Object Oriented Programming with C++(BCS306B)

Module – 5

Chapter 1 :

Exception Handling:

SYLLABUS:

Exception Handling: Exception Handling Fundamentals, Handling Derived-Class Exceptions, Exception Handling Options, Applying Exception Handling. The C++ I/O System Basics: C++ Streams, The C++ Classes, Formatted I/O.
File I/O: <fstream> and File Classes, Opening and Closing a File, Reading and Writing Text Files, Detecting EOF.

5.1 Exception Handling Fundamentals

- An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.
- Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.
- **throw** – A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try** – A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

The **try** and **catch** keywords come in pairs:

```
try
{
      // Block of code to try throw exception;
     // Throw an exception when a problem arise
}
catch ()
{
  // Block of code to handle errors
}
```

Example :

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

int main() {
  try {
    int age = 20;
    if (age >= 18) {
      cout << "Access granted - you are old enough.";
    } else {
      throw (age);
    }
  }
  catch (int myNum) {
    cout << "Access denied - You must be at least 18
              years old.\n";
    cout << "Age is: " << myNum;
  }
  return 0;
}
```

Output:

Access denied - You must be at least 18 years old.
Age is: 15

Example explained :

- We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.
- In the catch block, we catch the error and do something about it. The catch statement takes a **parameter**: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.
- If no error occurs (e.g. if age is 20 instead of 15, meaning it will be be greater than 18), the catch block is skipped:

5.2 Handling Derived-Class Exceptions

- The basic structure of exception handling in C++ involves using the “try” block to enclose the code that may throw an exception.
- The “catch” block is used to handle the exception that was thrown. The catch block is associated with the try block by following the try block immediately.

```
try
   {
        // code that may throw an exception
   }
        catch (exceptionType e)
   {
        // code to handle the exception
   }
```

- When an exception is thrown inside the try block, the program will immediately exit the current function and search for the nearest enclosing catch block.
- If a matching catch block is found, the program will transfer control to that catch block and execute the code inside it. If no matching catch block is found, the program will terminate.
- Here, we will examine the concept of exception handling and the ways to catch base and derived classes in C++.
- If the base and derived classes that we have defined are caught as exceptions, then the catch block for the derived class will be displayed before the base class in the output terminal.
- The above statement is correct, which can be verified by reversing the order of the base and derived classes, and in that case the catch block for the derived class will not be reached.

```
#include <iostream>
using namespace std;
class Base {};
class Derived : public Base {};

int main()
{
  try
  {
    throw Derived();
  }
  catch (Base& b)
  {
    cout << "Caught base exception" << endl;
  }
  catch (Derived& d)
  {
    cout << "Caught derived exception" << endl;
  }
  return 0;
}
```

Output:

Caught base exception

Thus it is proved that if catch block of the base class is written before the catch block for the derived class then the derived class catch block will never be reached.

5.3 Exception Handling Options

a. *Catching All Exceptions*

Catch block is used to catch all types of exception. The keyword “catch” is used to catch exceptions.

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

void func(int a) {
    try {
        if(a==0) throw 23.33;
        if(a==1) throw 's';
    } catch(...) {
        cout << "Caught Exception!\n";
    }
}
int main() {
    func(0);
    func(1);
    return 0;
}
```

Output

```
Caught Exception!
Caught Exception!
```

b. Restricting Exceptions

- You can **restrict the type of exceptions** that a function can throw outside of itself.
- To accomplish these restrictions, you must add a ***throw clause*** to a function definition

Here is the general **syntax**

```
ret-type func-name(arg list) throw(type list)
{
    //statements
}
```

- Here, only those data types contained in the comma-separated type list may be thrown by the function.
- Throwing any other type of expression will cause abnormal program termination.
- If you don't want a function to be able to throw any exceptions, then use an empty list.

