Overloading Functions and Operators

Function Overloading

A base class member function can be inherited in the derived class, however, sometimes it may need to be redefined according to the requirement in the derived class. This redefining of the base class member function in the derived class with the same name is called ***function overloading***. Function overriding implies member function in a derived class has the same name at that of base class member function, however, has a different implementation.

Operator Overloading

One of the key feature of 'C++' is that the objects of a class can be treated like the variables of built in data types. That is, C++ permits to perform all the arithmetic and logical operations on the objects of a class in the same way as performed on simple variables. To perform operations on simple variables some built in operators such as + , - , *, /, >, <, =, == etc. are provided. To use these operators with the objects of a class, C++ provides a way to give an additional meaning to those operators. The way of giving an additional meaning to these operators is known as ***operator overloading.***

Operator overloading is the process that enables an operator to exhibit different behavior, depending on the data being provided. It enables to change the functionality of the existing operators so that they can be used with classes.

Almost all the unary, binary and some special operators can be overloaded in C++.
The list of operators that can be overloaded.

Type	Operator
Unary	! (logical negation), & (address-of), * (pointer dereference), + (unary plus), - (unary minus), ++ (increment) and -- (decrement)
Binary	Arithmetic (+, -, *, /, %), logical (&&, ||), relational (<, >, <=, >=, ~=, ==), shorthand (+=, -=, %=, /=)
Special	<<, >>, (), =, {}, new, delete, new[], delete[]

Some operators that cannot be overloaded are listed here:
1) Dot operator (.)
2) Dereference pointer to class members (.*)
3) Scope resolution operator (: :)
4) Conditional operator (? :)
5) sizeof operator
6) pre-processor symbol (#)

An operator is overloaded with the help of special function called an *operator function.* An operator function can be defined either as a public member function of the class or an as a friend function. to overload an operator, these steps are to be followed.
1. Create the class for which an operator is to be overloaded.
2. Declare the operator function either as a public member function of the class or as a friend function of the same class.
3. Define the operator function either inside or outside the class definition.

Rules for overloading operators

1) The implementation of the operator can be changed, however, not the syntax for using the operator.
2) The precedence and the associatively of an operator cannot be changed.
3) The number of operands required with the operator cannot be changed.
4) New operators cannot be created, however, new definition for the existing operator can be created.
5) Overloaded operators except the function call operator '( )' cannot have default arguments.
6) All overloaded operators except the assignment operator '=' can be inherited by the derived classes.
7) There is no automatic "composition" of operators. That is, overloading '=' and '+' operators does not imply that the '+=' operator has been overloaded automatically it has to be overloaded explicitly.
8) The operators ( ), [ ], -> and = can be overloaded only as member functions and not as friend functions.

Overloading unary Operator

When unary operator are overloaded using member functions, the member operator function does not accept any argument. That is, parameter_list is empty for unary operators. This is because operand for the unary operator, that is, the object that invokes the member operator function is passed implicitly to the function.

A program to demonstrate the concept of overloading unary operator (increment and decrement)

```
#include<iostream.h>
#include<conio.h>

class complex
{
    int a,b,c;
   public:
       complex(){}
       void getvalue()
      {
              cout<<"Enter the Two Numbers:";
              cin>>a>>b;
      }

   void operator++()
     {
              a=++a;
              b=++b;
      }

      void operator--()
      {
              a=--a;
              b=--b;
       }

       void display()
       {
              cout<<a<<"+\t"<<b<<"i"<<endl;
        }
};

void main()
{
    clrscr();
    complex obj;
    obj.getvalue();
    obj.display();
    obj++;
    cout<<"Increment Complex Number\n";
    obj.display();
    obj--;
    cout<<"Decrement Complex Number\n";
    obj.display();
    getch();
}
```

Overloading Binary operator

Since the binary operator operate on two operands, one operand, that is, the object of the class invoking the function is passed implicitly to the member operator function. The other operand as an argument which can be passed either by value or by reference.

This is a sample program showing the **overloading** of **Assignment operators** in c++.

