Principles of Object Oriented Programming:

1. Basic concept of OOP: - Object oriented programming is a new way of solving problems with computers. In the 1970, the concept of the object becomes popular among researchers of programming languages. An object is a combination or collection of data and code designed to emulate a physical or abstract entity.

   Object-Oriented Programming is a programming methodology that associates data structures with a set of operators, which act upon it. In OOPs terminology, an instance of such an entity is known as an object. It gives importance to relationships between objects rather than implementation details. Hiding the implementation details within a object results in the user being re concerned with an object's relationship to the rest of the system, than the implementation of the object's behaviour.

2. Fundamental feature of OOP: -

   a) Encapsulation: - It is a mechanism that associates the code and the data it manipulates into a single unit. In C++, this is supported b a construct called class.

   b) Data Abstraction: - The technique of creating new data types that are well suited to an application to be programmed is known as data abstraction. It provides the ability to create user defined data types. The class in a construct in C++ for creating userdefned data type called abstract data type (ADT).

   c) Inheritance: - It allows the extension and reuse of existing code without having to rewrite the code from scratch. Inheritance involves the creation of new classes from the existing class.

   d) Polymorphism: - It allows a single name/operator to be associated with different operations depending on the type of data passed to it. In C++, it is achieved by function overloading, operator overloading and dynamic binding.

   e) Message Passing: - It is the process of invoking an operation on an object. In response to a message, the corresponding method (function) is executed in the object

   f) Extensibility: - It is a feature, which allows the extension of the functionality of the existing software components. In C++, this is achieved through abstract classes and inheritance

   g) Persistence: - The phenomenon where the object outlives the program execution time and exists between executions of a program is known as persistence. C++ does not support it.

   h) Delegation: - it is an alternative to class inheritance.

   i) Genericity: - It is a technique for defining software components that have more than one interpretation depending on the data type of parameters. Thus, it allows the declaration of data

items without specifying their exact data type. In C++, genericity is realized through function templates and class templates.

3. Concept of Class and object: - The object with same data structure and behavior are grouped into a class. All those objects possessing similar properties are grouped into the same unit.

4. Advantage of OOP: -

a) Since the objects are autonomous entities and share their responsibilitys only by executing methods relevant to the received messages, each object lends itself to greater modularity.

b) Information hiding and data abstraction increase reliability.

c) Dynamic binding increase flexibility by permitting the addition of a new class of objects without having to modify the existing code.

5. Concept of cin and cout: - C++ uses the bit-wise left-shift operator for performing output operation. The syntax is cout << variable;

The word cout is followed by the symbol <<, called the insertion or put-to operator, and then with the items that are to be output. Input from the standard stream is performed using the cin object. C++ uses the bit-wise right-shift operator for performing input operation. The syntax is

cin >> variable

The word cin is followed by the symbol >> and then with the variable, into which the input data is to be stored.

2. Procedural programming vs OOP: -

a) In procedural programming, programs are organized in the form of subroutines. But in OOP emphasizes on the object rather than the subroutines.
b) In procedural programming all data items are global. In OOPs, data is compartmentalized or encapsulated with the associated functions.
c) In procedural programming, subroutines are abstracted to avoid repetitions. But in OOPs, data abstraction is introduced in addition to procedural abstraction.
d) In procedural programming, data and operations are design separately. But in OOPs, data and associated operations are unified into a single unit called object.

Scope resolution operator (::) : - C++ supports a mechanism to access a global variable from a function in which a local variable is defined with the same name as a global variable. It is achieved using the scope resolution operator. The syntax is :: global variable

The global variable to be accessed must be preceded by the scope resolution operator

Reference variable: - Reference variable behaves similar to a value variable and a pointer variable. In the program code, it is used similar to that of a value variable, but has an action of a pointer variable. In other words, a reference variable acts as an alias variable, for the other value variable. The syntax is

Data type & referencevariable = valuvariable;

Functions:

1. Inline Function: - Inline functions are those whose function body is inserted in place of the function call statement during the compilation process. The concept of inline functions is similar to macro functions of C. An inline function definition is similar to an ordinary function except that the keyword inline precedes the function definition. The significant feature of inline functions is there is no explicit function call and body is substituted at the point of inline function call, thereby, the run-time overhead for function linkage mechanism is reduced.

```
inline returntype functionName(parameter)
{
    statements ;
}
```

C++ treats all the member functions that are defined within a class as inline functions and those defined outside as non-inline (outline). Member function declared outside the class declaration can be made inline by prefixing the inline to its definition.

