inherited
        ob.a2 = 10;//accessible as public member
        ob.b1=15;//accessible as own member
        ob.b2=20;//accessible as public member
        ob.show();
```

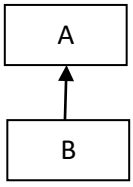
```
        ob.design();
}
```

Types of inheritance

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

1. Single inheritance

A derived class with only base class.

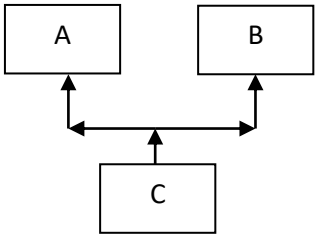


//Parent Class

//Derived Class

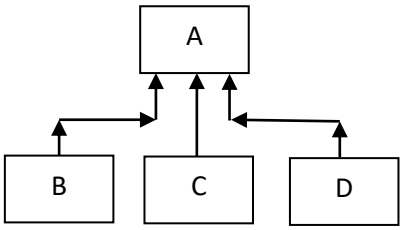
2. Multiple inheritance

Derivation of a class from several (two or more) base classes is called is called multiple inheritance.



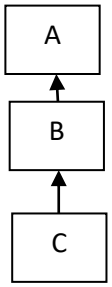
3. Hierarchical inheritance

Derivation of several classes from a single base class i.e. the traits of one class may be inherited by more than one class, is called hierarchical inheritance.



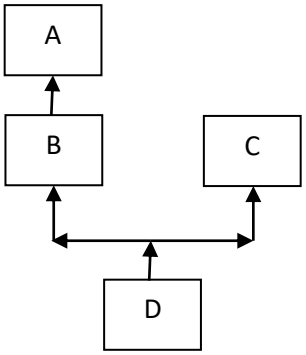
4. Multilevel inheritance

Derivation of a class from another derived class is called multilevel inheritance.



5. Hybrid Inheritance

Derivation of a class involving more than one form of inheritance is known as hybrid inheritance.



Virtual Function

```
class a{
        int b;
        public:
                a()
                {
                        b=10;
                }
                virtual void display()
                {
                        cout << b;
                }
};
class b : public a
{
        int d;
        public :
                b()
                {
```

```
                        d=20;
                }
                void display()
                {
                        cout << d;
                }
};
void main(){
 a oa, *p;
 b ob;
 p=&oa;
p->display();
 p=&ob;
 p->display();
}
```

In the above program the virtual keyword before the display function of the base class performs the runtime binding.

In the first call the display function of base class is executed and in the second call. After assigning the address of derived class to pointer p, display function derived class is executed.

Virtual Base Class

When a class is declared as virtual, the compiler takes necessary action to avoid duplication of member variables. So we make a class as virtual if it is a base class that has been used by more than one derived class as their base class.

Example

```
class A1
{
        protected :
          int a1;
};
class A2 : public virtual A1
{
        protected :
          int a2;
};
class A3 : public virtual A1
{
```

```
        protected:
          int a3;
};
class A4 : public virtual A2,A3
{
        int a4;
        public:
          void get()
          {
                  cin>>a1>>a2>>a3>>a4;
          }
          void put()
          {
                  cout<<a1<<a2<<a3<a4;
          }
};
void main()
{
        A4 ob;
        ob.get();
        ob.put();
}
```

Function Overloading

Function overloading means we can use the same function name to create functions that perform different tasks.

It is known a s function polymorphism in object oriented programming.

It is possible to have a set of function with one function name but with argument list.

Example

```
int add(int,int);
int add(int,int,int);
void main()
{
          int a,b;
          a=add(2,3);
          b=add(4,6,7);
```

```
}
int add(int p,int q)
{
        return (p+q);
}
int add(int x,int y, int z)
{
        return (x+y+z);
}
```

Operator Overloading

1. Assignment Operator Overloading

   Assignment operator can be overloaded to use it as copy constructor i.e. assigning one object to another object just like assigning value of one primitive variable to other primitive variable.

   Example

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

class Distance {
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }
    void operator = (const Distance &D ) {
      feet = D.feet;
      inches = D.inches;
    }
```

```
    // method to display distance
    void displayDistance() {
      cout << "F: " << feet <<  " I:" <<  inches << endl;
    }
};

int main() {
  Distance D1(11, 10), D2(5, 11);

  cout << "First Distance : ";
  D1.displayDistance();
  cout << "Second Distance :";
  D2.displayDistance();

  // use assignment operator
  D1 = D2;
  cout << "First Distance :";
  D1.displayDistance();

  return 0;
#include <iostream>
using namespace std;
const int SIZE = 10;

class safearay {
  private:
    int arr[SIZE];

  public:
    safearay() {
      register int i;
      for(i = 0; i < SIZE; i++) {
        arr[i] = i;
      }
    }

    int &operator[](int i) {
      if( i > SIZE ) {
         cout << "Index out of bounds" <<endl;
         // return first element.
         return arr[0];
      }
```

```
        return arr[i];
    }
};

int main() {
  safearay A;

  cout << "Value of A[2] : " << A[2] <<endl;
  cout << "Value of A[5] : " << A[5]<<endl;
  cout << "Value of A[12] : " << A[12]<<endl;

  return 0;
}
}
```

Output

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

2. **Subscript Operator Overloading**

The subscript operator [] is normally used to access array elements. This operator can be overloaded to enhance the existing functionality of C++ arrays.