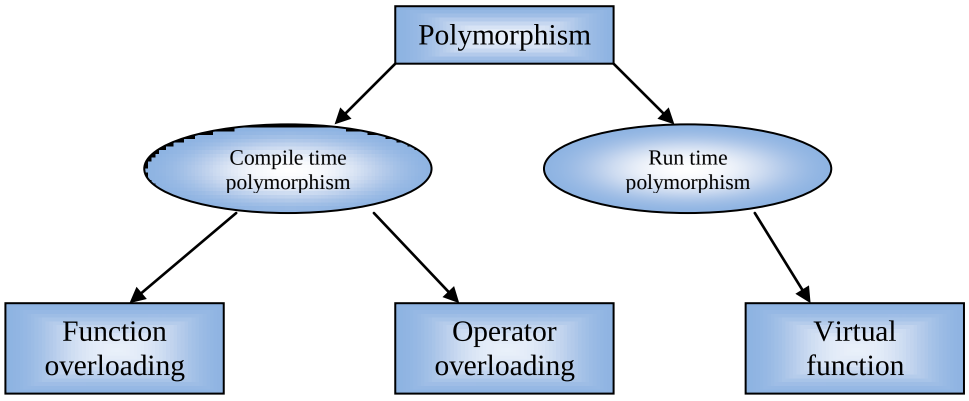
```
#include<iostream.h>
#include<conio.h>
Class sample
        {
                Private:
                Int x, y;
                Public:
Void getdata ()
        {
                Cout <<" \n enter the value of x and y";
                Cin>> x>>y;
        }
Void operator = (sample obj)
        {
                X= obj.x;
                Y= obj.y;
        }
Void display ()
        {
                Cout<<"\n value of x"<<x;
                Cout<<"\n value of y"<<y;
        }
        };
Void main ()
        {
                Sample obj1, obj2;
                Obj1.getdata ();
                Obj1.display ();
                Obj2=obj1;
                Obj2.display();
                getch ();
        }
```

Polymorphism and Virtual Functions

Polymorphism

Polymorphism is one of the crucial features of OOP. It simply means 'one name, multiple forms', we have already seen how the concept of polymorphism is implemented using the overloaded functions and operators. The 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 and therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called ***early binding or static binding or static linking.*** Also known as ***compile time polymorphism,*** early binding simply means that an object is bound to its function call at compile time.

It would be nice if the appropriate member function could be selected while the program is running. This is known as ***run time polymorphism.*** How could it happen? C++ supports a mechanism known as ***virtual function*** to achieve run time polymorphism.



At run time, when it is known class objects are under consideration, 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***. It is also known as ***dynamic binding*** because the selection of the appropriate function is done dynamically at run time. Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects.

Virtual Function

A virtual function is a member function declared inside the base class and whose signature. When the base class containing virtual function is inherited, the derived classes may implement their own versions of that function. the entire function of the base class can be replaced by a set of new implementation in the derived class

A member function can be made virtual by prefixing its declaration with the keyword ***virtual*** in the base class .

For example,

```
class base
{
        public :
                virtual void display ( )            //virtual function
                {
                 cout<<"Base class";
                }
};
```

In this class definitions, the function *display ( )* of the class *base* is declared *virtual.* Thus, the display ( ) function can be redefined in the derived classes of the class *base.*

Whenever a virtual function is inherited, its virtual nature is also inherited into all its derived classes and it is not required to use the keyword *virtual* in the subsequent derived classes.

For example,

```
class derived : public base
{
        public:
                void display ( )            //overriding virtual function
                {
                        cout <<"Derived class";
                }
};
```

In addition, when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden. This implies that a virtual function always remains virtual irrespective of the number of times it is inherited.

Rules for virtual functions

1) A virtual function must be defined as a public member of the class.
2) A virtual function is accessed by using an object pointer or object reference.
3) The base class virtual function and its redefined version in the derived classes must have the same prototype.
4) A virtual function cannot be global or static.
5) The constructors of a base class cannot be made virtual since at the time the constructor is invoked, the virtual table is not available in the memory.
6) One incrementing or decrementing the pointer of base type, it always point to the next or the previous object respectively, of its base class type, irrespective of the contents of the pointer.

