UNIT-II

Pointers in c++:

A **pointer** is a variable whose value is the address of another variable. every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory.

The general form of a pointer variable declaration is −

type *var-name;

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer.

Using Pointers in C++

(a) We define a pointer variable.

(b) Assign the address of a variable to a pointer.

(c) Finally access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Following example makes use of these operations −

```
#include <iostream>

using namespace std;

int main () {
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable

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

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

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

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

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

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

Pointer and Object:

the object pointers are declared by placing in front of a object pointer's name. It takes the following general form :

```
class-name * object-pointer ;
```

where class-name is the name of an already defined class and object-pointer is the pointer to an object of this class type. For example, to declare optr as an object pointer of Sample class type, we shall write

```
Sample *optr ;
```

where Sample is already defined class. When accessing members of a class using an object pointer, the arrow operator (->) is used instead of dot operator.

this Pointer in C++

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

In C++ programming, **this** is a keyword that refers to the current instance of the class.

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

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

C++ this Pointer Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

```
1.  #include <iostream>
2.  using namespace std;
3.  class Employee {
4.     public:
5.         int id; //data member (also instance variable)
6.         string name; //data member(also instance variable)
7.         float salary;
8.         Employee(int id, string name, float salary)
9.          {
10.             this->id = id;
11.             this->name = name;
12.             this->salary = salary;
13.         }
14.        void display()
15.         {
16.             cout<<id<<"  "<<name<<"  "<<salary<<endl;
17.         }
18. };
```

```
19. int main(void) {
20.     Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
21.     Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
22.     e1.display();
23.     e2.display();
24.     return 0;
25. }
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

Virtual function

Virtual function is a member function that is declared within the base class and can be redefined by the derived class.

Syntax:

There are two ways of creating a virtual function:

```
virtual void display() = 0; (OR)

virtual void display() { }
```

An Example program:

```
1.  #include <iostream>
2.  using namespace std;
3.  class base
4.  {
5.      public:
6.      void show()
7.      {
```

```
8.          std::cout << "Base class" << std::endl;
9.      }
10. };
11. class derived1 : public base
12. {
13.     public:
14.     void show()
15.     {
16.         std::cout << "Derived class 1" << std::endl;
17.     }
18. };
19. class derived2 : public base
20. {
21.     public:
22.     void show()
23.     {
24.         std::cout << "Derived class 2" << std::endl;
25.     }
26. };
27. int main()
28. {
29.     base *b;
30.     derived1 d1;
31.     derived2 d2;
32.     b=&d1;
33.     b->show();
34.     b=&d2;
35.     b->show();
36.     return 0;
37. }
```

In the above code, we have not used the virtual method. We have created a base class that contains the show() function. The two classes are also created named as '**derived1**' and '**derived2**' that are inheriting the properties of the base class. Both 'derived1' and 'derived2' classes have redefined the show() function. Inside the main() method, pointer variable 'b' of class base is declared. The objects of classes derived1 and derived2 are d1 and d2 respectively. Although the 'b' contains the addresses

of d1 and d2, but when calling the show() method; it always calls the show() method of the base class rather than calling the functions of the derived1 and derived2 class.

To overcome the above problem, we need to make the method as virtual in the base class. Here, virtual means that the method exists in appearance but not in reality. We can make the method as virtual by simply adding the virtual keyword proceeding to the function. In the above program, we need to add the virtual keyword that precedes to the show() function in the base class shown as below:

```
virtual void show()
    {
   std::cout << "Base class" << std::endl;
}
```

Once the above changes are made, the output would be:

Important points:

- It is a run-time polymorphism.
- Both the base class and the derived class have the same function name, and the base class is assigned with an address of the derived class object then also pointer will execute the base class function.
- If the function is made virtual, then the compiler will determine which function is to execute at the run time on the basis of the assigned address to the pointer of the base class.

pure virtual function

A pure virtual function is a virtual function that has no definition within the class. Let's understand the concept of pure virtual function through an example.

In the above pictorial representation, shape is the base class while rectangle, square and circle are the derived class. Since we are not providing any definition to the virtual function, so it will automatically be converted into a pure virtual function.

Characteristics of a pure virtual function

- A pure virtual function is a "do nothing" function. Here "do nothing" means that it just provides the template, and derived class implements the function.

- It can be considered as an empty function means that the pure virtual function does not have any definition relative to the base class.
- Programmers need to redefine the pure virtual function in the derived class as it has no definition in the base class.
- A class having pure virtual function cannot be used to create direct objects of its own. It means that the class is containing any pure virtual function then we cannot create the object of that class. This type of class is known as an abstract class.

