# CS-1004 Object Oriented programming Week 14

Instructor:
ATIYA
LECTURER
(COMPUTER SCIENCE DEPARTMENT)
NATIONAL UNIVERSITY- FAST (KHI CAMPUS)
EMAIL: ATIYA.JOKHIO@NU.EDU.PK
ROOM: (LECTURER ROOM#19, CS DEPT BASEMENT-2 )

## Introduction to filing

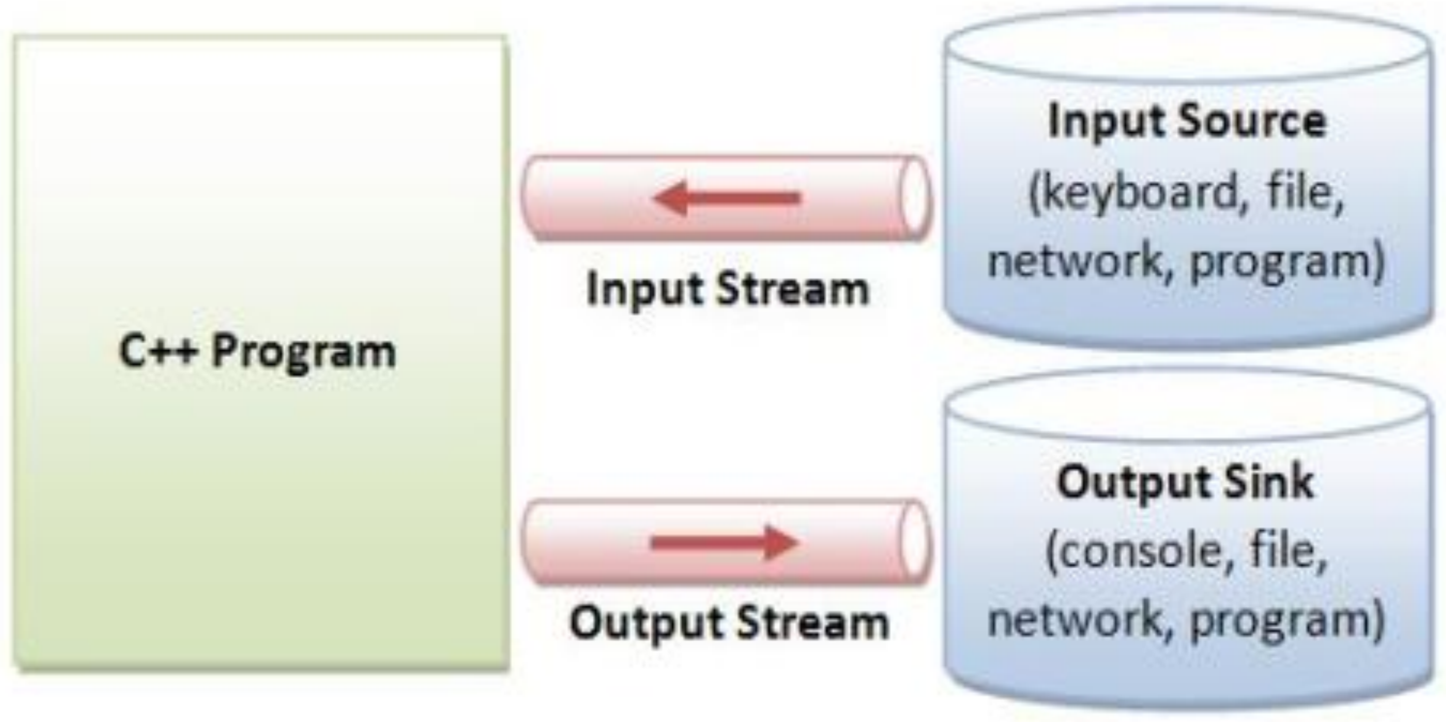
- Files are used to store data in a storage device permanently.
- File handling provides a mechanism to store the output of a program in a file and to perform various operations on it.
- A stream is an abstraction that represents a device on which operations of input and output are performed.

## Introduction to filing

- ASCII Text file
  - A text file can be a stream of characters that a computer can process sequentially.
- Binary file
  - A binary file is no different to a text file but it is a collection of bytes rather than characters.