- Attempting to throw an exception that is not supported by a function will cause the standard library function *unexpected*() to be called.
- By default, this causes *abort*() to be called, which causes abnormal program termination. However, you can specify your own unexpected handler.
- The following program shows how to restrict the types of exceptions that can be thrown from a function.

```
// Restricting function throw types.

#include <iostream>
using namespace std;

// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
   if(test==0) throw test; // throw int
   if(test==1) throw 'a'; // throw char
   if(test==2) throw 123.23; // throw double
}

int main()
{
   cout << "start\n";
 try
   {
     Xhandler(0); // also, try passing 1 and 2 to
     Xhandler()
   }
```

```
catch(int i)
    {
      cout << "Caught an integer\n";
    }
    catch(char c)
    {
      cout << "Caught char\n";
    }
    catch(double d)
    {
      cout << "Caught double\n";
    }

    cout << "end";

  return 0;
}
```

- In this program, the function Xhandler() can throw only integer, character, and double exceptions.
- If it attempts to throw any other type of exception, then an abnormal program termination will occur.

The following change to Xhandler() prevents it from throwing any exceptions:

```
// This function can throw NO exceptions!
void Xhandler(int test) throw()
{
   /* The following statements no longer work.  Instead,
      they will cause an abnormal program termination. */
  if(test==0) throw test;
  if(test==1) throw 'a';
  if(test==2) throw 123.23;
}
```

c. C++ Rethrowing an Exception

```
// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;
void Xhandler()
{
  try {
     throw "hello"; // throw a char *
      }
  catch(const char *)
      { // catch a char *
     cout << "Caught char * inside Xhandler\n";
     throw ; // rethrow char * out of function
     }
}
```

when an exception received by catch block is passed to another exception handler then such situation is referred to as rethrowing of exception.
This is done with the help of following statement,
throw;
The above statement does not contain any arguments. This statement throws the exception to next try catch block.

```
int main()
{
    cout << "Start\n";
    try
    {
      Xhandler();
    }
    catch(const char *)
    {
      cout << "Caught char * inside main\n";
    }
    cout << "End";
    return 0;
}
```

d. terminate() and unexpected() :

- **terminate()** and **unexpected()** are called when something goes wrong during the exception handling process. These functions are supplied by the Standard C++ library. Their prototypes are shown here:

```
void terminate( );
void unexpected( );
```

- These functions require the header **<exception>**.
- The **terminate()** function is called whenever the exception handling subsystem fails to find a matching **catch** statement for an exception. It is also called if your program attempts to rethrow an exception when no exception was originally thrown.

5.4 Applying Exception Handling

- An exception is an unexpected event that occurs during program execution.

  For example, divide_by_zero = 7 / 0;
- The above code causes an exception as it is not possible to divide a number by 0.
- The process of handling these types of errors in C++ is known as exception handling.
- In C++, we handle exceptions with the help of the try and catch blocks, along with the throw keyword.

  try - code that may raise an exception

  throw - throws an exception when an error is detected

  catch - code that handles the exception thrown by the throw keyword

  Note: The throw statement is not compulsory, especially if we use standard C++ exceptions.

Syntax for Exception Handling in C++:

The basic syntax for exception handling in C++ is given below:

```
try {

    // code that may raise an exception
    throw argument;
  }

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

- Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by the catch block.
- When an exception occurs, the throw statement throws an exception, which is caught by the catch block.
- The catch block cannot be used without the try block.

Example 1: C++ Exception Handling

```
// program to divide two numbers
// throws an exception when the divisor is 0

#include <iostream>
using namespace std;
int main() {

    double numerator, denominator, divide;

    cout << "Enter numerator: ";
    cin >> numerator;

    cout << "Enter denominator: ";
    cin >> denominator;

    try {

        // throw an exception if denominator is 0
        if (denominator == 0)
            throw 0;
```