Example

```
#include <iostream>
using namespace std;
const int SIZE = 10;

class safearay {
  private:
    int arr[SIZE];

  public:
    safearay() {
      register int i;
      for(i = 0; i < SIZE; i++) {
        arr[i] = i;
      }
    }

    int &operator[](int i) {
```

```
      if( i > SIZE ) {
        cout << "Index out of bounds" <<endl;
        // return first element.
        return arr[0];
      }

      return arr[i];
    }
};

int main() {
  safearay A;

  cout << "Value of A[2] : " << A[2] <<endl;
  cout << "Value of A[5] : " << A[5]<<endl;
  cout << "Value of A[12] : " << A[12]<<endl;

  return 0;
}
```

Output

```
Value of A[2] : 2
Value of A[5] : 5
Index out of bounds
Value of A[12] : 0
```

3. I/O Operator Overloading

C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<. The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.

Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object.

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

class Distance {
  private:
    int feet;           // 0 to infinite
```

```
    int inches;          // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }
    friend ostream &operator<<( ostream &output, const Distance &D ) {
      output << "F : " << D.feet << " I : " << D.inches;
      return output;
    }

    friend istream &operator>>( istream  &input, Distance &D ) {
      input >> D.feet >> D.inches;
      return input;
    }
};

int main() {
  Distance D1(11, 10), D2(5, 11), D3;

  cout << "Enter the value of object : " << endl;
  cin >> D3;
  cout << "First Distance : " << D1 << endl;
  cout << "Second Distance :" << D2 << endl;
  cout << "Third Distance :" << D3 << endl;

  return 0;
}
```

Output

```
Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
```

```
Third Distance :F : 70 I : 10
```

4. Class Member Access Operator Overloading

The class member access operator (->) can be overloaded but it is bit trickier. It is defined to give a class type a "pointer-like" behavior. The operator -> must be a member function. If used, its return type must be a pointer or an object of a class to which you can apply.

The operator-> is used often in conjunction with the pointer-dereference operator * to implement "smart pointers." These pointers are objects that behave like normal pointers except they perform other tasks when you access an object through them, such as automatic object deletion either when the pointer is destroyed, or the pointer is used to point to another object.

The dereferencing operator-> can be defined as a unary postfix operator. That is, given a class −

```
class Ptr {
   //...
   X * operator->();
};
```

Objects of class **Ptr** can be used to access members of class **X** in a very similar manner to the way pointers are used. For example –

```
void f(Ptr p ) {
   p->m = 10 ; // (p.operator->())->m = 10
}
```

The statement p->m is interpreted as (p.operator->())->m. Using the same concept, following example explains how a class access operator -> can be overloaded.

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

// Consider an actual class.
class Obj {
   static int i, j;
```

```
public:
  void f() const { cout << i++ << endl; }
  void g() const { cout << j++ << endl; }
};

// Static member definitions:
int Obj::i = 10;
int Obj::j = 12;

// Implement a container for the above class
class ObjContainer {
  vector<Obj*> a;

  public:
    void add(Obj* obj) {
      a.push_back(obj);  // call vector's standard method.
    }
    friend class SmartPointer;
};

// implement smart pointer to access member of Obj class.
class SmartPointer {
  ObjContainer oc;
  int index;

  public:
    SmartPointer(ObjContainer& objc) {
      oc = objc;
      index = 0;
    }

    // Return value indicates end of list:
    bool operator++() { // Prefix version
      if(index >= oc.a.size()) return false;
      if(oc.a[++index] == 0) return false;
      return true;
    }

    bool operator++(int) { // Postfix version
      return operator++();
    }

    // overload operator->
```

```
    Obj* operator->() const {
      if(!oc.a[index]) {
        cout << "Zero value";
        return (Obj*)0;
      }

      return oc.a[index];
    }
};

int main() {
  const int sz = 10;
  Obj o[sz];
  ObjContainer oc;

  for(int i = 0; i < sz; i++) {
    oc.add(&o[i]);
  }

  SmartPointer sp(oc); // Create an iterator
  do {
    sp->f(); // smart pointer call
    sp->g();
  } while(sp++);

  return 0;
}
```

Output

```
10
12
11
13
12
14
13
15
14
16
15
17
16
```

```
18
17
19
18
20
19
21
```

5. New and Delete operator overloading

The new and delete operators can also be overloaded like other operators in C++. New and Delete operators can be overloaded globally or they can be overloaded for specific classes.

- If these operators are overloaded using member function for a class, it means that these operators are overloaded **only for that specific class**
- If overloading is done outside a class (i.e. it is not a member function of a class), the overloaded 'new' and 'delete' will be called anytime you make use of these operators (within classes or outside classes). This is **global overloading**

Syntax for overloading the new operator :

```
void* operator new(size_t size);
```

The overloaded new operator receives size of type size_t, which specifies the number of bytes of memory to be allocated. The return type of the overloaded new must be void*.The overloaded function returns a pointer to the beginning of the block of memory allocated.