## Streams

- **Stream**
  - A transfer of information in the form of a sequence of bytes
- **I/O Operations**
  - Input: A stream that flows from an input device ( i.e.: keyboard, disk drive, network connection) to main memory
  - Output: A stream that flows from main memory to an output device ( i.e.: screen, printer, disk drive, network connection)


C++ Program
Input Stream
Output Stream
Input Source
(keyboard, file,
network, program)
Output Sink
(console, file,
network, program)

## Iostream Library Header Files

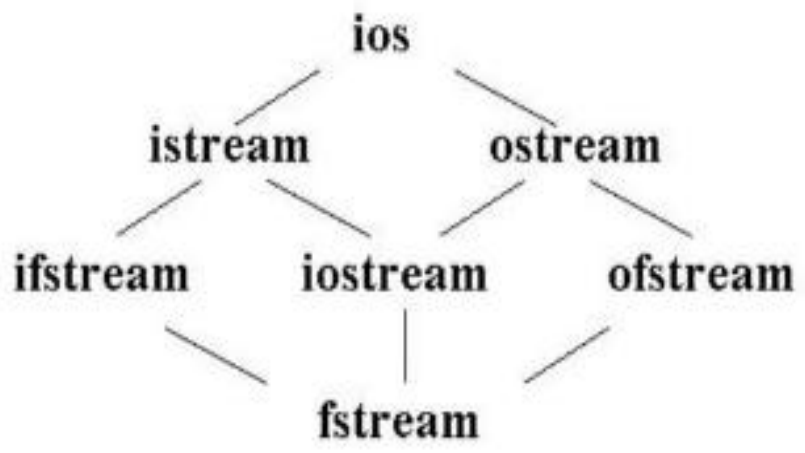
- **<iostream.h>:** Contains **cin & cout** objects

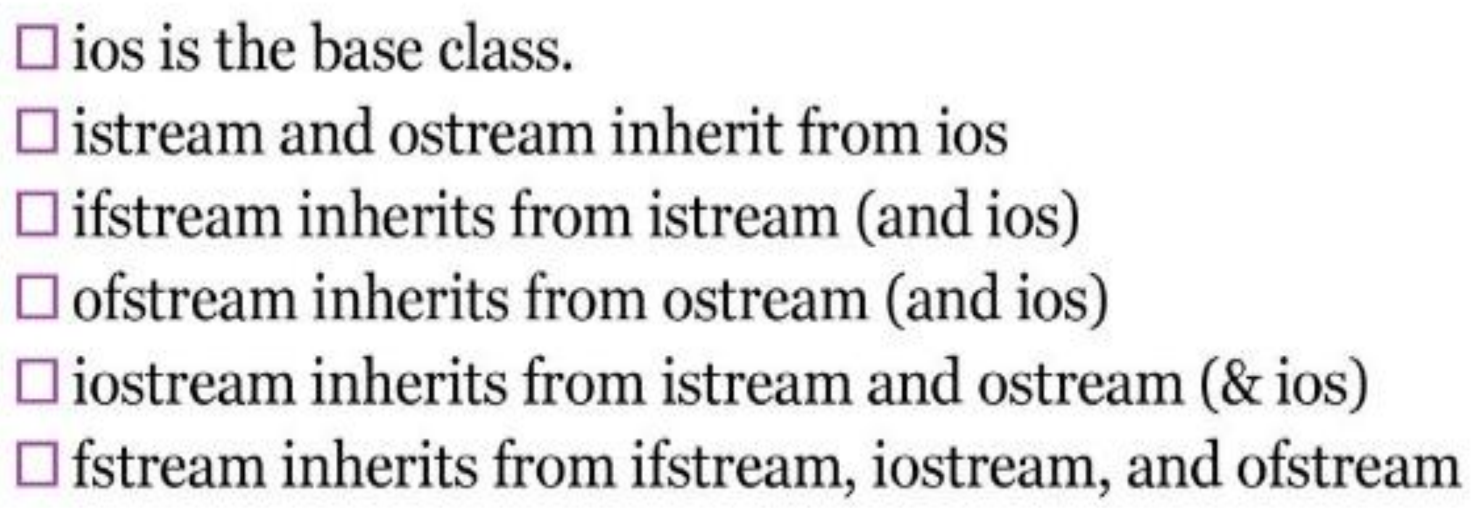
## Why use File Handling in C++:-

For permanent storage.

The transfer of input - data or output - data from one computer to another can be easily done by using files.