Object Slicing

A virtual function should always be accessed either using an object pointer or reference. In case an attempt is made to do so using an object, a phenomenon known as ***object slicing*** takes place.

Pure Virtual Function

A virtual function that has no definition within the base class is known as ***pure virtual***

function (also called do-nothing functions). A virtual function can be made pure by appending the pure specifier "=0" to its declaration in the base class.
The syntax,

virtual return_type function_name (parameter_list)=0;

Note that if a virtual function is declared as pure, it must be redefined in all of its derived classes otherwise a program error occurs.

Abstract Classes

A class that contains at least one pure virtual function is known as an ***abstract class or abstract base class.*** An abstract class is different from a polymorphic class. the objects of an abstract class cannot be created, as at least one of its members lacks implementation. This implies that abstract classes can only be used to act as a base class for other classes and not for instantiation. An abstract class can be derived from a non-abstract class by overriding its non-pure virtual function with a pure virtual function.

Virtual Destructors

A destructor of the derived class is called before the destructor of the base class. However, an exception arises in destructors when the pointer to the base class is made to point to the object of derived class and memory is allocated to the object using the new operator. Now if the memory is de-allocated using the delete operator, only the base class destructor is called and the derived class destructor is not called at all as the pointer is of base class type. This results in memory leak, which causes the loss of memory access due to the wrong destructor being invoked

A destructor can be made virtual by prefixing its declaration with the keyword *virtual* in the base class. Note that the difference between a virtual member function and a virtual destructor is that of the function is redefined in the derived class then in the former case, only the derived class version of the function is called, whereas, in the latter case, both the derived and base class versions of the destructor are called.

Templates and Exception Handling

Code reusability is one of the main advantages of object oriented programming. One way to achieve code reusability is to write and compile the code and call it any number of times in a program. For example, a sum() function defined to add two integers can be called any number of times with different integer values. This provides a way to re-use an object code. In such situations, where the same operation is to be performed on different data types, templates can be used.

Templates enable us to define a generic function or a generic class that operates on various data types, instead of creating a separate function or class for each type. Templates are also known as generics or parameterized types. The C++ template mechanism allows a generic type to be passed as parameter in the definition ofa function or a class so that they can work with different types of data. Such approach of writing programs is known as generic programming.

When templates are used with functions they are known as ***function templates*** and when they are used with classes, they are known as ***class templates.***

Function Templates

While writing overloaded functions, there may be situation when the same code has to be written repeatedly for different types of data. For example, to add two integer and floating point numbers, two separate functions have to be created for each of the data type.
For example,
Overloaded functions to add two numbers of int and float types.

```
int sum (int x, int y)
{
        return x + y;
}

float sum(float x, float y)
{
        return x + y;
}
```

A function template, also known as generic function, defines a group or a family of functions having the same functionality but working on different data types. While using function templates, only one function signature needs to be defined and the compiler automatically generates the required for handling the individual data types.

Defining a function template

The definition of templates is similar to the definition of ordinary functions. The only difference is that it is prefixed by the statement *template <class T>* which is known as ***template statement***. It informs the compiler that a template is being declared. When a template is defined, it does not actually define a function rather it provides a blueprint for generating many functions. compiler uses this blueprint to automatically generate a new member of the family of functions when it is required first.

Syntax,

```
template<class T>                //template statement
return_type function_name (parameter_list)
{
        // function body
}
```

The letter T in template definition is called ***template argument or type parameter.*** It is a placeholder name for the data type used by the function.

Like ordinary functions, a function template can be defined either before or after main( ).

To declaring and defining a function template consider these statement.

```
template <class T>               //function template prototype
T sum( T x, T y);

template <class T>               //function template definition
T sum (T x, T y)
{
        return x + y;
}
```

Calling function templates

When the function template is called with the actual data type, the compiler generates a specific version of the function template for that data type. A function generated from a function template is known as a ***template function*** (also known as generated or instantiated function) and the process is known as ***function template instantiation.*** The definition created from a template instantiation is called ***specialization.***

When the function template is called, the data type T is dynamically determined by the compiler according to the parameter passed and replaced with actual data type.