Syntax for overloading the delete operator :

```
void operator delete(void*);
```

The function receives a parameter of type void* which has to be deleted. Function should not return anything.

Both overloaded new and delete operator functions are **static members** by default. Therefore, they don't have access to **this pointer** .

```
// CPP program to demonstrate
// Overloading new and delete operator
// for a specific class
#include<iostream>
#include<stdlib.h>

using namespace std;
class student
{
        string name;
        int age;
public:
        student()
        {
                cout<< "Constructor is called\n" ;
        }
        student(string name, int age)
        {
                this->name = name;
                this->age = age;
        }
        void display()
        {
                cout<< "Name:" << name << endl;
                cout<< "Age:" << age << endl;
        }
        void * operator new(size_t size)
        {
                cout<< "Overloading new operator with size: " << size << endl;
                void * p = ::new student();
                //void * p = malloc(size); will also work fine

                return p;
        }

        void operator delete(void * p)
        {
                cout<< "Overloading delete operator " << endl;
                free(p);
        }
};

int main()
```

```
{
        student * p = new student("Yash", 24);

        p->display();
        delete p;
}
```

Output

```
Overloading new operator with size: 16
Constructor is called
Name:Yash
Age:24
Overloading delete operator
```

Templates

- Template is simple yet very powerfull tool in C++.
- Templates are the foundation of generic programming, which involves writing code in a way i.e. independent of a particular type.
- Template is a blueprint or formula for creating a generic class or function.
- Template is of 2 types:-
    - Function Template
    - Class Template

Function Template:-

1. Function templates are special functions that can operate with generic types.
2. This allows us to create a function template whose functionality can be adapted to more than one tye or class without repeating the entire code for each type.
3. The simple idea is to pass data type as a parameter so that we don't need to write same code for different data types
4. We write a generic function that can be used for different data types.

Function Overloading:-	*Function template:-
int add(int x,int y){} float add(float x,float y){} double add(double x,double y){}	template <typename T> T add(T x,T y) {}

int main()
{
add(5,4);
add(2.3f,4.2f);
add(5.3232,4324.126)
}
	int main
	{
	add<int>(3,7);
	add<float>(3.3,7.5);
	add<double>(3.55,7.66);
	}

Example:-

```
#include<iostream>
int add(int x,int y)
{
return(x+y);
}
float add(float x,float y)
{
return(x+y);
}
double add(double x,double y)
{
return(x+y);
}
int main()
{
cout<<"addition of  two integers"<<add(3,4);
return 0;
}
```

Using Template:-

```
#include<iostream>
template <typename T>
T add(T x,T y)
{
return(x+y);
}
Int main(){
Cout<<add<int>(3,4);
Cout<<add<float>(3.4f,2.5f);
Cout<<add<double>(3.45,4.23);
return 0;
```

```
}
```

Using Two different data types:-

```
#include<iostream>
template <typename T,typename U>
U add(T x,U y)
{
return(x+y);
}
int main(){
Cout<<add<double>(3.0,4.5);
return 0;
}
```

O/p-7.5

Class Templates:-

Sometimes we need a class implementation i.e. same for all classes oly the data type used are different.
Normally we would need to create a different class for each data type or create different member variables and functions with single class.
In class Template we write a class that can be used for different data types.

Class stack
{
Public:
Int arr[5];
Private:
Push();
Pop();
}
	Class stack
	{
	Public:
	Char arr[5];
	Private:
	Push();
	Pop();
	}

Example:

```
 #include<iostream>
template <typename T>
class weight
{
  private:
        T kg;
   public:
   void setData(T x)
```

```
   {
     Kg=x;
  }
T getData()
  {
   return kg;
  }
 };
int main()
{
  weight <int>obj;
obj.setData(5);
cout<<obj.getData();

weight <double>obj1;
obj1.setData(4.56575)  ;
cout<<obj1.getData();
return 0;
}
```

Assignment:-Write a class template to pass another a double data type in the above example.

STL

The C++ STL (Standard Template Library) is a generic collection of class templates and algorithms that allow programmers to easily implement standard data structures like queues, lists, and stacks.
STL provides many algorithms.
Some of the most used algorithms are :-

1. sort (first_iterator, last_iterator) – To sort the given vector.
2. reverse(first_iterator, last_iterator) – To reverse a vector.
3. *max_element (first_iterator, last_iterator) – To find the maximum element of a vector.
4. *min_element (first_iterator, last_iterator) – To find the minimum element of a vector.
5. accumulate(first_iterator, last_iterator, initial value of sum) – Does the summation of vector elements
6. count(first_iterator, last_iterator,x) – To count the occurrences of x in vector.
7. find(first_iterator, last_iterator, x) – Points to last address of vector ((name_of_vector).end()) if element is not present in vector.
8. binary_search(first_iterator, last_iterator, x) – Tests whether x exists in sorted vector or not.
9. lower_bound(first_iterator, last_iterator, x) – returns an iterator pointing to the first element in the range [first,last) which has a value not less than ‘x’.

10. upper_bound(first_iterator, last_iterator, x) – returns an iterator pointing to the first element in the range [first,last) which has a value greater than 'x'.