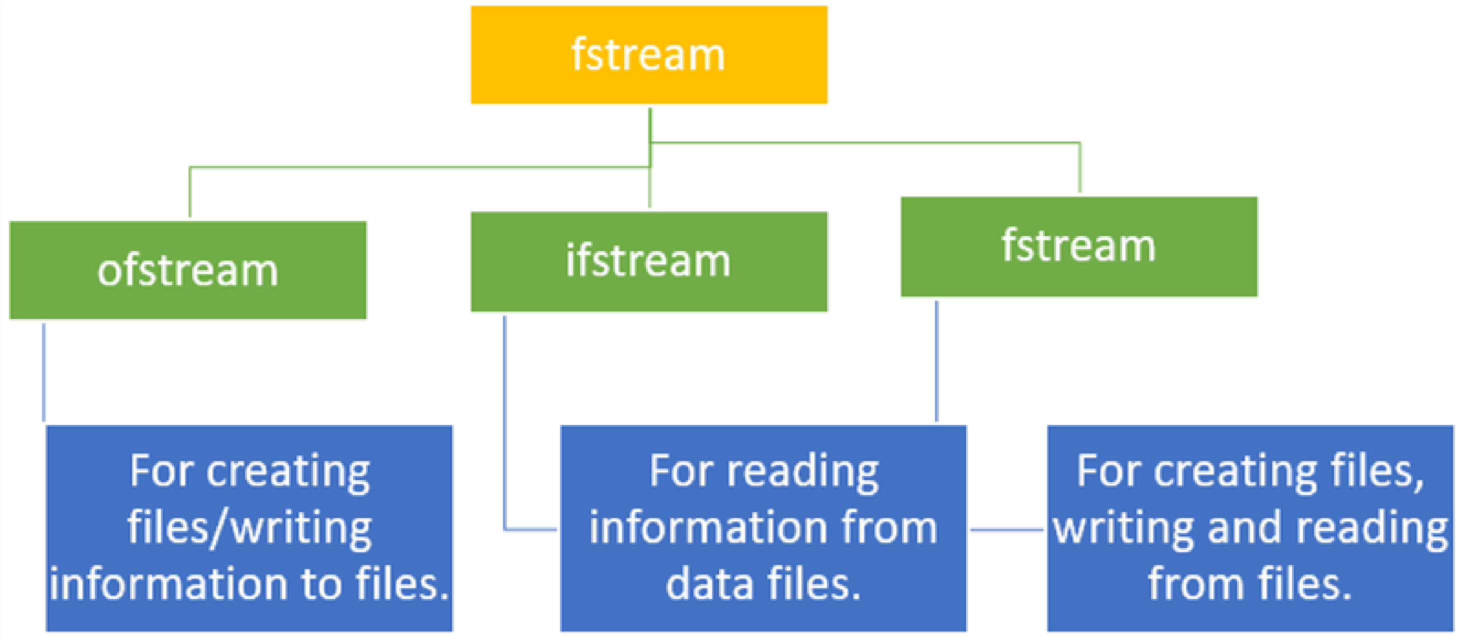
## Classes for stream input and output




- □ ios is the base class.
- □ istream and ostream inherit from ios
- □ ifstream inherits from istream (and ios)
- □ ofstream inherits from ostream (and ios)
- □ iostream inherits from istream and ostream (& ios)
- □ fstream inherits from ifstream, iostream, and ofstream

# File handling Classes

- These includes:
  - **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


fstream
ofstream
ifstream
fstream
For creating files/writing information to files.
For reading information from data files.
For creating files, writing and reading from files.