For example,

```
template<class T>
T sum( T x, T y)
{
        return x + y;
}

main ( )
{
        //calling sum() with int data type
        cout << "sum(25,75) = " <<sum(25,75)<<endl;
        //calling sum() with float data type
        cout << "sum(25.15, 75.12) = " <<sum(25.15, 75.12)<<endl;
}
```

Class Templates

A Class template also known as a generic class, defines a group or a family or classes having the same class definition but handling different type of data. While using class templates, only one class definition needs to be created and the compiler automatically generates the required classes for handling the individual data types.

A class template is used in those situation where two or more classes have the same class definition but handle different data types.

Defining a class templates

A class template must also be defined before creating its objects. The definition of a class template is similar to an ordinary class definition, except that it is prefixed by the statement ***template <class T>*** . The syntax,

```
Template <class T>
class class_name
{
        // data members and member functions of the class
};
```

For example,

```
template <class T>
class calculator
{
        T a, b;
        public :
                void getdata(T x, T y)
                {
                        a = x;
                        b = y;
                }
        T add( )
                { return a+b; }
        T diff ( )
                { return a-b; }
        T prod ( )
                { return a*b; }
        T div ( )
                { return a/b; }
        void display ( )
        {
                cout<<endl<<" Numbers Are:  "<< a<<" and "<<b<<endl;
                cout<<"The Sum is = "<<add( )<<endl;
                cout<<"The Difference  is = "<<diff( )<<endl;
                cout<<"The Product is = "<<prod( )<<endl;
                cout<<"The Quotient is = "<<div( )<<endl;
        }
};
```

Creating instance of class template

After defining the class template, the next step is to create the required classes by providing the actual type for the type parameter. This process is known as ***class template instantiation.*** A class generated from a class template is known as a template class also known as generated or instantiated class.

The syntax,

classtemplate_name <type> object_name (argument_list)

where,

classtemplate_name = the name of class template
type = any C++ data type
object_name = name of the object

for example,

```
calculator<int>c1;
calculator<float>c2;
```

```
int main()
{
        calculator<int>c1;               //instantiating class template with int data type
        calculator<float>c2              //instantiating class template with float data type
        c1.getdata(10,25)l
        c1.display();
        cout<<endl;
        c2.getdata(5.15, 15.20)
        c2.display();
}
```

Exception Handling

An exception is an unexpected event that occurs during the execution of a program and alters the normal flow of the program or terminates the program abnormally. In simple terms, it can be defined as run-time anomaly. The errors that can cause exceptions can range from simple programming errors, such as opening an invalid file for reading, invalid array index, etc. to serious errors such as running out of memory.

Exceptions can be classified into two types, namely, ***synchronous exceptions and asynchronous exceptions.*** The exceptions that occur at specific program statements are called ***synchronous exceptions***. For example, deleting from an empty list, invalid array index, stack overflow, etc. are synchronous exceptions. On the other hand, the exceptions that are issued by external sources which are beyond the control of the program are called ***asynchronous exception.*** For example, hardware interrupts such as pressing CTRL – C to interrupt a program is an asynchrony ous exception.

Conventional error-handling mechanism

In the conventional method of handling run-time errors, the error codes are returned from the function in which the error has occurred and are passed to the calling function. The calling function checks for the return values and then appropriately handles the error.