The feature of inline member functions is useful only when they are short. Declaring a function having many statements, as inline is not advisable, since it make the object code of a program very large. The following simple rules in deciding as to when inline functions should be used:

- In general inline function should not be used.
- For a very short and simple function, inline function can be used.
- It is only useful to implement an inline function if the time spent during a function call is more compared to the function body execution time.

Inline function has one disadvantage, the compiler inserts the actual code and therefore it should be known at compile-time. Hence, an inline function cannot be located in a run-time library.

Overloaded function: - Function polymorphism, or function overloading is a concept that allows multiple functions to share the same name with different argument types. Function polymorphism implies that the function definition can have multiple forms. Assigning one or more function body to the same name is known as function overloading or function name overloading.

In C++, two or more functions can be given the same name provided the signature (parameters count or their data types) of each of them is unique either in the number or data type of their arguments. C++ does not permit overloading of function differing only in their return value.

Default arguments: - Normally, a function call should specify all the arguments used I the function definition. In a C++ function all, when one or more arguments are omitted, the function may be defined to take default values for the omitted arguments by providing the default values in the function prototype.

Parameters without default arguments are placed first and those with default values are placed later. Hence the feature of default arguments allows the same function to be called with fewer arguments than defined in the function prototype.

To establish a default value, the function prototype or the function definition must be used. The compiler checks the function prototype with the arguments in the function call to provide default values (if applicable) to those arguments, which are omitted. The arguments specified in the function call explicitly always override the default values specified in the function prototype. In a function call, all the trailing missing arguments are replaced by default arguments. Functions may be defined with more than one default argument.

Default arguments must be known to the compiler prior to the invocation of a function. It reduces the burden of passing arguments explicitly at the point of the function call.

Example:

```
void printline (char = '*', int = 10);
void main( )
{       printline( );
        printline('#');
        printline('#',15);
}
```

In this example function printline has two default arguments (* and 10). When we call the function first without any arguments, compiler takes all the default arguments. When we call it second time with one argument (#), compiler replace the first default argument by # and take the second argument the default argument (10). In the third function call we provide two arguments (#,15), compiler replace both the default argument by (#,15).

A default argument can appear either in the function prototype or definition. Once it is defined, it cannot be redefined. Variable names may be omitted while assigning default values in the prototype.

Return by reference: - A function that returns a reference variable is actually an alias for the referred variable. This method of returning references is used in operator overloading to form a cascade of member function calls specified in a single statement. The function, which returns by reference, can appear on the lift-hand side of an assignment statement.

Example: in this example the function max return a reference of x or y which is larger. In the function call point the value 125 will be assigned in that variable which will be returned by the max(a,b);

```
int &max(int &x, int &y);
void main( )
{       int a,b,c;
        cin>>a>>b;
        max(a,b) = 125;
        cout<<a<<b;
}
int &max(int &x, int &y)
{   (x > y) ? return(x) : return(y);
}
```

5. Constructors and Destructors:

Concept of constructors: - A constructor is a special member function whose main operation is to allocate the required resources such as memory and initialize the objects of its class. A constructor is distinct from other member functions of the class and it has the same name as its class. It is executed automatically when an object of that class is created. It is generally used to initialize object member parameters and allocate the necessary resources to the object members. The constructor has no return value specification. It is of course possible to define a class without any constructor. The syntax for defining a constructor is same as the function definition except that constructor has no return type. It can be defined either within, or outside the body of a class. It can access any data member like all other member functions but cannot be invoked explicitly and must have public status to serve its purpose.