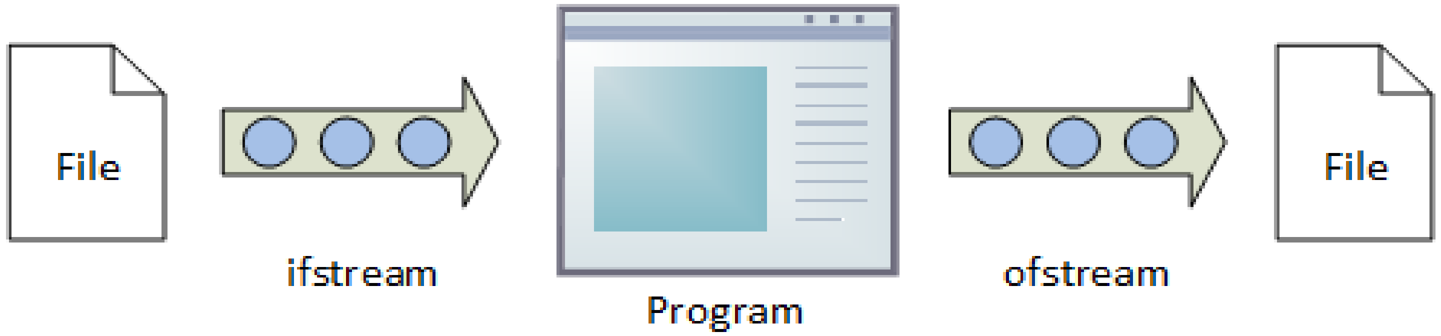
## File I/O

- To perform file I/O, you must include the header **<fstream> in your program**
- It defines several classes, including **ifstream, ofstream, and fstream**
- C++ views each file as a sequence of bytes

## File Stream

There are three types of file streams: input, output, and input/output

```
ifstream  in;     // input stream object
ofstream  out;    // output stream object
fstream  io;      // input/output stream object
```


File
ifstream
Program
ofstream
File

## Ofstream predefined methods

- This class is used to prepare an object which performs write operations on file.

- its predefined methods are

  - open()
  - write()
  - put()
  - seekp()
  - tellp()
  - close()

## ifstream predefined methods

- it is used to prepare an object which performs read operations on file.
- Its predefined methods are
- open()
- read()
- get()
- getline()
- seekg()
- tellg()
- close()

## fstream predefined methods

**It performs both read and write operations of file**

## Reading from a file

*//open a file for reading using constructor*
**ifstream in(“myfile”);**

*//open a text file for reading*
**ifstream in(“myfile.txt”);**

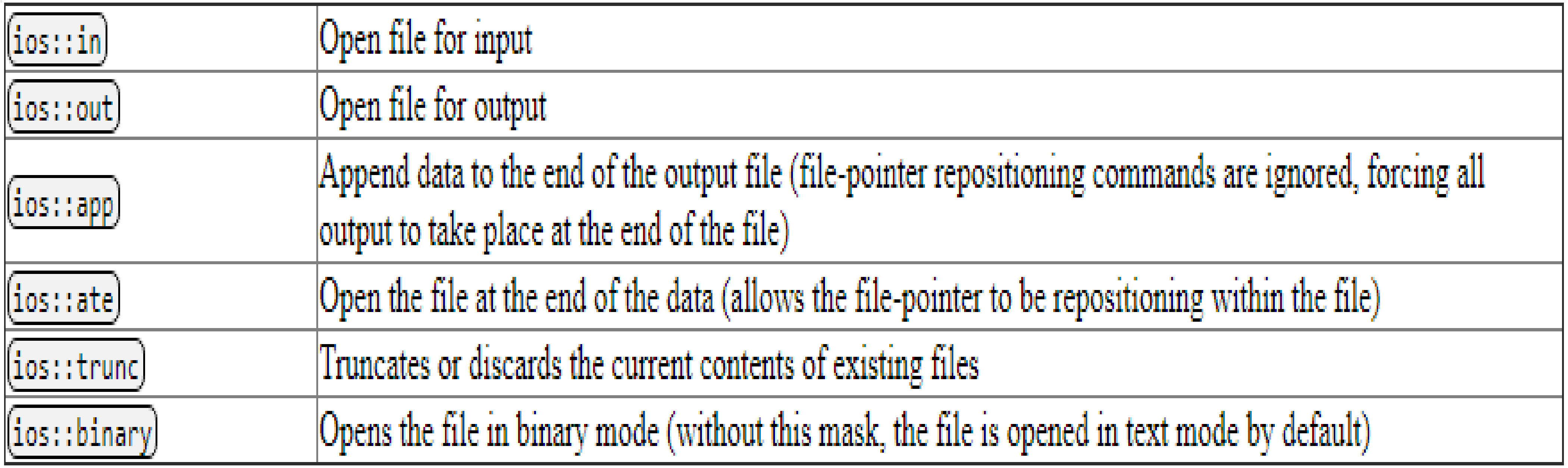
- if for some reason, file cannot be opened then ifstream object has the value ***false***