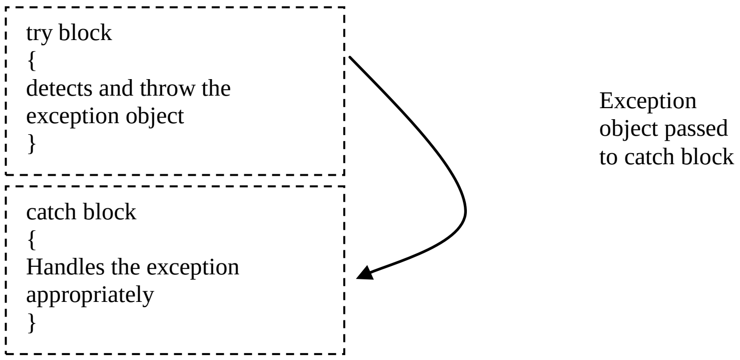
This method has few limitations:

1) The function that checks the error has to be called explicitly. Thus, it is difficult to handle the errors that can occur in a class constructor as the constructor is called implicitly. Hence, this method is not suitable for classes.
2) It does not provide a way to separate the error handling code from the code written to handle the other tasks of the program.

C++ Exception Handling Mechanism

To overcome the limitations of conventional error-handling mechanism, an exception handling mechanism was introduced in C++ in 1989. C++ exception handling mechanism not only guarantees the detection and handling of run-time errors but also provides a way to separate the error-handling code from the rest of the program.

C++ exception handling mechanism uses three keywords ***try, throw*** and ***catch.*** A set of statements that need to be monitored for the exceptions is contained in the **try block.** The try block is surrounded by braces and prefixed by the keyword *try.* Whenever an exception occurs within the try block, it is thrown using the **throw statement.** This passes the control to the **catch block,** which handles the appropriately. The throw statement and the catch block are prefixed by the keyword ***throw*** and ***catch,*** respectively.



The catch block is also known as ***exception handler*** and it must immediately follow the try block that throw an exception.
The syntax,

```
    try                               // try block
    {
        …
        throw statement;              // throw statement
    }                                 // end of try block
    catch (data_type parameter)       // catch block
    {
        …
    }                                 // end of catch block
```

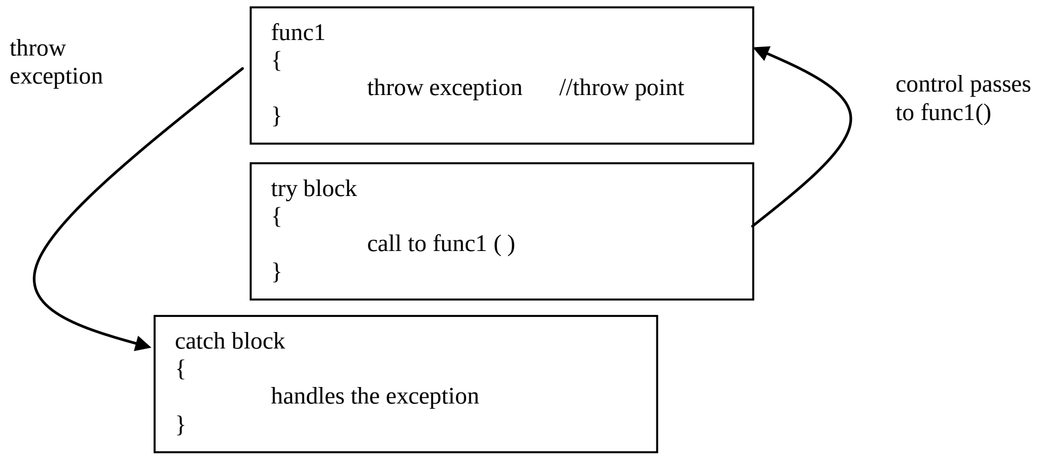
Throwing Mechanism

Whenever an exception occurs it is thrown using the throw statement. The throw statement can be written in any of these forms.

```
    throw (exception);
    throw exception;
    throw;
```

Throwing exception from within a function

It is not always necessary that the exception is thrown from within the try block. It can also be thrown by a function defined outside the try block. However, the function throwing the exception must be called within the function at which the thrown statement is executed is called the ***throw point.*** Once an exception is thrown to the catch block, the control cannot return to the throw point. That is, any statement after the throw statement will not be executed.