Example

```
// A C++ program to demonstrate working of sort(),
// reverse()
#include <algorithm>
#include <iostream>
#include <vector>
#include <numeric> //For accumulate operation
using namespace std;

int main()
{
	// Initializing vector with array values
	int arr[] = {10, 20, 5, 23 ,42 , 15};
	int n = sizeof(arr)/sizeof(arr[0]);
	vector<int> vect(arr, arr+n);

	cout << "Vector is: ";
	for (int i=0; i<n; i++)
		cout << vect[i] << " ";

	// Sorting the Vector in Ascending order
	sort(vect.begin(), vect.end());

	cout << "\nVector after sorting is: ";
	for (int i=0; i<n; i++)
	cout << vect[i] << " ";

	// Reversing the Vector
	reverse(vect.begin(), vect.end());

	cout << "\nVector after reversing is: ";
	for (int i=0; i<6; i++)
		cout << vect[i] << " ";

	cout << "\nMaximum element of vector is: ";
	cout << *max_element(vect.begin(), vect.end());

	cout << "\nMinimum element of vector is: ";
	cout << *min_element(vect.begin(), vect.end());
```

```
        // Starting the summation from 0
        cout << "\nThe summation of vector elements is: ";
        cout << accumulate(vect.begin(), vect.end(), 0);

        return 0;
}
```

Output

```
Vector before sorting is: 10 20 5 23 42 15
Vector after sorting is: 5 10 15 20 23 42
Vector before reversing is: 5 10 15 20 23 42
Vector after reversing is: 42 23 20 15 10 5
Maximum element of vector is: 42
Minimum element of vector is: 5
The summation of vector elements is: 115
```

Sequence Containers

Sequence containers implement data structures which can be accessed sequentially.

- array: Static contiguous array (class template)
- vector: Dynamic contiguous array (class template)
- deque: Double-ended queue (class template)
- forward_list: Singly-linked list (class template)
- list : Doubly-linked list (class template)

Associative Containers

Associative containers implement sorted data structures that can be quickly searched (O(log n) complexity).

- Set: Collection of unique keys, sorted by keys (class template)
- Map: Collection of key-value pairs, sorted by keys, keys are unique (class template).
- multiset: Collection of keys, sorted by keys (class template)
- multimap: Collection of key-value pairs, sorted by keys (class template)

Iterators

Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequence of numbers, characters etc. They reduce the complexity and execution time of program.

Operations of Iterators

1. begin(): this function is used to return the beginning position of the container
2. end():this function is used to return the after end position of the container
3. advance():this function is used to increment the iterator position till the specified number mentioned in its arguments
4. next():this function returns the new iterator would point after advancing the positions mentioned in the arguments.
5. prev():this function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.
6. inserter():This function is used to insert the elements at any position in the container. It accepts 2 arguments, the container and iterator to position where the elements have to be inserted.

Vectors

1. Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.
2. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
3. In vectors, data is inserted at the end.
4. Inserting at the end takes differential time, as sometimes there may be a need of extending the array.
5. Removing the last element takes only constant time because no resizing happens.
6. Inserting and erasing at the beginning or in the middle is linear in time.

Example

```
// C++ program to illustrate the
// iterators in vector
#include <iostream>
#include <vector>

using namespace std;

int main()
{
        vector<int> g1;
```

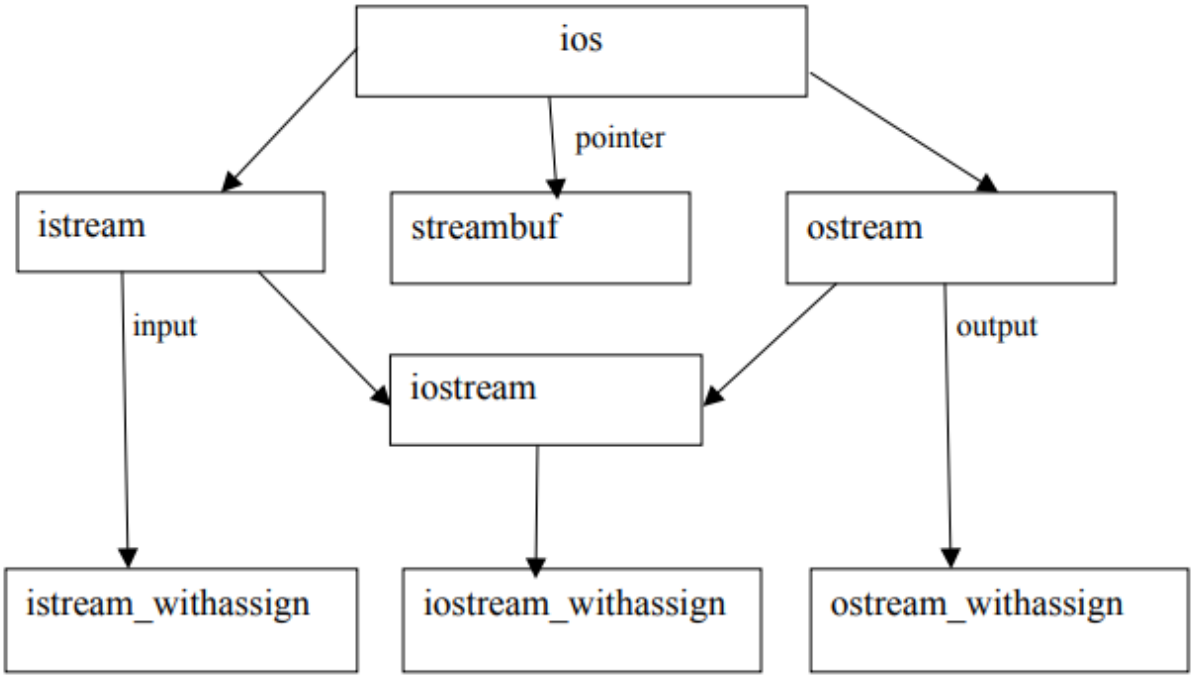
```
        for (int i = 1; i <= 5; i++)
                g1.push_back(i);

        cout << "Output of begin and end: ";
        for (auto i = g1.begin(); i != g1.end(); ++i)
                cout << *i << " ";

        cout << "\nOutput of cbegin and cend: ";
        for (auto i = g1.cbegin(); i != g1.cend(); ++i)
                cout << *i << " ";

        cout << "\nOutput of rbegin and rend: ";
        for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
                cout << *ir << " ";

        cout << "\nOutput of crbegin and crend : ";
        for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
                cout << *ir << " ";

        return 0;
}
```