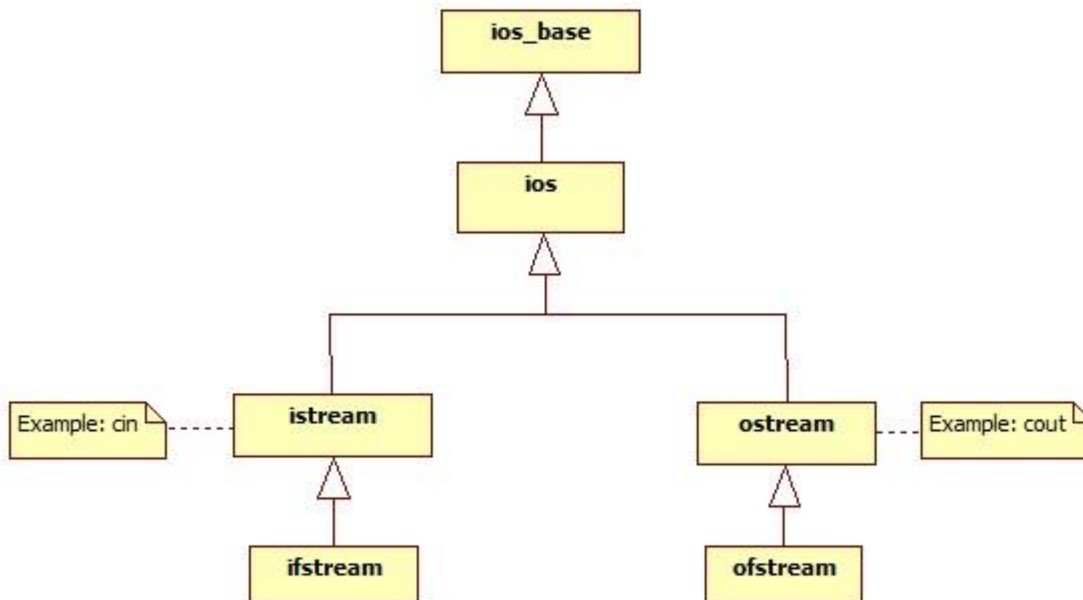
Differentiate Between the virtual and Pure virtual Function

Virtual function	**Pure virtual function**
A virtual function is a member function in a base class that can be redefined in a derived class.	A pure virtual function is a member function in a base class whose declaration is provided in a base class and implemented in a derived class.
The classes which are containing virtual functions are not abstract classes.	The classes which are containing pure virtual function are the abstract classes.
In case of a virtual function, definition of a function is provided in the base class.	In case of a pure virtual function, definition of a function is not provided in the base class.
The base class that contains a virtual function can be instantiated.	The base class that contains a pure virtual function becomes an abstract class, and that cannot be instantiated.
If the derived class will not redefine the virtual function of the base class, then there will be no effect on the compilation.	If the derived class does not define the pure virtual function; it will not throw any error but the derived class becomes an abstract class.
All the derived classes may or may not redefine the virtual function.	All the derived classes must define the pure virtual function.

Concept of stream:

- A C++ ***stream*** is a flow of data into or out of a program, such as the data written to cout or read from cin.
- For this class we are currently interested in four different classes:
    - **istream** is a general purpose input stream. cin is an example of an istream.

- **ostream** is a general purpose output stream. cout and cerr are both examples of ostreams.
- **ifstream** is an input file stream. It is a special kind of an istream that reads in data from a data file.
- **ofstream** is an output file stream. It is a special kind of ostream that writes data out to a data file.



Cout and Cin Object:

Standard output stream(Cout)

The **cout** is a predefined object of **ostream** class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

Example program for output stream (cout):

```
#include <iostream>
using namespace std;
int main( ) {
  char ary[] = "Welcome to C++ tutorial";
  cout << "Value of array is: " << ary << endl;
}
```

Output:

```
Value of array is: Welcome to C++ tutorial
```

Standard input stream (cin)

The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

Example program for input stream (cin):

```
#include <iostream>
using namespace std;
int main( ) {
int age;
cout << "Enter your age: ";
cin >> age;
cout << "Your age is: " << age << endl;
}
```

Output:

```
Enter your age: 22
Your age is:
T
```

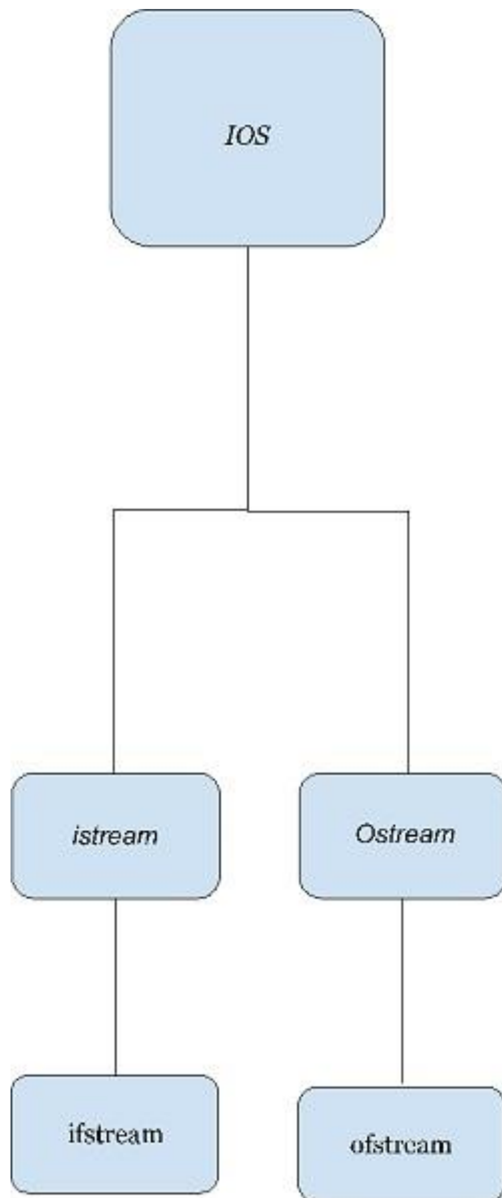
The stream classes”

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.