For example,

```
void square_root (int x)
{
    try
    {
        if ( x >= 0)
            cout<<"Square root of the number is  = "<<sqrt(x);
        else
            throw(x);
    }
    catch (int i)
    {
        cout<<endl<<"Exception negative number";
    }
}
```

```
main ( )
{
        square_root (5);
        square_root (-5);
        square_root (-6);
}
```

Catching Mechanism

When a exception is thrown, it needs to be handled appropriately. The code that handles the exception is written in the catch block. As started, catch accepts an argument of any built in or user defined data type.

Exception Handling and Constructor and Destructors

Sometimes a situation may arise that the exceptions, which occur while creating the objects of a class, need to be handled. Since constructors cannot return a value, an exception must be thrown inside the constructor of the class. If an exception is thrown from within a constructor, then the object being constructed will not be constructed properly. Therefore, the destructor for the object will never be called. However, the destructors for completely constructed objects will be called.

To effectively handle exception in constructors, a new entity called an ***exception class*** need to be created. The throw statement, in this case, throws an object of the exception class, which is handled by the catch block accepting an object exception class.

An exception class can be defined either globally or in the public section of another class. moreover, the body of the exception class can be either empty or it can contain data members and member functions.

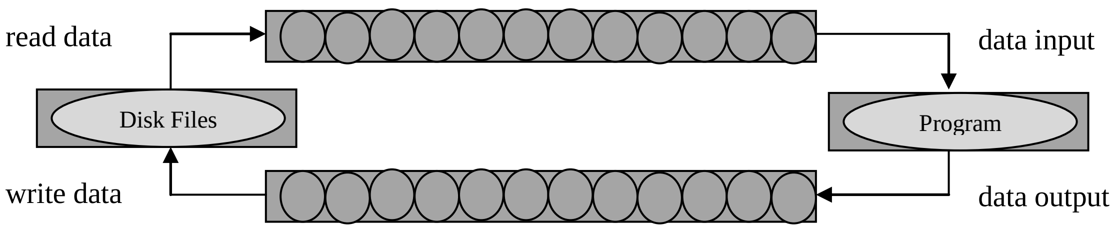
DATA FILE OPERATIONS

A file is a collection of related data stored on some storage device. The data stored in a file is permanent and can easily be accessed as and when required. Moreover, the data stored in a file by one program can be accessed or modified by various programs as per their requirements.

File Streams

A file stream refers to the flow of data between a program and the files. Depending on the flow of data either from file or to file, stream can be classified into two types.

1) **Input stream:** It reads the data from the file and supplies it to the program.
2) **Output stream :** It receives data from the program and writes it to the file.



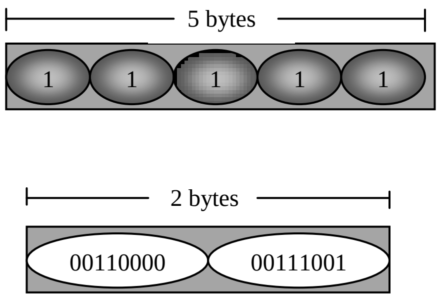
There are two modes by which data can be transferred to and from streams. Text mode and binary mode. A file which is opened in text mode is known as ***text file,*** whereas a file opened in binary mode is known as ***binary file.*** the text file and binary file are differ from each other

1) Storage of numbers: A text file stores data as a sequence of characters while the binary file stores data as sequence of bytes.

2) Handling newline character: In text files, some character translations take place while data is being read from or written to the file. While in binary files, no such character translation takes place. Thus, the number of bytes read or written is same as that in the file.

3) Representation of end of file: In text files, the end of file is represented by a special character, whose ASCII value is 26. While in binary files, there is no such special character to detect the end of file, the end of file is determined by keeping track of the number of character present in the file.

Representation of a number in character and binary format



File Stream Classes

The classes specific to the disk file I/O operations are known as ***file stream classes*** and are declared in the header <fstream>. The header <fstream> defines several classes, including *ifstream, ofstream,* which are used for working with files. These classes are derived from *istream, ostream and iostream* respectively which in turn are derived from ios class. The file specific classes are also derived from *fstreambase* class.