Output

```
Output of begin and end: 1 2 3 4 5
Output of cbegin and cend: 1 2 3 4 5
Output of rbegin and rend: 5 4 3 2 1
Output of crbegin and crend : 5 4 3 2 1
```

Stream Classes

1. The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files.
2. These classes are called stream classes. Following diagram shows the hierarchy of the stream classes used for input and output operations with the console unit.
3. These classes are declared in the header file iostream. The file should be included in all programs that communicate with the console unit.



4. ios is the base class for istream(input stream) and ostream(output stream) which are base classes for iostream(input/output stream).
5. The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.
6. The class ios provides the basic support for formatted and unformatted input/output operations.
7. The class istream provides the facilities for formatted and unformatted input while the class ostream(through inheritance) provides the facilities for formatted output.
8. The class iostream provides the facilities for handling both input output streams.
9. Three classes namely istream_withassign, ostream_withassign and iostream_withassign add assignment operators to these classes.

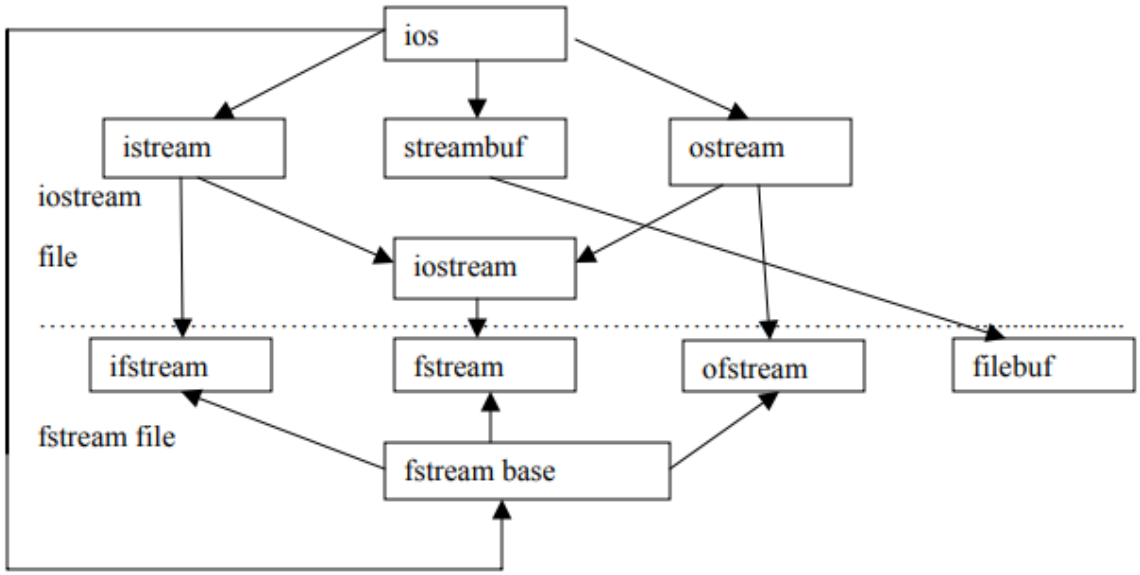
Class Name	Contents
ios(General input/output stream class)	Contains basic facilities that are ued by all other input and output classes Also contains a pointer to buffer object(streambuf object) Declares constants and functions that are necessary for handling formatted input and output operations
istream(input stream)	Inherits the properties of ios Declares input functions such as get(),getline() and read() Contains overloaded extraction operator>>
ostream(output stream)	Inherits the property of ios Declares output functions put() and write() Contains overloaded insertion operator <<

iostream (input/output stream)	Inherits the properties of ios stream and ostream through multiple inheritance and thus contains all the input and output functions
Streanbuf	Provides an interface to physical devices through buffer Acts as a base for filebuf class used ios files