The **iostream.h** library holds all the stream classes in the C++ programming language.

Let's see the hierarchy and learn about them,



Now, let’s learn about the classes of the *iostream* library.

ios class − This class is the base class for all stream classes. The streams can be input or output streams. This class defines members that are independent of how the templates of the class are defined.

istream Class − The istream class handles the input stream in c++ programming language. These input stream objects are used to read and interpret the input as a sequence of characters. The cin handles the input.

ostream class − The ostream class handles the output stream in c++ programming language. These output stream objects are used to write data as a sequence of characters on the screen. cout and puts handle the out streams in c++ programming language.

Manipulators are special functions that can be included in the I/O statement to alter the format parameters of a stream. To access manipulators, the file iomanip.h should be included in the program. In this article, we have shared the list of Manipulators in C++ and functions of manipulators with examples.

What are Manipulators in C++?

manipulators are simply an instruction to the output stream that modify the output in various ways. In other words, we can say that Manipulators are operators that are used to format the data display.

Advantages and Purpose of Manipulators

- It is mainly used to make up the program structure.
- Manipulators functions are special stream function that changes certain format and characteristics of the input and output.
- To carry out the operations of the manipulators <iomanip.h> must be included.
- Manipulators functions are specially designed to be used in conjunction with insertion (<<) and extraction (>>) operator on stream objects.
- Manipulators are used to changing the format of parameters on streams and to insert or extract certain special characters.

Unformatted and formatted io operation:

Function	Working
width()	To set the width of the field. Output will be displayed in the given width.
precision()	To set the number of decimal places to display in a *float* value.
fill()	To set the character to be displayed in the blank space of a field.
setf()	To set various flags for formatting output.
unsetf()	To remove the flag settings.

Types of Manipulators in C++

They are of two types one taking arguments and the other without argument.

1. Non-argument manipulators (Without Parameters)

Non-argument manipulators are also known as “Parameterized manipulators”. These manipulators require iomanip header. Examples are setprecision, setw and setfill.

2. Argumented manipulators (With parameters)

Argument manipulators are also known as “Non parameterized manipulators”. These manipulators require iostream header. Examples are endl, fixed, showpoint, left and flush.

Standard input/output Manipulators in C++

Here is the list of standard input/output **Manipulators** and their Functions in C++

- **setw (int n)** – To set field width to n
- **Setbase** – To set the base of the number system
- **stprecision (int p)** – The precision is fixed to p
- **setfill (Char f)** – To set the character to be filled
- **setiosflags (long l)** – Format flag is set to l
- **resetiosflags (long l)** – Removes the flags indicated by l
- **endl** – Gives a new line

- **skipws** – Omits white space in input
- **noskipws** – Does not omit white space in the input
- **ends** – Adds null character to close an output string
- **flush** – Flushes the buffer stream
- **lock** – Locks the file associated with the file handle
- **ws** – Omits the leading white spaces present before the first field
- **hex, oct, dec** – Displays the number in hexadecimal or octal or in decimal format

C++ Manipulators

- Manipulators are operators used in C++ for **formatting output**. The data is manipulated by the programmer's choice of display.
- In this C++ tutorial, you will learn what a manipulator is, **endl** manipulator, **setw** manipulator, **setfill** manipulator and setprecision manipulator are all explained along with syntax and examples.

endl Manipulator:

- This manipulator has the same functionality as the 'n' newline character.

For example:

```
1. cout << "Exforsys" << endl;
2. cout << "Training";
```

setw Manipulator:

- This manipulator sets the minimum field width on output.

```
Syntax:

setw(x)
```

- Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator.
- The header file that must be included while using setw manipulator is .

Example:

```
#include <iostream>
#include <iomanip>

void main( )
{
int x1=123,x2= 234, x3=789;
cout << setw(8) << "Exforsys" << setw(20) << "Values" << endl
<< setw(8) << "test123" << setw(20)<< x1 << endl
<< setw(8) << "exam234" << setw(20)<< x2 << endl
<< setw(8) << "result789" << setw(20)<< x3 << endl;
}
```

```
Output:
test                   123
exam                   234
result                   789
```

setfill Manipulator:

- This is used after setw manipulator.
- If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

Example:

```
#include <iostream>
#include <iomanip>
void main()
{
cout << setw(15) << setfill('*') << 99 << 97 << endl;
}
```

```
Output:
```

```
********9997
```

setprecision Manipulator:

- The setprecision Manipulator is used with floating point numbers.
- It is used to set the number of digits printed to the right of the decimal point.
- This may be used in two forms:

1. fixed
2. scientific

- These two forms are used when the keywords fixed or scientific are appropriately used before the setprecision manipulator.
- The keyword fixed before the setprecision manipulator prints the floating point number in fixed notation.
- The keyword scientific, before the setprecision manipulator, prints the floating point number in scientific notation.

Example:

```
#include <iostream>
#include <iomanip>
void main( )
{
float x = 0.1;
cout << fixed << setprecision(3) << x << endl;
cout << scientific << x << endl;
}
```

```
Output:
0.100
1.000e-001
```

- The first cout statement contains fixed notation and the setprecision contains argument 3.

- This means that three digits after the decimal point and in fixed notation will output the first cout statement as 0.100. The second cout produces the output in scientific notation.
- The default value is used since no setprecision value is provided.