Type of Constructor: - The constructor, which does not take arguments explicitly, is called default constructor.

i) Parameterized constructor: - Constructors can be invoked with arguments. The argument list cab be specified with braces. Constructor with arguments is called parameterized constructors. Like the function overloading, constructor overloading is also possible in C++.

ii) Constructors with Default Arguments: - Like function with default argument, constructor can be also designed with default argument. If any arguments are passed during the creation of an object, the compiler selects the

suitable constructor with default arguments. A constructor that has all default arguments is similar to a default constructor, because it can be called without any explicit arguments. This may also lead to error.

```
Class complex
{         int x;
          int y;
  public:
          complex ( )
          {    x = y = 0;
          }
          complex (int a=1, int b=2)
          {     x=a; y=b;
          }
};
void main ( )
{    complex c1;
     complex c2(5);
     complex c3(3,4);
}
```

In this example class complex contain two constructors. When we create object c1 without any parameters then compiler call the default constructor. When we create object c2 with parameter (5) then compiler call parameter constructor with default argument b=2. When we create c3 with parameter (3,4) then compiler call the parameter constructor without any default argument because both the arguments are passed during the creation of c3.

iii) Copy Constructor: - The parameter of a constructor can be of any of the data types except an object of its own class as a value parameter. However, a class's own object can be passed as a reference parameter, not a value parameter. Such constructor, having a reference to an instance of its own class as an argument is known as copy constructor.

```
Class complex
{         int x;
          int y;
  public:  complex (int a=1, int b=2)
           {     x=a; y=b;
           }
           complex (complex &a)
          {  x=a.x ; y = a.y;
          }
};
void main ( )
{    complex c1;
     complex c2(3,4);
     complex c3 = c2;
}
```

A copy constructor copies the data members from one object to another. If an argument is passed b value, its copy constructor would cal itself to copy the actual parameter to the formal parameter. This process would go on-and-on until the system runs out of memory.

A copy constructor also executes when the arguments are passed by value to functions, and when values are returned from functions.

Dynamic Initialization of objects: - Allocation of memory to objects at the time of their construction is known as dynamic construction. The allocated memory can be released when the object is no longer needed at runtime and is known as dynamic destruction. We have to use "new" operator for dynamic allocation and "delete" operator for dynamic destruction. The following points can be emphasized on dynamic initialization of object:

- A constructor of the class makes sure that the data

```
class list
{   int size;
    int *x;
    public :
          list (int n)
          {   size = n;
              x = new int [size];
          }
          ~ list ( )
          {    delete x;
          }
};
main()
{   int k;
    cin >> k;
    list a (k);
}
```

members are initially 0-pointer (NULL)

- A constructor with parameters allocates the right amount of memory resources.
- A destructor releases all the allocated memory

Difference between Constructor & Destructor: -

	Constructor	Destructor
1	Arguments can be passed	Arguments cannot be passed
2	More than one constructor can be declared within one class by differentiating their no of arguments	Since, it cannot take argument so only one destructor can be declared
3	It is not virtual	It can be virtual
4	Name is same with class	Name is same with class but started by symbol (~)

6. Operator Overloading: - The operator-overloading feature of C++ is one of the methods of realizing polymorphism. It means performing many actions with a single operator. The precedence relation of overloadable operators and their expression syntax remains the same even after overloading. Except the following operators, others can be overloaded.

:: (scope resolution) .(member selection) .* (member selection through pointer to member)

The keyword *operator* facilitates overloading of the C++ operator. The operator overloaded in a class is known as *overloaded operator function*

```
ReturnType operator OperatorSymbol (argument list)
{
    body of the operator function
}
```

Overloading without explicit arguments to an operator function is known as unary operator overloading and overloading with a single explicit argument is known as binary operator overloading. Similar to other functions of a class, an overloaded operator member function can be either defined within the body of a class or outside the body of a class, but prototype must declare inside the class.