```
 // not executed if denominator is 0
 divide = numerator / denominator;
cout << numerator << " / " << denominator << " = " <<
divide << endl;
    }
 catch (int num_exception)
    {
        cout << "Error: Cannot divide by " <<
num_exception << endl;
    }

    return 0;
}
```

Output 1 :
Enter numerator: 72
Enter denominator: 0
Error: Cannot divide by 0

Output 2:
Enter numerator: 72
Enter denominator: 3
72 / 3 = 24

Module – 5
Chapter 2 :

The C++ I/O System Basics:

5.5 C++ Streams

- In **C++ stream** refers to the stream of characters that are transferred between the program thread and i/o.
- *Stream classes* in C++ are used to input and output operations on files and io devices. These classes have specific features and to handle input and output of the program.

Basic Input / Output in C++

Input Stream: If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.

Output Stream: If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.

Header files available in C++ for Input/Output operations are:

iostream: iostream stands for standard input-output stream. This header file contains definitions to objects like cin, cout, cerr etc.

iomanip: iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision etc.

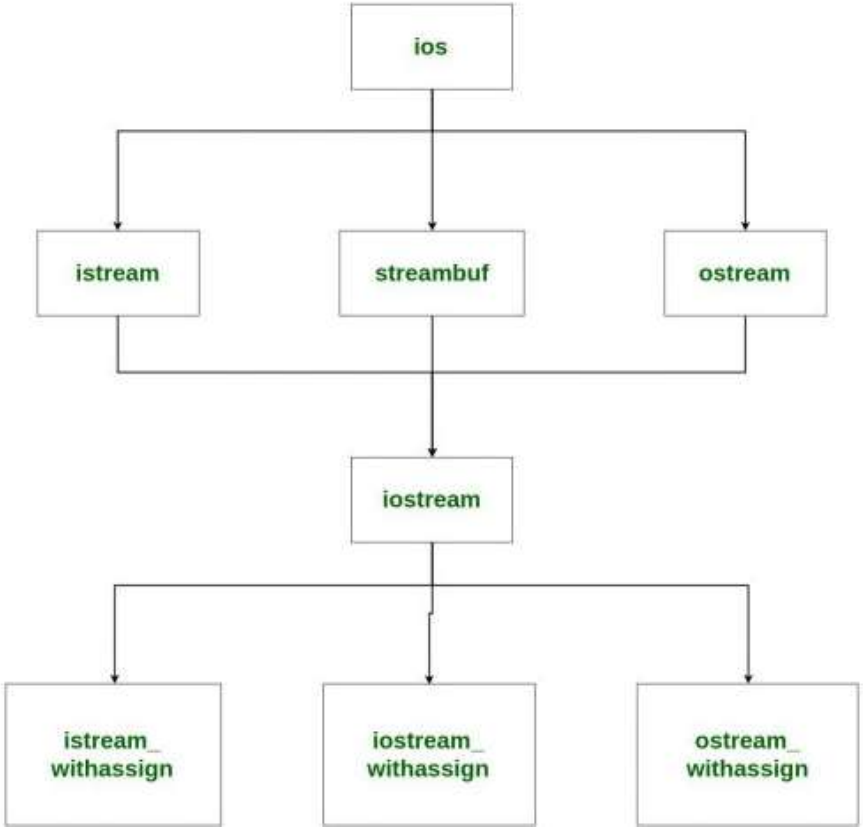
fstream: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

5.6 The C++ Classes

Various stream classes in C++ are as follows:

1. Istream class
2. Ostream class
3. iostream class
4. Ios class
5. istream_withassign Class
6. ostream_withassign Class

Heirarchy of Stream Classess in iostream.h



- The C++ stream class hierarchy consists of a number of classes that define and provide different flows for objects in the class.
- The hierarchy is structured in a way that starts with the top class, which is the ios class, followed by other classes such as the *istream, ostream, iostream, istream_withassign,* and *ostream_withassign* classes.
- The *ios* class is the parent class in the hierarchy and both the istream and *ostream* classes inherit from it. These two classes together form the *ios* class, which is the highest level of the entire C++ stream class hierarchy.
- Other classes in the hierarchy provide functions for various operations, including assignment operations, such as the *_withassign* classes.

1. istream Class

- istream being a part of the ios class which is responsible for tackling all the input stream present within the stream.
- It provides functions for handling all the strings, chars, and objects within the istream class which comprises all these functions such as get, read, put, etc.

Example

- This program illustrates the istream class which takes a variable as an input then it makes use of the inbuilt functions like *get* to handle the input stream.