Dec, Oct,Hex Manipulator

All the numbers are displayed and read in decimal notation by default. However, you may change the base of an integer value to octal or hexadecimal or back to a decimal using the manipulator's oct, hex or dec, respectively. These manipulators are preceded by the appropriate variables to be used with.

```
1
2   #include<iostream.h>
3   #include<conio.h>
4   int: main() {
5    int i;
6    cout<<"Enter hexadecimal number =";
7    cin>>hex>>i;
8    cout:<<"Hexadecimal value = "<<hex<<i<<endl;
9    cout<<"Octal Value = "<<oct<<i<<endl;
10   cout<<"Dcimal Value = "<<dec<<i<<endl;
11   getch();
12   return 0;
13  }
14  Output :
15  Enter hexadecimal = f
16  Hexadecimal = f
17  Octal value = 17
    Decimal value = 15
```

In the above program, the variable i is inputted in hexadecimal form using hex manipulator in cin statement. To display the number in different notations like hexadecimal, octal and decimal, the manipulator's hex, oct and dee are used.

The base of a number remains the same until changed explicitly. For example, suppose the base of a number i of integer type is changed to hexadecimal (hex) during output. In that case, it will display the output only in hexadecimal form until being specified explicitly using dee or oct manipulators.

setbase(b) Manipulator

The setbase () manipulator is used to change the base of a numeric value during inputting and outputting. It is an alternative to Dec, Oct and hex manipulators. It is a function that takes a single integer argument(b) having values 8, 10 or 16 to set the base of the numeric value to octal, decimal and hexadecimal, respectively. The default base is 10.

```
1
2   #include<iostream.h>
3   #include<iomanip.h>
4   int main() {
5    int num;
6    cout<<"Enter number in Octal form = ";
7    cin>>setbase(8)>>num;
8    cout<<"Value of number in decimal form = "<<setbase(10)<<num<<endl;
9    cout<<"Value of number in octal form = "<<setbase(8)<<num<<endl;
10   cout<<"Value of number in hexadecimal form = "<<setbase(l6)<<num;
11   return 0;
12  }
13  Output
14  Enter numberin Octal form = 21
15  Value of number in decimal fonn = 17
16  Value of number in octal fonn = 21
    Value of number in hexadecimal fonn = 11
```

In the above program, the value of variable num is inputted in octal form using setbase(8) manipulator in the cin statement. The value of the variable num is displayed in different number systems by using setbase(b) manipulator with arguments values 10, 8 and 16.

setw(w) Manipulator

The setw() stands for set width. It is a function that takes a single integer argument which specifies the amount of space used to display the required value. We typically use the setw() manipulator for displaying output so that it becomes more understandable. It can be used to format only one value at a time.

```
1    #include<iostream.h>
2    #include<iomanip.h>
3    #include<conio.h>
4    int main() {
5     int age = 22,rollno = 910l;
6     cout<<setw(l2)<<"My Rollno is"<<setw(8)<<rollno<<endl;
7     cout<<setw(l2)<<"My Aqe is"<<setw(8)<<age;
8     getch();
9     return 0;
10   }
```

In the above program, the setw() manipulator causes the number or string that follows it in the cout statement to be displayed within the specified width. The value to be displayed is right-justified. The values of variable rollno and age are displayed within 8 spaces specified using setw(S) manipulator. The value is padded with leading blank spaces as it doesn’t fill the whole width.

If the value is larger than the specified width, it will not truncate the value, and the value is printed as it is. For example

```
1    cout<<setw(2)<<"Roll is"<<setw(2)<<rollno;
```

the output is

setfill(c) Manipulator

The setfill() manipulator is used in conjunction with the setw() manipulator. The compiler leaves the empty spaces on the left side of the required value if the set width is greater than the needed space. If you wish to fill the blank space with an alternative character instead of a blank space, you can use the setfill () manipulator.

Generally, you will not want to change the fill character. However, one common example of when you may want to is creating a program that prints cheques. To prevent the cheque amount from being altered by the user, computer-generated cheque amounts are usually printed with leading asterisks(*). This is done in C++ with a setfill() manipulator.

The setfill () manipulator takes a single character as an argument and fills the empty spaces with the specified character c on the left of the value displayed if the width specified using setw() manipulator is greater than the value to be displayed.

```
1
2    #inclucle<iostream.h>
3    #include<iomanip.h>
4    #include<conio.h>
5    int main() {
6     int age = 22,rollno = 910l; cout<<setfill('#');
7     cout<<setw(4)<<age<<setw(6)<<rollno<<endl;
8     cout<<setw(6)<<age<<setw(8)<<rollno;
9     getch();
10    return 0;
11   }
12   Output :
13   ##22##9101
     ####22####9101
```

setprecision(n) Manipulator

The setprecision() manipulator is used to control the precision of floating-point numbers, i.e. the number of digits to the right of the decimal point. By default, the precision of the floating-point number displayed is 6. This precision can be modified by using a setprecision () manipulator. This function takes an integer argument n that specifies the number of digits displayed after the decimal point. The floating-point number will be rounded to the specified precision.

```
1    #include<iostream.h>
2    #includi<iomanip.h>
3    #inelude<eonio.h>
4    int main() {
5     float a = 129.455396;
6     cout<<setprecision(2)<<a<<endl;
7     cout<<setprecision(3)<<a;
8     getch();
9     return 0;
10   }
11   Output :
     129.46
     129.455
```

12
13

In the above program, the setprecision (2) rounds the number a to 2 decimal places after the decimal point, i.e. 129.46.