The binary overloaded operator function takes the first object as an implicit operand and the second operand must be passed explicitly. The left hand side operand is used to invoke the operator function and the right hand side operand is passed as an argument to the operator function.

Inheritance: - Inheritance is a technique of organizing information in a hierarchical form. It allows new classes to be built from older and less specialized classes instead of being rewritten from scratch. The technique of building new classes from the existing classes is called inheritance.

Inheritance, a prime feature of OOPs can be stated as the process of creating new classes (called derived classes), from the existing classes (called base classes). The derived class inherits all the capabilities of the base class and can add new feature into it. The base class remains unchanged. C++ allow another access specifier that is protected which is mainly used in inheritance. The syntax is :

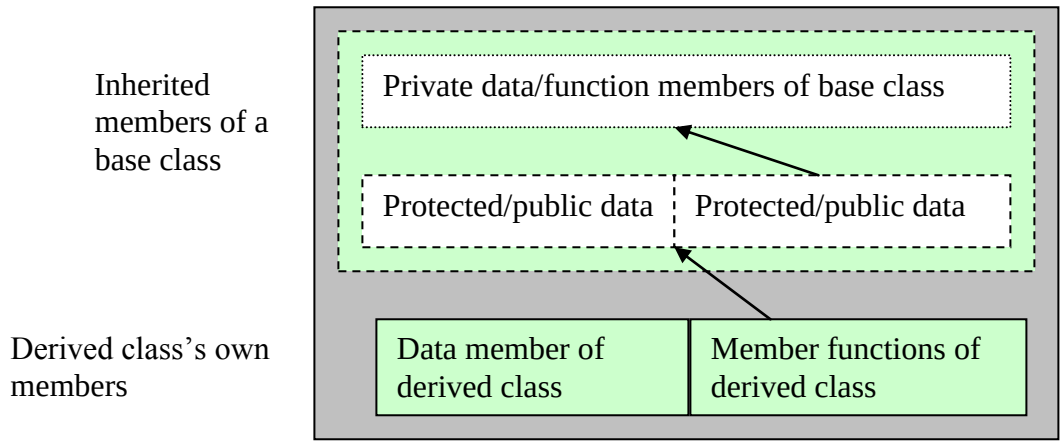
```
Class DerivedClass :  VisibilityMode  BaseClass
{
        statements
}
```

The VisibilityMode is optional. It may be either public or private. By default it is private.

Inheritance of a base class with visibility mode public, by a derived class, causes public members of the base class to become public members of the derived class and the protected members of the base class become protected members of the derived class. Object of the derived class can access public members of the base class that are inherited as public using the dot operator. But the protected members cannot be accessed.

Inheritance of a base class with visibility mode private by a derived class, causes public members of the base class to become private members of the derived class and the protected members of the base class become private members of the derived class. Since all the members of the base class become private in derived class so object of the derived class cannot access any of these members.

Base class visibility	Derived class visibility	
	Public derivation	Private derivation
private	Not inherited	Not inherited
protected	protected	Private
Public	Public	private



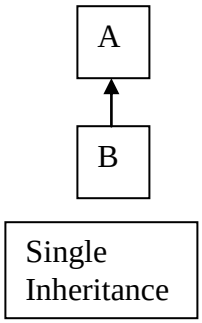
Advantages of Inheritance: Inheritance offers the following advantages --

1. Developement model closer to real life object model with hierarchical relationships
2. Reusability -- facility to use public methods of base class without rewriting the same
3. Extensibility -- extending the base class logic as per business logic of the derived class

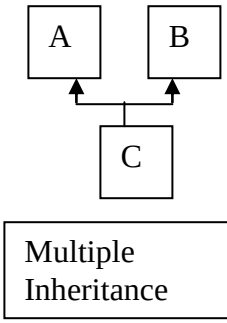
4. Data hiding -- base class can decide to keep some data private so that it cannot be altered by the derived class