```
#include <iostream>
using namespace std;
int main()
{
   char p;
   cin.get(p);
   cout<< p;
}
```

```
Output:
R
R
```

2. ostream Class

- This class as part of the ios class is also considered as a base class that is responsible for handling output stream and provides all the necessary functions for handling chars, strings, and objects such as put, write, etc.

Example

- This program demonstrates the ostream class as part of the ios class where the first initialized char defined is scanned and then it gets the scanned character and the ostream function takes care to write or put the value to the function.

```
#include <iostream>
using namespace std;
intmain()
{
  char r_t;
  cin.get(r_t);
  cout.put(r_t);
}
```

```
Output:
A
A
```

3. iostream Class

- iostream class is the next hierarchy for the ios class which is essential for input stream as well as output stream because istream class and ostream class gets inherited into the main base class.
- As the name suggests it provides functionality to tackle the objects, strings, and chars which includes inbuild functions of put, puts, get, etc.

Example : This program is used to demonstrate the iostream class which comprises functions like write to print the input stream with the required number of values as input as shown in the output.

```
#include <iostream>
using namespace std;
intmain()
{
  cout.write("educba_portal", 9);
}
```

```
Output:
 educba_po
```

4. ios Class

- ios class is the highest class in the entire hierarchical structure of the C++ stream. It is also considered as a base class for istream, ostream, and streambuf class.
- It can be said that the ios class is basically responsible for providing all the input and output facilities to all the other classes in the stream class of C++.

```
#include <iostream>
using namespace std;
intmain()
{
  cout<<"Get the value for the _io_stream generation";
  return 0;
}
```

```
Get the value for the _io_stream genration.
```

5. istream_withassign Class

- This class is considered as a variant for the istream class that provides the class privilege for the class to assign object.
- The predefined object which can be called a build in the function of this class is used which is responsible for providing getting the stream facility and thus allows the object to reassign at the run time for different stream objects.

Example

This program demonstrates the istream_withassign class which is responsible for creating the object of the class as shown in the given output.

```
#include <iostream>
using namespace std;
intmain()
{
char istream_withassign[8];
std::cin.get(istream_withassign, 8);
std::cout<< istream_withassign << '\n';
std::cin.get(istream_withassign, 8);
std::cout<< istream_withassign << '\n';
return 0;
}
```

Output:
Readytofight
Readyto
Fight

6. ostream_withassign Class

This class is responsible for providing object assigned to the class and is considered as a variant itself for the ostream class of the C++ stream. All the build-in functions such as cout, cerr, clog is the already present objects of the same class and are reassigned at execution time for the different ostream object.

Example :

This program demonstrates the ostream_withassign class which is responsible for creating the object of the class as shown in the given output.

```
#include <iostream>
using namespace std;
Int main()
{
char ostream_withassign[10];
std::cin.get(ostream_withassign, 10);
std::cout<<ostream_withassign<< '\n';
std::cin.get(ostream_withassign, 10);
std::cout<<ostream_withassign<< '\n';
return 0;
}
```

Output:

```
gethteworld
gethtewor
ld
```

5.7 Formatted I/O

- C++ helps you to format the I/O operations like determining the number of digits to be displayed after the decimal point, specifying number base etc.
- The **iomanip.h** and **iostream.h** header files are used to perform the formatted IO operations in C++.

In C++, there are two ways to perform the formatted IO operations.

- **Using the member functions of ios class.**
- **Using the special functions called manipulators defined in iomanip.h.**

5.7.1 Formatted IO using ios class memebers

- The **ios** class contains several member functions that are used to perform formmated IO operations.
- The **ios** class also contains few format flags used to format the output. It has format flags like **showpos**, **showbase**, **oct**, **hex**, etc. The format flags are used by the function **setf( )**.

- The following table provides the details of the functions of **ios** class used to perform formatted IO in C++.

Function	Description
width(int)	Used to set the width in number of character spaces for the immediate output data.
fill(char)	Used to fill the blank spaces in output with given character.
precision(int)	Used to set the number of the decimal point to a float value.
setf(format flags)	Used to set various flags for formatting output like showbase, showpos oct, hex, etc.
unsetf(format flags)	Used to clear the format flag setting.

- All the above functions are called using the built-in object **cout**.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    cout << "Example for formatted IO" << endl;

    cout << "Default: " << endl;
    cout << 123 << endl;

    cout << "width(5): " << endl;
    cout.width(5);
    cout << 123 << endl;

    cout << "width(5) and fill('*'): " << endl;
    cout.width(5);
    cout.fill('*');
    cout << 123 << endl;
```

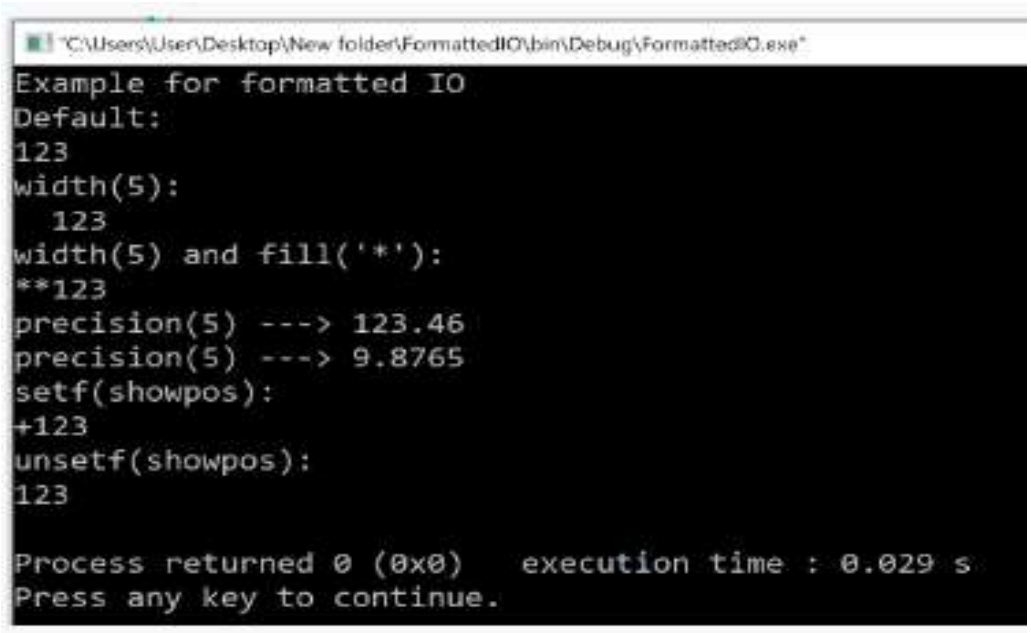
```
    cout.precision(5);
    cout << "precision(5) ---> " << 123.4567890 << endl;
    cout << "precision(5) ---> " << 9.876543210 << endl;

    cout << "setf(showpos): " << endl;
    cout.setf(ios::showpos);
    cout << 123 << endl;

    cout << "unsetf(showpos): " << endl;
    cout.unsetf(ios::showpos);
    cout << 123 << endl;

    return 0;
}
```

Output :



5.7.2 Formatted IO using manipulators :

- The **iomanip.h** header file contains several special functions that are used to perform formmated IO operations.
- The following table provides the details of the special manipulator functions used to perform formatted IO in C++.

Function	Description
setw(int)	Used to set the width in number of characters for the immediate output data.
setfill(char)	Used to fill the blank spaces in output with given character.
setprecision(int)	Used to set the number of digits of precision.
setbase(int)	Used to set the number base.
setiosflags(format flags)	Used to set the format flag.
resetiosflags(format flags)	Used to clear the format flag.