The description of all file specific classes are:

1) **ifstream:** The ifstream is an input file stream class which provides functions for performing reading operations only. It contains functions such as *get ( ), getline( ), read ( ) , seekg ( ) and tellg ( )* which are derived from istream class.

2) **ofstream:** the ofstream is an output file stream class which provides functions for performing writing operations only. It contains functions such as *put ( ) , write ( ), seekp ( ), and tellp ( )* which are derived from ostream class.

3) **fstream:** the fstream is an I/O file stream class and hence, provides functions fro performing the input as well as output operations. It contains all the functions of istream and ostream classes which are inherited from iostream class.

4) **filebuf:** The filebuf class is used to manage buffered I / O of file stream. It contains open() and close() functions.

5) **fstreambase:** The fstreambase class serves as a base class for ifstream, ofstream and fstream classes. it contains open() and close() functions and also other operations which are common to all file streams.

Opening and Closing a file

To perform any operation on a file, it needs to be opened first. A file is opened by linking it to a stream. Thus for opening a file, first an object of the particular stream class is created and then associated with the file. the stream class to be used for creating the stream object depends upon the type of operations to be performed on the file.

Opening a file using a constructor.

When a file is opened using the constructor of a stream class, the file name is passed to it as an argument. then the constructor initializes the stream object with the file name passed to it.

Syntax,

stream_class stream_object ("file_name")

where,

stream_class	= the name of the stream class whose object is to be created.
stream_object	= the object name which can be any valid identifier
file_name	= the name of the file to be opened.

For example, a file named "marks" can be opened for input, output and both using constructor

```
ifstream infile("marks");          //for input only
ifstream outfile("marks")          // for output only
ftream iofile("marks")             // for both input and output
```

Opening a file using the open() function

The function open() which is a member function of fstreambase class has the same parameter as the constructor of stream class. however unlike constructor, the creation of stream object and its linking with a particular file using open() function are performed in separated statements.

Syntax for open() function.

stream_class stream_object;
stream_object.open("file_name");

For example

```
ifstream infile;                    //create input stream
infile.open("marks");               // connect stream to file marks

ofstream outfile;                   // create output stream
outfile.open("marks");              // connect stream to file marks

fstream iofile;                     //create input/output stream
iofile.open("marks");               // connect stream to file marks
```

Specifying file modes

The file has been opened so far by specifying only one argument in both the constructor and open() function. however, another argument can also be included in both the functions that specifies the file mode. The file mode describes how a file is to be used, that is, to read from it, to write to it, to append to it, and so on.

The general form of open() function with two arguments:

```
stream_object.open("file_name", file_mode);
```

The various modes that a file can have

Mode	Description
ios::in	Opens a file for reading
ios::out	Opens a file for writing
ios::trunc	Delete the contents of the file if it already exists, that is, truncates the file to zero length.
ios::app	Appends the data at the end of the file
ios::ate	Moves to the end of file on opening, however permits addition as well as modification of data anywhere in the file
ios::binary	Opens the file in binary mode
ios::nocreate	Opens fails if the file does not exist does not create a new file
ios::noreplace	Open fails if the file already exists does not replace the existing file with new one

Points must be kept in mind while opening a file

1) A file name can include a path specifier.
2) Opening a file in the ios::out mode also opens it in the ios::trunk mode
3) Both ios::app and ios::ate modes, a file is created by the specified name, if it does not exist.
4) The mode ios::app mode can be used with the files capable for output only.
5) By default, all the files are opened in text mode. Hence, to open a file in binary mode, the file mode ios::binary must be specified as shown here,

```
ofile.open("account",ios::app::binary);
```

Closing a file

When a stream object goes out of scope, the destructor of the object is invoked implicitly, which closes the associated file automatically. However, a file can also be closed explicitly by using the function close() which is a member function of the fstreambase class. the close () function takes no parameter and returns no value.