Type of Inheritance: -

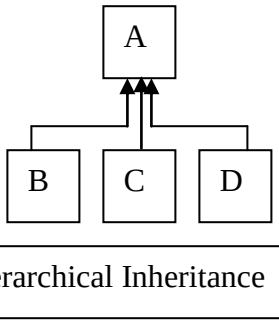
1. Single Inheritance: - Derivation of a class from only one base class is called single inheritance.
2. Multiple Inheritance: - Derivation of a class from several base classes is called multiple inheritance.
3. Hierarchical Inheritance: - Derivation of several classes from a single base class is called 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.
6. Multipath Inheritance: - Derivation of a class from other derived classes, which are derived from the same base class is called multipath inheritance



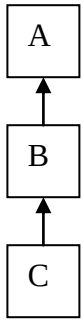
Single Inheritance



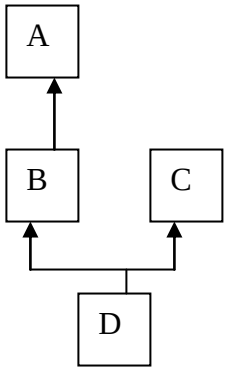
Multiple Inheritance



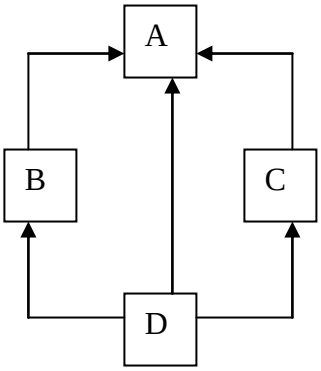
Hierarchical Inheritance



Multilevel Inheritance



Hybrid Inheritance



Multipath Inheritance

Constructors in Derived classes: - The derived class need not have a constructor as long as the base class has a no-argument constructor. However, if the base class has constructors with arguments then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructor. When an object of a derived class is created, the constructor of the base class is executed first and later the constructor of the derived class.

Pointers: -

New Operator: - The new operator offers dynamic storage allocation. The syntax is

Datatype * new datatype[size]

For example: int *a

A = new int[100];

Delete Operator: - This operator is used to return the memory allocated by the new operator back to the memory pool. The syntax is delete pointervariable.

Null pointer: -

Constant pointer: - The pointer which content can not be modified is called constant pointer.

Datatype * const pointervariablename;

Pointer to constant: - The pointer which hold the address of a constant is called pointer to constant.

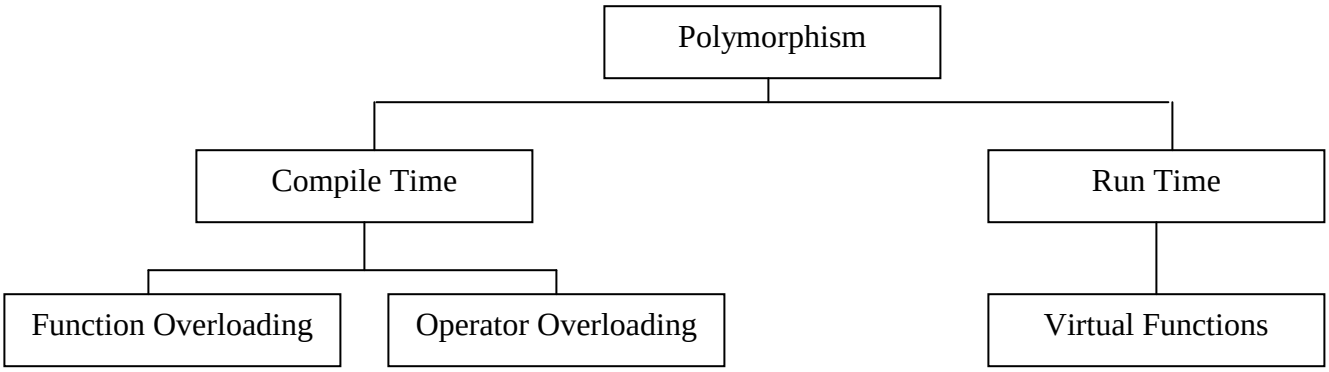
Datatype const * pointervariablename;

Pointers to object and pointers to pointer: -

Virtual function & polymorphism: -

Pointer to derived objects: - Pointer can be used with the objects of base classes or derived classes. Pointer to object of a base class is type-compatible with pointers to objects of a derived class. Thus a base class pointer may address an object of its own class or an object of any class derived from the base class. But the base class pointer can not access all the member of a derived class. Base class pointer can only access those members of the derive class which are inherited from the base class. It can not access those members which are declared inside the derived class.