10. Objects cin and cout are used for input and output of data by using the overloading of >> and << operators.
11. The >> operator is overloaded in the istream class and << is overloaded in the ostream class.
12. The following is the format for reading data from keyboard:
    cin>>variable1>>variable2>>……..>>variable n, where variable 1 to variable n are valid C++ variable names that have declared already.
13. This statement will cause the computer to stop the execution and look for the input data from the keyboard.the input data for this statement would be
    data1 data2……..data n
    The input data are separated by white spaces and should match the type of variable in the cin list spaces, newlines and tabs will be skipped.
14. The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. For example consider the code
    int code;
    cin>> code;
    Suppose the following data is entered as input
    42580
    the operator will read the characters upto 8 and the value 4258 is assigned to code. The character D remains in the input streams and will be input to the next cin statement.
    The general form of displaying data on the screen is
    cout <<item1<<item2<<……<<item n
    The item item1 through item n may be variables or constants of any basic type.

File Stream Classes

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



Steps of File Operations:-

For using a disk file the following things are necessary

1. Suitable name of file
2. Data type and structure
3. Purpose
4. Opening Method

Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains Openprot constant used in the open() of file stream classes.Also contain close() and open() as members.
fstreambase	Provides operations common to file streams.Serves as a base for fstream,ifstream and ofstream class.Contains open() and close() functions.
ifstream	Provides input operations.Contains open() with default input mode.Inherits the functions get(),getline(),read(),seekg(),tellg() functions from istream.
ofstream	Provides output operations.Contains open() with default output mode.Inherits put(),seekp(),tellp() and write() functions from ostream
fstream	Provides support for simultaneous input and output operations.Contains open with default input mode.Inherits all the functions from istream and ostream classes through iostream

The filename is a string of characters that makeup a valid filename for the operating system. It may contain two parts ,primary name and optional period with extension.

Examples are Input.data, Test.doc etc. For opening a file firstly a file stream is created and then it is linked to the filename.A file stream can be defined using the classes ifstream, ofstream and fstream that contained in the header file fstream.The class to be used depends upon the purpose whether the write data or read data operation is to be performed on the file.A file can be opened in two ways:
(a) Using the constructor function of class.
(b) Using the member function open() of the class.
The first method is useful only when one file is used in the stream.The second method is used when multiple files are to be managed using one stream.

Opening Files using Constructor:

While using constructor for opening files,filename is used to initialize the file stream object.This involves the following steps

(i) Create a file stream object to manage the stream using the appropriate class i.e the class ofstream is used to create the output stream and the class ifstream to create the input stream.

(ii) Initialize the file object using desired file name.
For example, the following statement opens a file named "results" for output:

```
ofstream outfile("results"); //output only
```

This create outfile as an ofstream object that manages the output stream. Similarly ,the following statement declares infile as an ifstream object and attaches it to the file data for reading (input).

```
ifstream infile ("data"); //input only
```

The same file name can be used for both reading and writing data.
For example

```
Program1 ……..
……
ofstream outfile ("salary"); //creates outfile and connects salary to it
……
……..
Program 2
……
……
```

ifstream infile (“salary”); //creates infile and connects salary to it
……..
…….
The connection with a file is closed automatically when the stream object expires i.e when a program terminates.In the above statement ,when the program 1 is terminated,the salary file is disconnected from the outfile stream.The same thing happens when program 2 terminates.
Instead of using two programs,one for writing data and another for reading data ,a single program can be used to do both operations on a file.
……
……. outfile.close(); //disconnect salary from outfile and connect to infile ifstream infile (“salary”);
…….
……
infile.close();
The following program uses a single file for both reading and writing the data .First it take data from the keyboard and writes it to file.After the writing is completed the file is closed.The program again opens the same file read the information already written to it and displays the same on the screen.
Program

```
WORKING WITH SINGLE FILE
//Creating files with constructor function
#include <iostream.h>
#include<fstream.h>
int main(){
ofstream outf(“ITEM”);
cout << “enter item name”;
char name[30];
cin>>name;
outf<<name<<”\n”;
cout<<”enter item cost”;
float cost;
cin>>cost;
outf<<cost<<”\n”;
outf.close();
ifstream inf(“item”);
inf>>name;
inf>>cost;
cout<<”\n”;
cout<<”item name: ”<<name<<”\n”;
```

```
cout<<"item cost:"<<cost<<"\n";
inf.close();
return 0;
}
```

Opening Files using open()

The function open() can be used to open multiple files that uses the same stream object.

For example to process a set of files sequentially,in such case a single stream object can be created and can be used to open each file in turn.