## Alternate Syntax

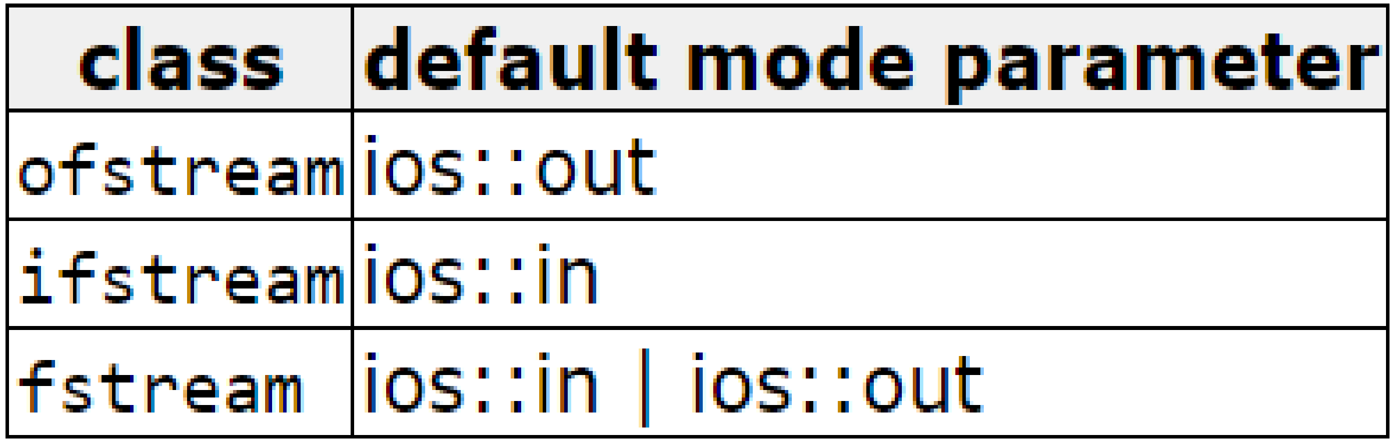
An alternate way to open a file for read/write is by using open() function

```
ofstream out;
out.open("test", ios::out);
```

## ios Modes



| | |
|---|---|
| `ios::in` | Open file for input |
| `ios::out` | Open file for output |
| `ios::app` | Append data to the end of the output file (file-pointer repositioning commands are ignored, forcing all output to take place at the end of the file) |
| `ios::ate` | Open the file at the end of the data (allows the file-pointer to be repositioning within the file) |
| `ios::trunc` | Truncates or discards the current contents of existing files |
| `ios::binary` | Opens the file in binary mode (without this mask, the file is opened in text mode by default) |



| class | default mode parameter |
|---|---|
| ofstream | ios::out |
| ifstream | ios::in |
| fstream | ios::in \| ios::out |

## ORing

All these flags can be combined using the bitwise operator OR (|).

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app );
```

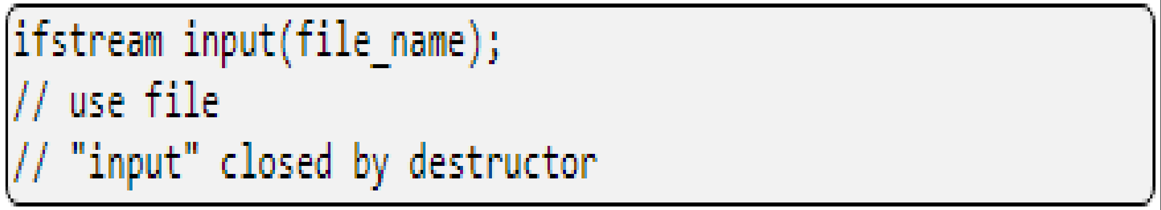
## Closing a file

To close a file, we can use the member function **close( )**

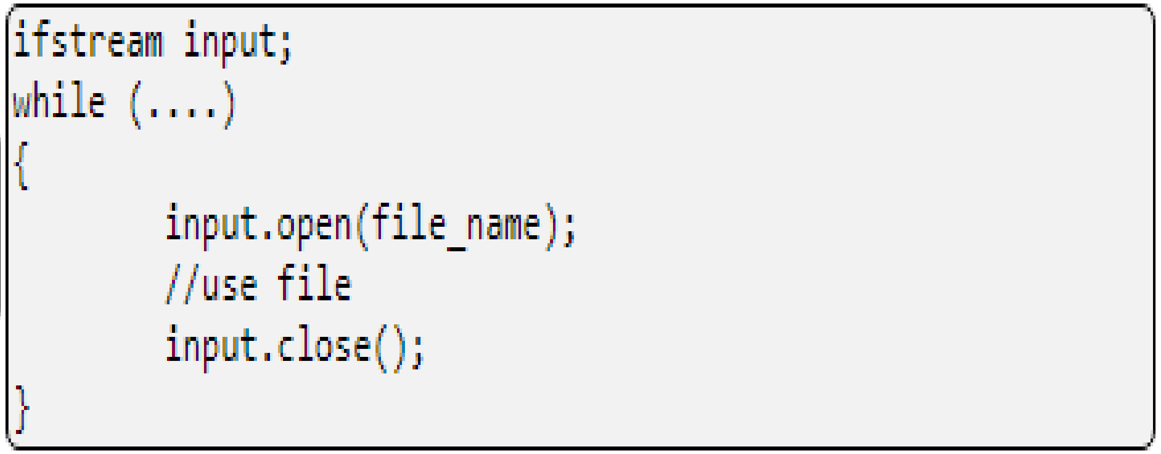
*Example:*

**mystream.close();**

## Opening And Closing Files