Virtual function: -



Definition: - In C++, the polymorphism indicates the form of a member function that can be changed at runtime. Such member functions are called virtual functions. Virtual functions allow programmers to declare functions in a base class, which can be defined in each derived class. C++ provides a solution to invoke the exact version of the member function, which has to be decided at runtime using virtual functions. The keyword virtual is used to declare a virtual function.

Features of virtual function: -

a) The virtual functions must be members of some class
b) They cannot be static members
c) They are accessed by using object pointers
d) A virtual function can be a friend of another class
e) A virtual function I a base class must be defined
f) We cannot have virtual constructors, but have virtual destructors.

Early binding and late binding: - Polymorphism can be implemented in two different ways – compile time and run time. In compile time polymorphism, overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time. This is called *early binding* or *static binding* of *static linking*. At run time, when it is known what class objects are under considerations, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as *late binding* or *dynamic binding.*

Pure virtual function: - A pure virtual function is a function declared in a base class that has no definition relative to the base class. It is a null body function. The declaration syntax is

Virtual return type function Name(argument list) = 0;

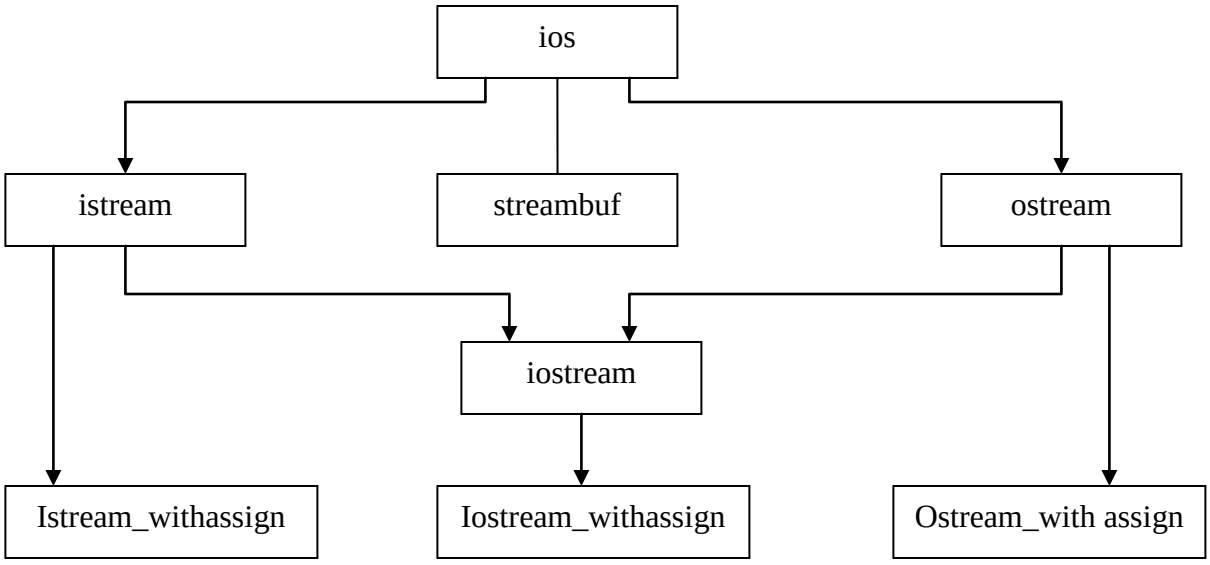
A class containing pure virtual functions cannot be used to declare any object of its own. Such a base class is called abstract base class.