The syntax,

```
stream_object.close();
```

For example,

```
iofile.close();          //close connection
```

Reading and Writing files

Once a file is opened, it can be used for reading or writing text as well as raw data through various I/O operations provided by stream classes. as the stream classes are device independent, the same I/O operations formatted as well as unformatted I/O used with console I/O can be used with files.

1) Using << and >> operators

The insertion operator (<<) and extraction operator (>>) have been used with the standard stream objects cout and cin, respectively. However, the operators << and >> are overloaded in the classes ostream and istream, respectively to work in the same way with file stream objects. the only difference is that instead of using cout and cin a stream object linked to a file is used.

2) Using put() and get()

The functions put() and get() which are the member functions of classes ostream and istream, respectively, are used to write and read a single character at a time from the linked stream. The get() function has many forms, however, the commonly used form along with the form of put() function.

```
istream &get(char &ch);
ostream &put(char ch);
```

The get() function reads a single character form the associated stream and puts that value in ch and the put() function writes the value of ch to the stream. Both the functions return a reference to the stream.

3) Using getline()

The function getline() which is a member of the input stream class, reads multiple character form the associated stream. the getline() function is overloaded in several forms. However, the prototype of the commonly used form is given below,

```
ifstream &getline( char *buf, streamsize num, char delim);
```

4) Using read() and write()

The function read() and write(), also known as binary I/O functions are used to handle block of binary data unlike <<, >>, get() and put(), the functions read() and write() operate on binary values and not on the text, that is, they treat data as a sequence of bytes, both the function can read and write an entire structure in one I/O operation. Hence, the data members are not required to be read from or written to separately.

```
istream &read(char *buf, streamsize num);
ostream &write(char *buf, streamsize num);
```

Detecting End of File

While reading data from the file, the end position of the file is required to be detected so that any further attempt to read data from the file can be prevented. When the end of file is encountered, a signal *end-of-file (EOF)* is sent to the program by the operating system. This signal can be detected by using the function ***eof()*** which is a member function of ios class. the use of function eof() is given,

```
while (!filestream.eof())        //until eof reached
{
        //process the file
}
```

Here, the function eof() returns zero as long as the end-of-file is not reached. As soon as the end-of-file is reached, it evaluates to a non-zero value and the loop terminates.

File pointers

In C++ every file is associated with two file pointers, namely get pointer and put pointer. the get pointer specifies the position in file from where the next read operation will start and the put pointer specifies the position in file form where the next write operation will start, when the file is opened in input mode, the get pointer is placed at the beginning of the file.

Specifying the position

The manipulation of the pointer can be accomplished through the function: tellg(), tellp(), seekg() and seekp(), which are provided by the file stream classes.

Function	Prototype	Purpose
tellg()	pos_type tellg()	Returns the current position of the get pointer
tellp()	pos_type tellp()	Returns the current position of the put pointer
seekg()	ifstream &seekg(pos_type pos)	Moves the get pointer to the specified location pos from the beginning of the file where pos is a positive integer value
seekp()	ofstream &seekp(pos_type pos)	Moves the put pointer to the specified location pos from the beginning of the file where pos is a positive integer value.

For example,

```
ofstream outfile("myfile", ios::app);
int current_pos = outfile.tellp();
```

the first statement moves the put pointer to the end of the file as myfile is opened in append mode and the second statement returns the number of bytes in the file.

```
ifstream infile("myfile");
infile.seekg(5);
```

the second statement sets the get pointer at the byte number 5, that is the 5th byte in the file. note that the bytes in a file are numbered form zero.

Randomly Accessing a file

Randomly accessing a file means directly reaching the desired record / object in the file. to access an object in a file directly, the put pointer is placed at the beginning of the object by skipping (objet_no-1) *object_size number of bytes from the beginning of the file. once the put pointer is placed at the appropriate object number, the object can be modified. Note that a object can be accessed directly only if the file consists of similar objects, that is, object of equal size. The size of each object can be determined using the sizeof operator.

```
int object_size=sizeof(object);
int position = (n-1) *object_size;
```

where, position provides the byte number of the first byte of the nth object and the file poiner can be moved to this position by using function seekg() or seekp().