This can be done as follows;

File-stream-class stream-object;

stream-object.open ("filename");

The following example shows how to work simultaneously with multiple files

Program

```
#include<iostream.h>
#include<fstream.h>
int main()
{
        ofstream fout;
        fout.open("country");
        fout<<"United States of America \n";
        fout<<"United Kingdom";
        fout<<"South Korea";
        fout.close();
        fout.open("capital");
        fout<<"Washington \n";
        fout<<"London \n";
        fout<<"Seoul\n";
        fout.close();
        const int N=80;
        char line[N];
        ifstream fin;
        fin.open("country");
        cout<<"contents of country file\n";
```

```
        while(fin)
        {
                fin.getline(line,N);
                cout<<line;
        }
        fin.close();
        fin.open("capital");
        cout<<"contents of capital file \n";
        while(fin)
        {
                fin.getline(line,N);
                cout<<line;

        }
        fin.close();
        return 0;
}
```

Finding End of File:

While reading a data from a file,it is necessary to find where the file ends i.e end of file.The programmer cannot predict the end of file,if the program does not detect end of file,the program drops in an infinite loop.To avoid this,it is necessary to provide correct instruction to the program that detects the end of file.Thus when end of file of file is detected,the process of reading data can be easily terminated. An ifstream object such as fin returns a value of 0 if any error occurs in the file operation including the end-of – file condition.Thus the while loop terminates when fin returns a value of zero on reaching the end-of –file condition.There is an another approach to detect the end of file condition.The statement

```
        if(fin1.eof() !=0 )
        {
        exit(1);
        }
```

returns a non zero value if end of file condition is encountered and zero otherwise.Therefore the above statement terminates the program on reaching the end of file.

File Opening Modes:

The ifstream and ofstream constructors and the function open() are used to open the files.Upto now a single arguments a single argument is used that is filename.However,these functions can take two arguments, the second one for specifying the file mode.The general form of function open() with two arguments is:

stream-object.open("filename",mode);

The second argument mode specifies the purpose for which the file is opened.The prototype of these class member functions contain default values for second argument and therefore they use the default values in the absence of actual values.The default values are as follows :

ios::in for ifstream functions meaning open for reading only.

ios::out for ofstream functions meaning open for writing only.

The file mode parameter can take one of such constants defined in class ios.The following table lists the file mode parameter and their meanings.

Parameter	Meaning
ios::app	Append to end-of-file
ios::ate	Go to end-of-file on opening
ios::binary	Binary file
ios::in	Open file for reading only
ios::nocreate	Open fails if file the file does not exist
ios::noreplace	Open fails if the file already exists
ios::out	Open file for writing only
ios::trunk	Delete the contents of the file if it exists

File Pointers and Manipulators:

Each file has two pointers known as file pointers,one is called the input pointer and the other is called output pointer..The input pointer is used for reading the contents of of a given file location and the output pointer is used for writing to a given file location.Each time an input or output operation takes place,the appropriate pointer is automatically advanced.

Default actions:

When a file is opened in read-only mode,the input pointer is automatically set at the beginning so that file can be read from the start.Similarly when a file is opened in write-

only mode the existing contents are deleted and the output pointer is set at the beginning.This enables us to write the file from start.In case an existing file is to be opened in order to add more data,the file is opened in 'append' mode.This moves the pointer to the end of file.

Functions for Manipulations of File pointer:

All the actions on the file pointers takes place by default.For controlling the movement of file pointers file stream classes support the following functions

seekg()	Moves get pointer (input)to a specified location.
seekp()	Moves put pointer (output) to a specified location.
tellg()	Give the current position of the get pointer.
tellp()	Give the current position of the put pointer.

For example, the statement

infile.seekg(10);

moves the pointer to the byte number 10.The bytes in a file are numbered beginning from zero.Therefore ,the pointer to the 11th byte in the file.Consider the following statements:

ofstream fileout;

fileout.open("hello",ios::app);

int p=fileout.tellp();

On execution of these statements,the output pointer is moved to the end of file "hello" And the value of p will represent the number of bytes in the file.

Specifying the Offset:

'Seek' functions seekg() and seekp() can also be used with two arguments as follows:

seekg (offset,refposition);

seekp (offset,refposition);

The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition.The refposition takes one of the following three constants defined in the ios class:

ios::beg	Start of file
ios::cur	Current position of the pointer
ios::end	End of file

The seekg() function moves the associated file's 'get' pointer while the seekp() function moves the associated file's 'put 'pointer.The following table shows some sample pointer offset calls and their actions.fout is an ofstream object.

Seek Call	**Action**
fout.seekg(o,ios::beg)	Go to start
fout.seekg(o,ios::cur)	Stay at the current position
fout.seekg(o,ios::end)	Go to the end of file
fout.seekg(m,ios::beg)	Move to (m+1)th byte in the file
fout.seekg(m,ios::cur)	Go forward by m byte from current position
fout.seekg(-m,ios::cur)	Go backward by m bytes from current position.
fout.seekg(-m,ios::end)	Go backward by m bytes from the end

Pointer

A **pointer** is a variable whose value is the address of another variable.

Syntax:

type *var-name;

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Example:

```
#include <iostream>

using namespace std;

int main () {
   int  var = 20;   // actual variable declaration.
```

```
  int  *ip;        // pointer variable

  ip = &var;       // store address of var in pointer variable

  cout << "Value of var variable: ";
  cout << var << endl;

  // print the address stored in ip pointer variable
  cout << "Address stored in ip variable: ";
  cout << ip << endl;

  // access the value at the address available in pointer
  cout << "Value of *ip variable: ";
  cout << *ip << endl;

  return 0;
}
```

Output

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

Address of Operator &

The & is a unary operator that returns the memory address of its operand. For example, if var is an integer variable, then &var is its address. This operator has the same precedence and right-to-left associativity as the other unary operators.