The **iomanip.h** also contains the following format flags using in formatted IO in C++.

Flag	Description
endl	Used to move the cursor position to a newline.
ends	Used to print a blank space (null character).
dec	Used to set the decimal flag.
oct	Used to set the octal flag.
hex	Used to set the hexadecimal flag.

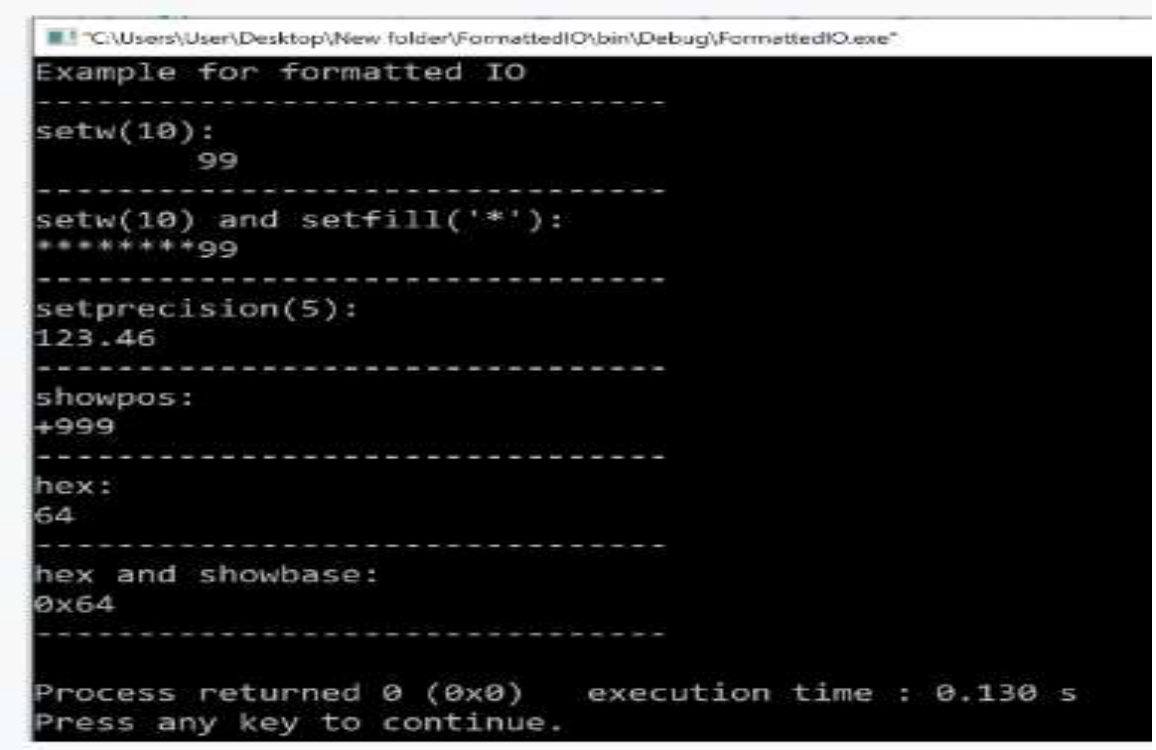
Example - Code to illustrate the formatted IO using manipulators

```
#include <iostream>
#include <fstream>
using namespace std;
void line() {
    cout << "------------------------------" << endl;
}
int main()
{
    cout << "Example for formatted IO" << endl;
    line();
    cout << "setw(10): " << endl;
    cout << setw(10) << 99 << endl;
    line();
    cout << "setw(10) and setfill('*'): " << endl;
    cout << setw(10) << setfill('*') << 99 << endl;
    line();
    cout << "setprecision(5): " << endl;
    cout << setprecision(5) << 123.4567890 << endl;
    line();
```

```
    cout << "showpos: " << endl;
    cout << showpos << 999 << endl;
    line();
    cout << "hex: " << endl;
    cout << hex << 100 << endl;
    line();
    cout << "hex and showbase: " << endl;
    cout << showbase << hex << 100 << endl;
    line();

    return 0;
}
```

Output :



Module – 5
Chapter 3 :

File I/O:

5.8 <fstream> and File Classes

File handling in C++ is a mechanism to store the output of a program in a file and help perform various operations on it. Files help store these data permanently on a storage device.