Streams

Stream: - A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called the output stream.

The data in the input stream can come from the keyboard or any other storage device. The data in the output stream can go to the screen or any other storage device. C++ contains several pre-defined streams that are automatically opened when a program begins its execution.

Stream Class: - 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. These classes are called stream classes. These classes are declared in the header file iostream.



The class ios provides the basic support for formatted and unformatted I/O operatins. The class istream provides the facilities for formatted and unformatted input. The class ostream provides the facility for formatted output. The class iostream provides the facilities for handling both input and output streams. Three class namely, istream_with assign, ostream_withassign, and iostream_withassign add assignment operators to these classes.

Header files

IOS flags: -

1. ios::left	left justified output
2. ios::right	right justified output
3. ios:: scientific	scientific notation
4. ios:: fixed	fixed point notation
5. ios:: doc	decimal base
6. ios:: oct	octal base
7. ios:: hex	hexadecimal base

String

File: - A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files. Different file stream classes are

1. fstreambase: - It provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream class. It contains open() and close() function.
2. ifstream: - It provides input operations. It contains open(), get(), getline(), read(), seekg() and tellg() function
3. ofstream: - It provides output operations. It contains open(), put(), seekp(), tellp() and write() function.
4. fstream: - It provides simultaneous input and output operations. It contains open() and all the functions from istream and ostream.

Open() function: - open() function can be used to open multiple files that use the same stream object.

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

The second argument mode specifies the purpose for which the file is opened. It may contain the following values:

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::out	open file for writing only
ios::trunc	delete the content of the file if it exists

eof() function: - It is used to check the end of the file. It returns a non-zero value if the end of file is encountered, and a zero otherwise. *Fileobject.eof()*

seekg() function: - Move the input file pointer to a specified location. *Seekg(offset, refposition);*

the parameter offset represents the number of bytes the file pointer is to be moved from the loation specified by the parameter refpositin. The refposition takes one of the following three constants

ios::beg	start of the file
ios::curr	current position of the pointer
ios::end	end of the file

seekp() function:- Moves put output file pointer to a specified location. *Syntax is same with seekg*

tellg() function: - Gives the current position of the input file pointer.

put() function: - This function write a single character to the associated stream.

Syntax is fileobject.put(character)

get() function: - This function can read a character from the associated stream.

Syntax is fileobject.get(character variable)

Template

Template in C++ is used to define genetic classes and functions and thus support for generic programming. It allows the construction of a family of template functions and classes to perform the same operation on different data types.

Class template: - The template declared for classes are called class templates. Syntax is

```
Template < class T >
class classname
{
        T DataMeberName;
        -------
        -------

}
```

The prefix statement template < class T > tells the compiler that we are going to declare a template and use T as a type name in the declaration. Now we can create object holding different data types as

classname <data type> objectname;

We can use more than one generic data type in a class template. They are declared as a comma-separated list within the template specification as follows

```
Template < class T1, class T2 >
class classname
{
        T1 DataMeberName;
        T2 DataMeberName;
        -------
        -------

}
```

Function template: - The template declared for functions are called function templates. The syntax is

```
template < class T >
returntype functionname (arguments of type T)
{
        function body
}
```

A function generated from a function template is called a *template function*. The T is called the generic data type which representation is not known in the declaration of the function template. It is known only at the point of a call to a function template. We can use more than one generic data type in the function template.

```
template < class T1, class T2 >
returntype functionname (arguments of type T)
{
        function body
}
```

Smart Pointer: - Overloading -> operator is primarily useful for creating “smart pointer”, that is , objects that act like pointers and in addition perform some action whenever an object is accessed through them.

Void pointer: - A pointer to any type of object can be assigned to a variable to type void*. A void* can be assigned to another void*. A void* can be compared for equality and inequality and can be explicitly converted to another type. To use a void*, we must explicitly convert it to a pointer to a specific type

Exception Handling