You should read the & operator as **"the address of"** which means **&var** will be read as "the address of var".

Indirection Operator *

It is a unary operator that returns the value of the variable located at the address specified by its operand.

Example

```
#include <iostream>

using namespace std;
```

```
int main () {
   int  var;
   int  *ptr;
   int  val;

   var = 3000;

   // take the address of var
   ptr = &var;

   // take the value available at ptr
   val = *ptr;
   cout << "Value of var :" << var << endl;
   cout << "Value of ptr :" << ptr << endl;
   cout << "Value of val :" << val << endl;

   return 0;
}
```

Output

```
Value of var :3000
Value of ptr :0xbff64494
Value of val :3000
```

Pointers Vs Arrays

Pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing. Consider the following program −

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
```

```
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // let us have array address in pointer.
   ptr = var;

   for (int i = 0; i < MAX; i++) {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // point to the next location
      ptr++;
   }

   return 0;
}
```

Output

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

However, pointers and arrays are not completely interchangeable. For example, consider the following program –

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};
```

```
  for (int i = 0; i < MAX; i++) {
    *var = i;    // This is a correct syntax
    var++;       // This is incorrect.
  }

  return 0;
}
```

It is perfectly acceptable to apply the pointer operator * to var but it is illegal to modify var value. The reason for this is that var is a constant that points to the beginning of an array and can not be used as l-value.

Because an array name generates a pointer constant, it can still be used in pointer-style expressions, as long as it is not modified. For example, the following is a valid statement that assigns var[2] the value 500 –

```
*(var + 2) = 500;
```

Function Pointers

A function pointer is a variable that stores the address of a function that can later be called through that function pointer.

Syntax

void (*foo)(int);

Example

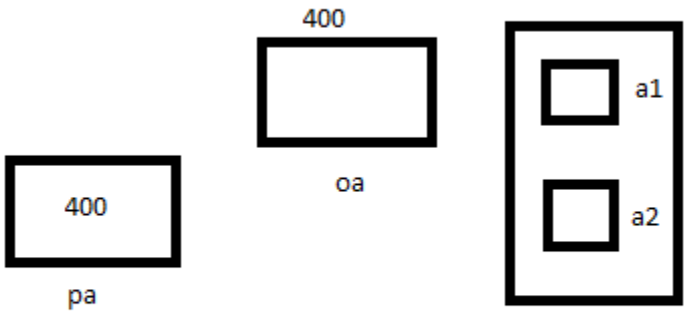
```
#include <iostream.h>
void my_int_func(int x)
{
   Cout<<x;
}


int main()
{
   void (*foo)(int);
```

```
   foo = &my_int_func;

   /* call my_int_func (note that you do not need to write (*foo)(2) ) */
   foo( 2 );
   /* but if you want to, you may */
   (*foo)( 2 );

   return 0;
}
```

Pointer to Object

Program

```
class a
{
        int a1;
        public :
                int a2;
                void show()
                {
                        cout<<a1<<a2;
                }
};
void main()
{
        a oa;
        a *pa;
        oa.a1=5;//Not accessible
        oa.a2=10//accessible
        pa->a2=20;//not accessible as pointer is not assigned any object
        pa=&oa;
        pa->a2=20;//accessible
        pa->a1=10;//not accessible as private
        pa->show();
}
```



Program

```
class a
{
        public :
                void show()
                {
                        cout<<"A SHOW";

                }
                void disp()
                {
                        cout<<"A DISP";
                }
};
class b : public a
{
        public :
                void show()
                {
                        cout<<"B SHOW";
                }
                void disp()
                {
                        cout<<"B DISP";
                }
```

```
};
void main()
{
        b ob;
        ob.show();//B SHOW
        a oa;
        oa.show();//A SHOW
        a *pa;
        pa=&oa;
        pa->show();//A SHOW
        pa->disp();//A DISP
        pa=&ob;
        pa->show();
        pa->disp();

}
```

New Operator

The new operator is used to create object.

Syntax

Pointer variable = new datatype;

Here the pointer variable is the pointer of any built-in data type including array or any user defined data type including class and structure.

New operator allocate sufficient memory to hold data object.

Example :- int *p = new int;

We can also initialize the memory by using new operator. The syntax is:

Pointer variable = new type value;

Example:- &p=new int(25);

The new operator can be used to create memory space for any data type including user defined data type. For creating array dynamic syntax is

pointer variable = new type (size);

Example : - int *p=new int(r);

Delete Operator

The delete operator is used to deallocate the memory created by new operator at runtime.

Once the memory is no longer needed it should be free so that it becomes available for reuse.

Syntax

delete pointer_variable;

Example:- delete p;

In the delete operator the pointer variable is the pointer that points to the data object created by the new.

Advantages of new operator over malloc function

1. It automatically compute size of data object so no need to use size of operator.
2. It automatically return the current pointer type so there is no need to use type cast.
3. It is possible to initialize object while creating memory space.
4. The new and delete operation can be overloaded.

……