In C++ programming

we are using the **iostream** standard library, it provides **cin** and **cout** methods for reading from input and writing to output respectively.

To read and write from a file we are using the standard C++ library called **fstream**. Let us see the data types define in fstream library is:

Data Type	Description
fstream	It is used to create files, write information to files, and read information from files.
ifstream	It is used to read information from files.
ofstream	It is used to create files and write information to the files.

C++ FileStream example: writing to a file

Let's see the simple example of writing to a text file **testout.txt** using C++ FileStream programming.

```
1. #include <iostream>
2. #include <fstream>
3. using namespace std;
4. int main () {
5.   ofstream filestream("testout.txt");
6.   if (filestream.is_open())
7.   {
8.     filestream << "Welcome to C++.\n";
```

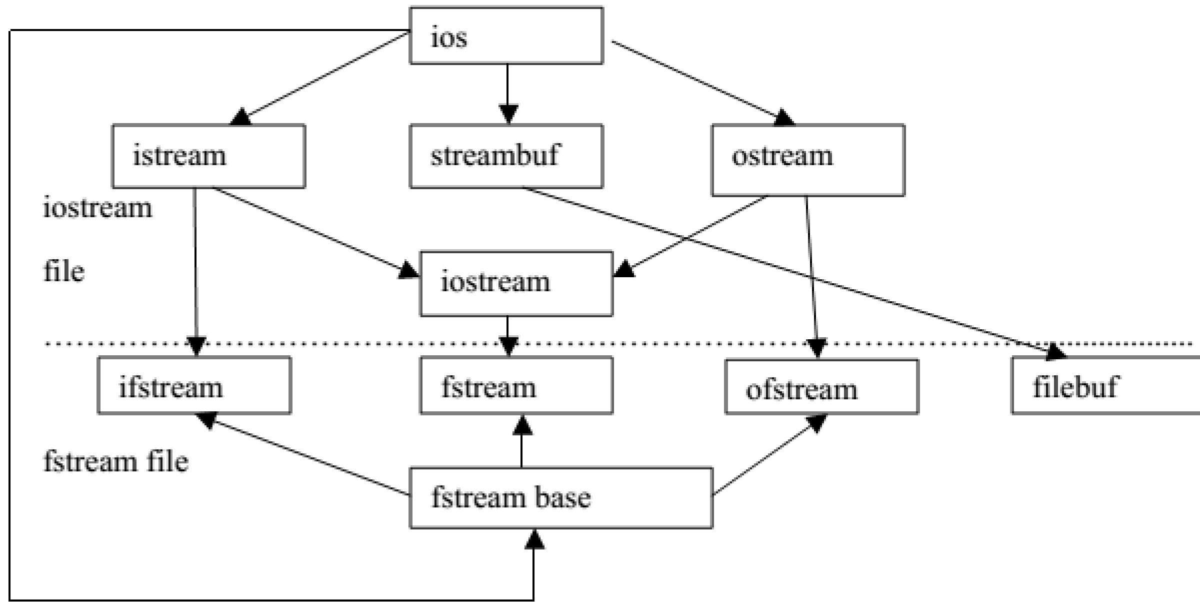
```
9.     filestream << "C++ Tutorial.\n";
10.    filestream.close();
11.   }
12.   else cout <<"File opening is fail.";
13.   return 0;
14. }
```

Output:

```
The content of a text file testout.txt is set with the data:
Welcome to javaTpoint.
C++ Tutorial.
```

File stream classes:

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



Stream classes for file operations

A file is a collection of related data stored in particular area on the disk.
The data transfer can take place in two ways,
1. Data transfer between the console unit and the program.
2. Data transfer between the program and a disk file.

File streams are used to transfer data between program and a disk file. Therefore a file stream acts as an interface between the programs and the files. The file stream I/O operations are similar to, consolestream I/O operations, The stream that provides the data to the program is called as an input stream and the stream that is used to receive the data from the program is called as an output stream. An input stream is used to extract the data from a file and an output stream is used to insert the data to a file. Performing input operations on file streams requires creation of an input stream and linking it with the program and the input file. Similarly performing output operations on file streams requires establishment of an output stream with the necessary links with the program and the output file.
The figure below shows the file stream hierarchy.

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

File management functions

the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types −

Sr.No	Data Type & Description
1	**ofstream** This data type represents the output file stream and is used to create files and to write information to files.
2	**ifstream**

	This data type represents the input file stream and is used to read information from files.
3	**fstream** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out**

	Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows −

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

Live Demo

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

int main () {
   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
```

```
   infile.close();

   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output −

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

C++ File Streams

A stream is a name given to a flow of data at the lowest level. At the lowest level, data is just the binary data without any notion of data type. Different streams are used to represent the different kinds of data flow such as whether data is flowing into the memory or out of the memory.

Each stream is associated with a particular class, that contains the member functions and definitions for dealing with that particular kind of data flow. For instance, the ifstream class represents the input disk files.

The stream that supplies data to the program is known as the input stream. It reads the data from the file and hands it over to the program. The stream that receives data from the program is known as output stream. It writes the received data to the file.