The fstream Library

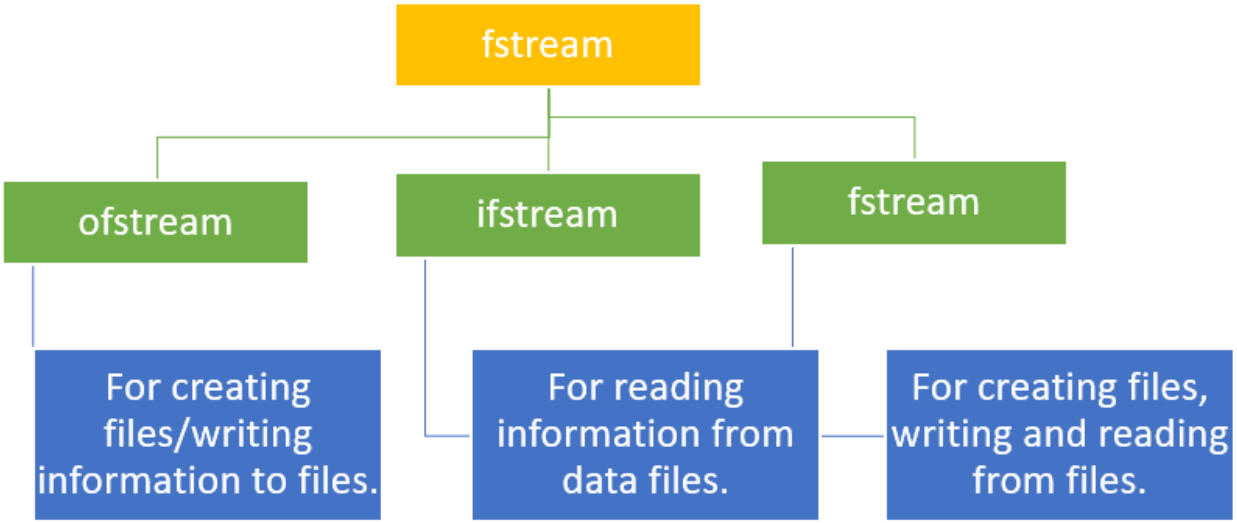
The fstream library provides C++ programmers with three classes for working with files. These classes include:

ofstream– This class represents an output stream. It's used for creating files and writing information to files.

ifstream– This class represents an input stream. It's used for reading information from data files.

fstream– This class generally represents a file stream. It comes with ofstream/ifstream capabilities. This means it's capable of creating files, writing to files, reading from data files.

The following image makes it simple to understand:



To use the above classes of the fstream library, you must include it in your program as a header file. Of course, you will use the #include preprocessor directive. You must also include the iostream header file.

Example:

```
#include<iostream>
#include<fstream>
```

5.9 Opening and Closing a File

- Before performing any operation on a file, you must first open it. If you need to write to the file, open it using fstream or ofstream objects.
- If you only need to read from the file, open it using the ifstream object.
- The three objects, that is, fstream, ofstream, and ifstream, have the open() function defined in them. The function takes this syntax:

```
open (file_name, mode);
```

The **file_name** parameter denotes the name of the file to open.

The **mode** parameter is optional. It can take any of the following values:

Value	Description
ios:: app	The Append mode. The output sent to the file is appended to it.
ios::ate	It opens the file for the output then moves the read and write control to file's end.
ios::in	It opens the file for a read.
ios::out	It opens the file for a write.
ios::trunc	If a file exists, the file elements should be truncated prior to its opening.