The file I/O system of C++ contains a set of classes that define the file handling methods. These classes, designed to manage the disk files, are declared in the header file named fstream.h. Therefore, we must include this file in a program that works with files. The classes defined inside the header file, fstream.h, derive from the classes under the header file iostream.h, the header file that manages console I/O operations.

Therefore, if you include the header file fstream.h in your file handling programs, you need not to include iostream.h header file as classes of fstream.h inherit from iostream.h only.

The function of these classes have been summarized in the following table :

Class	Functions
filebuf	It sets the file buffers to read and write. It contains close() and open() member functions to it
fstreambase	This is the base class for fstream, ifstream and ofstream classes. Therefore, it provides operations commo streams. It also contains the functions open() and close()

ifstream	Being an input file stream class, it provides the input operations for file. It inherits the functions get(), getli and functions supporting random access (seekg() and tellg()) from istream class defined inside the header iostream.h
ofstream	Being an output file stream class, it provides the output operations. It inherits the functions put() and writ functions supporting random access (seekp() and tellp()) from ostream class defined inside the header fil
fstream	It is an input-output file stream class. It provides support for the simultaneous input and output operation all the functions from istream and ostream classes through the iostream class defined inside the header fil

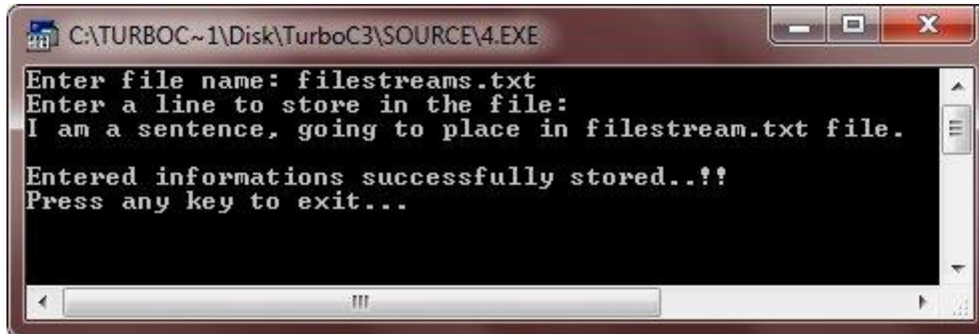
C++ File Streams Example

Here is a simple example program related to C++ file stream. This programs only ask to enter the file name and then to enter a line to store the line in this file.

```
/* C++ File Streams */

#include<conio.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
   char inform[80];
   char fname[20];
   char ch;
   clrscr();

   cout<<"Enter file name: ";
   cin.get(fname, 20);
   ofstream fout(fname, ios::out);

   if(!fout)
   {
      cout<<"Error in creating the file..!!\n";
      cout<<"Press any key to exit...\n";
      getch();
      exit(1);
   }
   cin.get(ch);

   cout<<"Enter a line to store in the file:\n";
   cin.get(inform, 80);
   fout<<inform<<"\n";

   cout<<"\nEntered informations successfully stored..!!\n";
   fout.close();
   cout<<"Press any key to exit...\n";
```

```
    getch();
}
```

Here is the sample run of the above C++ program:



More Examples

File Stream Classes:

filebuf: Its purpose is to set the file buffers to read and write. Openprot constant used in the open() of the file stream classes. It also contain close() and open() as members.

fstreambase: It provides operations common to the file streams. It serves as a base for fstream, ifstream and ofstream class. It also contain open() and close() functions.
ifstream: It provides input operations. It contains open() with default input mode. Inherits the functions get(), getline(), read(), seekg() and tellg() functions from istrem
ofstream: It provides output operations. It contains open() with default output mode. Inherits put(), seekp(), tellp() and write() functions from ostream.
fstream: It provides support for simultaneous input and output operations. inherits all the functions from istream and ostream classes through iostream.

Types of File Stream Classes

A file is a collection of related data stored in particular area on the disk.

The data transfer can take place in two ways,

1. Data transfer between the console unit and the program.

2. Data transfer between the program and a disk file.

File streams are used to transfer data between program and a disk file. Therefore a file stream acts as an interface between the programs and the files. The file stream I/O operations are similar to, consolestream I/O operations, The stream that provides the data to the program is called as an input stream and the stream that is used to receive the data from the program is called as an output stream. An input stream is used to extract the data from a file and an output stream is used to insert the data to a file. Performing input operations on file streams requires creation of an input stream and linking it with the program and the input file. Similarly performing output operations on file streams requires establishment of an output stream with the necessary links with the program and the output file.

The figure below shows the file stream hierarchy.

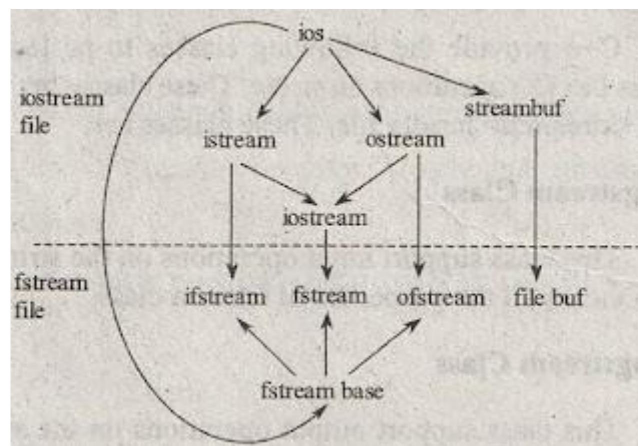


Figure: File Stream Class Hierarchy

C++ contains several classes that are used to perform file I/O operations. These are ifstream, ofstream and fstream classes which are derived from fstream base class and iostream class. The file stream classes are declared in the header file “fstream.h”.

1. ifstream Class

This class supports input operations on the files. It inherits the functions get( ), getline( ), read( ), seekg( ) and tellg( ) from istream class. When the file is opened in default mode, it contains open( ) function.

The functions of ifstream class are discussed below,

(i) get()

This function is used to read a single character from the file. The get pointer specifies the character which has to be read from the file. This function belongs to fstream class.

Also read :Explain in detail I/O streams along with an example program

(ii) getline()

This function reads a line of text which ends with a newline character. It cam be called using cin object as follows,

cin.getline(line, size)

Where line is a variable that reads a line of text.

Size is number of characters to be read getline() function reads the input until it encounters '\n' or size –1 characters are read. When '\n' is read. it is not saved instead it is replaced by the null character.

(iii) read()

This function reads a block of data of length specified by an argument 'n'. This function reads data sequentially. So when the EOF is reached before the whole block is read then the buffer will contain the elements read until EOF.

General syntax,

istream & read(char*str, size n);

(iv) seekg()

This function is used to shift the input pointer (i.e., get) to the given location. It belongs to ifstream. It is a file manipulator.

Also read :Draw console stream class hierarchy and explain its members

(v) tellg()

This function is used to determine the current position of the input pointer. It, belongs to ifstream class.

2. ofstream Class

This class supports output operations on the files. It inherits the functions put(), seekp(), tellp() and write() from ostream class. When the file is opened in default mode, it also contains the open() function.

The functions of ofstream class are discussed below,

(i) put();

This function is used to write a single character to the file. A stream object specifies to which file the character should be written. The character will be located at the position specified by the put pointer. This function belongs to fstream class.

(ii) . seekp()

This function is used to shift the output pointer (i.e., put) to the given location. It belongs to ofstream class. It is also a file manipulator.

(iii) tellp()

Also read :Discuss in brief about streams in C++

This function is used to determine the current position of the output pointer. It belongs to ofstream class.

(iv) write()

This function displays a line on the screen. It is called using cout object as follows,

cout.write(line, size)

Where line is the string to be displayed.

Size is the number of characters to be displayed.

If the size of string is greater than the line (i.e., text to be displayed) then write() function stop displaying on encountering null character but displays beyond the bounds of line.

(3) fstream Class

This class provides support for both input and output operations on the files at the same time. It inherits all the member functions of the istream and ostream classes through iostream class. when a file is opened in default mode, it also contains open() function.

(4) fstream Baseclass

This is the base class for ifstream, ofstream and fstream classes. It provides the operations that are common to the file streams. It contains open( ) and class( ) functions.

Also read :Differentiate between Machine level, Assembly and High level languages

(5) filebuf Class

This class is used to set the file buffers to read and write operations. it contains the functions open() and close(). It also contains a constant named Openport which is used

in opening of file stream classes using open() function.

(i) open()

This function is used to create new files as well as open existing files.

General Form

Stream-object open (“file name”, mode);

Where, ‘mode’ specifies the purpose of opening a file. This ‘mode’ argument is optional, if it is not given, then the prototype of member functions of the classes ifstream and ostream uses the default, values for this argument. The default value are,

ios :: in for ifstream functions i.e., open in read only mode

and ios :: out for ofstream functions i.e., open.in write only mode.

(ii) close()

A file can be closed using member function close(). This function neither takes any parameter nor returns any value.

Syntax

stream_name.close();

Here,

stream_name is the name of the stream to which file is linked.

Whether it is the programming world or not, files are vital as they store data. This article discuss working of file handling in C++. Following pointers will be covered in the article,

- Opening a File
- Writing to a File
- Reading from a File
- Close a File

File Handling In C++

Files are used to store data in a storage device permanently. File handling provides a mechanism to store the output of a program in a file and to perform various operations on it.

A stream is an abstraction that represents a device on which operations of input and output are performed. A stream can be represented as a source or destination of characters of indefinite length depending on its usage.

In C++ we have a set of file handling methods. These include ifstream, ofstream, and fstream. These classes are derived from fstrembase and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in fstream and therefore we must include fstream and therefore we must include this file in any program that uses files.

In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream.

- ofstream: This Stream class signifies the output file stream and is applied to create files for writing information to files
- ifstream: This Stream class signifies the input file stream and is applied for reading information from files
- fstream: This Stream class can be used for both read and write from/to files.

All the above three classes are derived from fstreambase and from the corresponding iostream class and they are designed specifically to manage disk files. C++ provides us with the following operations in File Handling:

- Creating a file: open()
- Reading data: read()
- Writing new data: write()
- Closing a file: close()

Moving on with article on File Handling in C++

Opening a File

Generally, the first operation performed on an object of one of these classes is to associate it to a real file. This procedure is known to open a file.

We can open a file using any one of the following methods:
1. First is bypassing the file name in constructor at the time of object creation.
2. Second is using the open() function.

To open a file use

```
1    open() function
```

Syntax

```
1    void open(const char* file_name,ios::openmode mode);
```

Here, the first argument of the open function defines the name and format of the file with the address of the file.