It is possible to use two modes at the same time. You combine them using the | (OR) operator.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
      fstream my_file;
      my_file.open("my_file", ios::out);
      if (!my_file)
      {
            cout << "File not created!";
      }
      else
      {
            cout << "File created successfully!";
            my_file.close();
      }
      return 0;
}
```

Output: File created successfully!

Once a C++ program terminates, it automatically

- flushes the streams
- releases the allocated memory
- closes opened files.

However, as a programmer, you should learn to close open files before the program terminates.

The fstream, ofstream, and ifstream objects have the close() function for closing files. The function takes this syntax:

```
void close();
```

5.10 Reading and Writing Text Files

You can write to file right from your C++ program. You use stream insertion operator (<<) for this. The text to be written to the file should be enclosed within double-quotes.

Output: File created successfully!

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
	fstream my_file;
	my_file.open("my_file.txt", ios::out);
	if (!my_file) {
		cout << "File not created!";
	}
	else {
		cout << "File created successfully!";
		my_file <<"AIT CSE";
		my_file.close();
	}
	return 0;
}
```

How to Read from Files

- You can read information from files into your C++ program. This is possible using stream extraction operator (>>).
- You use the operator in the same way you use it to read user input from the keyboard. However, instead of using the cin object, you use the ifstream/ fstream object.

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
	fstream my_file;
	my_file.open("my_file.txt", ios::in);
	if (!my_file) {
		cout << "No such file";
	}
	else {
		char ch;
		while (1) {
			my_file >> ch;
			if (my_file.eof())
				break;
			cout << ch;
		}
	}
	my_file.close();
	return 0;
}
```

Output: AIT CSE

5.11 Detecting EOF

- EOF() is a bool-type constant method that returns true when the end of the file is reached and it returns false when there is still data to be read.
- It is a method of input-output stream class (ios) that reads the data of a file until the end is encountered.

Syntax:

bool eof() const;

- This method is a Boolean method of constant type, which means it cannot be changed. Either it returns true or false. This function has no parameters.

Return Type:

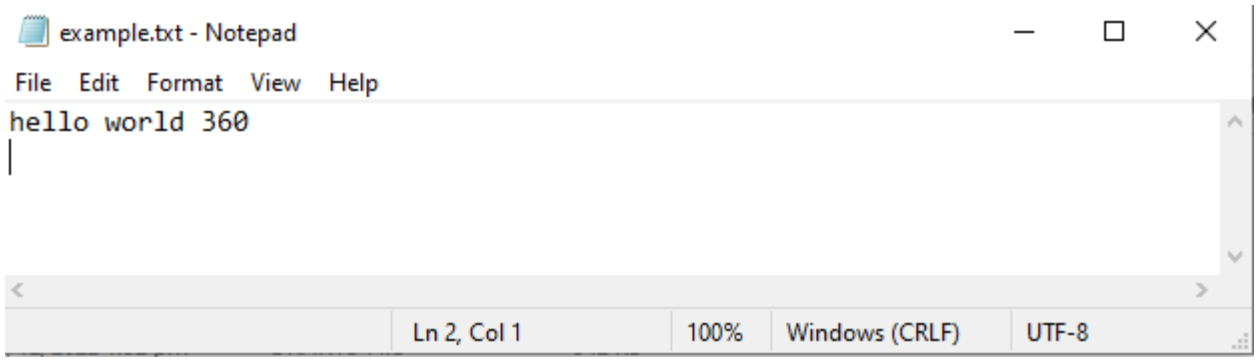
- When it returns “true”, it means that the end is reached. When it returns “false”, it means that the end is not encountered yet.

Use EOF() to Read a File that is Already Created

- The program reads the data from a text file. When it reaches the end, it displays a message. To do all these, we call the eof() function.

Create a File:

- First of all, create a text file, “example.txt”. Add some content to it and then save the file. This file should be stored in a similar folder as the C++ code.



```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   ifstream ist("example.txt");
   char ch;
   while (ist.get(ch))
      cout << ch;
   if (ist.eof())
     cout << "[EoF reached]\n";
   else
      cout << "[Error Reading]\n";
   ist.close();
   return 0;
}
```

Output: hello world 360 [EoF reached]