The second argument represents the mode in which the file has to be opened. The following modes are used as per the requirements.

Modes	***Description***
in	Opens the file to read(default for ifstream)
out	Opens the file to write(default for ofstream)
binary	Opens the file in binary mode
app	Opens the file and appends all the outputs at the end
ate	Opens the file and moves the control to the end of the file
trunc	Removes the data in the existing file
nocreate	Opens the file only if it already exists
noreplace	Opens the file only if it does not already exist

Example

```
1    fstream new_file;
2    new_file.open("newfile.txt", ios::out);
```

In the above example, new_file is an object of type fstream, as we know fstream is a class so we need to create an object of this class to use its member functions. So we create

new_file object and call open() function. Here we use out mode that allows us to open the file to write in it.

Default Open Modes :

- ifstream ios::in
- ofstream ios::out
- fstream ios::in | ios::out

We can combine the different modes using or symbol | .

Example

ofstream new_file;

```
1    new_file.open("new_file.txt", ios::out | ios::app );
```

Here, input mode and append mode are combined which represents the file is opened for writing and appending the outputs at the end.

As soon as the program terminates, the memory is erased and frees up the memory allocated and closes the files which are opened. But it is better to use the close() function to close the opened files after the use of the file.

Using a stream insertion operator << we can write information to a file and using stream extraction operator >> we can easily read information from a file.

Example of opening/creating a file using the open() function

```
1    #include<iostream>
2    #include <fstream>
3    using namespace std;
4    int main()
5    {
6    fstream new_file;
7    new_file.open("new_file",ios::out);
8    if(!new_file)
9    {
     cout<<"File creation failed";
```

```
10                }
11               else
12                {
13   cout<<"New file created";
14   new_file.close(); // Step 4: Closing file
15                }
16              return 0;
17                }
18
```

Output:

```
C:\Users\Lenovo\Desktop>g++ new_file.cpp -o new_file.exe

C:\Users\Lenovo\Desktop>new_file.exe
New file created
C:\Users\Lenovo\Desktop>
```

Explanation

In the above example we first create an object to class fstream and name it 'new_file'. Then we apply the open() function on our 'new_file' object. We give the name 'new_file' to the new file we wish to create and we set the mode to 'out' which allows us to write in our file. We use a 'if' statement to find if the file already exists or not if it does exist then it will going to print "File creation failed" or it will gonna create a new file and print "New file created".

Moving on with article on File Handling in C++

Writing to a File

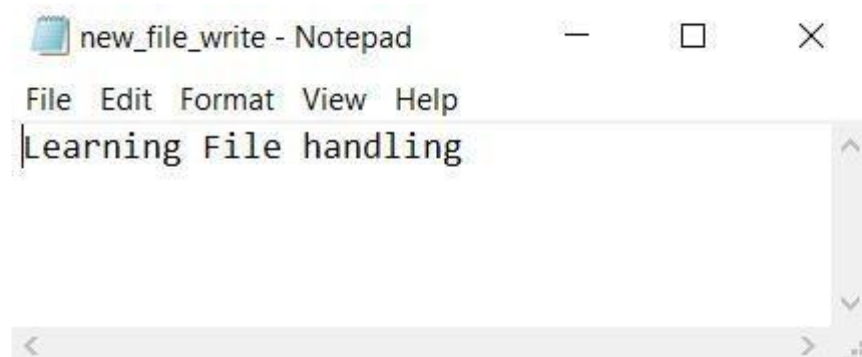
Example:

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4   int main()
```

```
5   {
6   fstream new_file;
7   new_file.open("new_file_write.txt",ios::out);
8   if(!new_file)
9   {
10  cout<<"File creation failed";
11  }
12  else
13  {
14  cout<<"New file created";
15  new_file<<"Learning File handling";    //Writing to file
16  new_file.close();
17  }
18  return 0;
19  }
```

Output:

```
C:\Users\Lenovo\Desktop>g++ new_file.cpp -o new_file.exe

C:\Users\Lenovo\Desktop>new_file.exe
New file created
C:\Users\Lenovo\Desktop>
```



Explanation

Here we first create a new file “new_file_write” using open() function since we wanted to send output to the file so, we use ios::out. As given in the program, information typed inside the quotes after Insertion Pointer “<<” got passed to the output file.

Moving on with this article on File Handling in C++

Reading from a File

Example

```
1
2   #include <iostream>
3   #include <fstream>
4   using namespace std;
5   int main()
6   {
7   fstream new_file;
8   new_file.open("new_file_write.txt",ios::in);
9   if(!new_file)
10  cout<<"No such file"; } else { char ch; while (!new_file.eof()) { new_file >>c
11  cout << ch;
12  }
13  new_file.close();
14  return 0;
    }
```

Output:

```
LearningFilehandlingg[Finished in 1.7s]
```

Explanation

In this example, we read the file that generated id previous example i.e. new_file_write. To read a file we need to use 'in' mode with syntax ios::in. In the above example, we print the content of the file using extraction operator >>. The output prints without any space because we use only one character at a time, we need to use getline() with a character array to print the whole line as it is.

Moving on with this article on File Handling in C++

Close a File

It is simply done with the help of close() function.

Syntax: File Pointer.close()

Example

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4  int main()
5  {
6  fstream new_file;
7  new_file.open("new_file.txt",ios::out);
8  new_file.close();
9  return 0;
10 }
```

Output:

The file gets closed.