```
ifstream input(file_name);
// use file
// "input" closed by destructor
```



```
ifstream input;
while (....)
{
        input.open(file_name);
        //use file
        input.close();
}
```

## File open using constructor method

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
        ofstream f("XYZ");
        f << "hello";
        f.close();
}
```

## Opening File- Example

```
#include<iostream>
#include <fstream>
using namespace std;
int main()
{
fstream new_file;
new_file.open("new_file",ios::out);
if(!new_file)
{
cout<<"File creation failed";
}
else{
cout<<"New file created";
new_file.close();
}
return 0;}
```

## Open()

- Opening a file associates a file stream variable declared in the program with a physical file at the source, such as a disk.
- In the case of an input file:
  - the file must exist before the open statement executes.
  - If the file does not exist, the open statement fails and the input stream enters the fail state
- An output file does not have to exist before it is opened;
  - if the output file does not exist, the computer prepares an empty file for output.
  - If the designated output file already exists, by default, the old contents are erased when the file is opened.

# File I/O Example: Writing

- First Method (use the constructor)

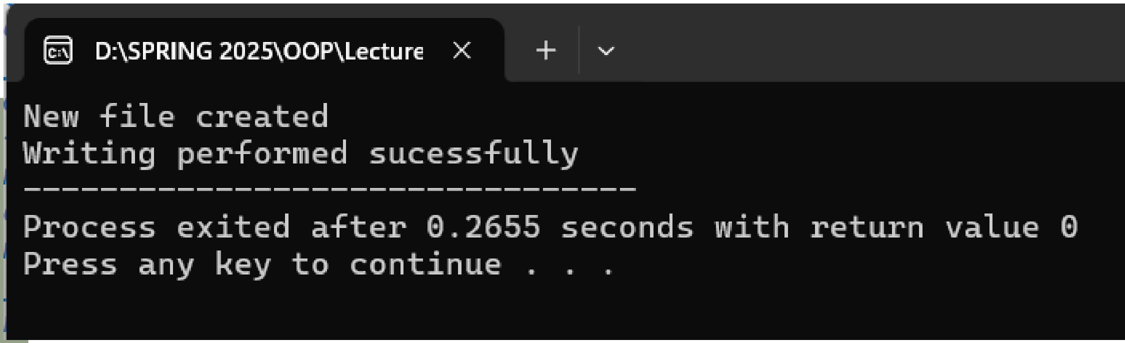
```
#include <fstream>
using namespace std;
int main()
{/* declare and automatically open the file*/
ofstream outFile("fout.txt");
//behave just like cout, put the word into  the file
outFile << "Hello World!";
outFile.close();
return 0;
}
```

- Second Method ( use Open function)

```
#include <fstream>
using namespace std;
int main(){// declare output file variable
ofstream outFile;
// open an exist file fout.txt
   outFile.open("fout.txt”);
//behave just like cout, put the word into  the file
outFile << "Hello World!";
outFile.close();
return 0;}
```

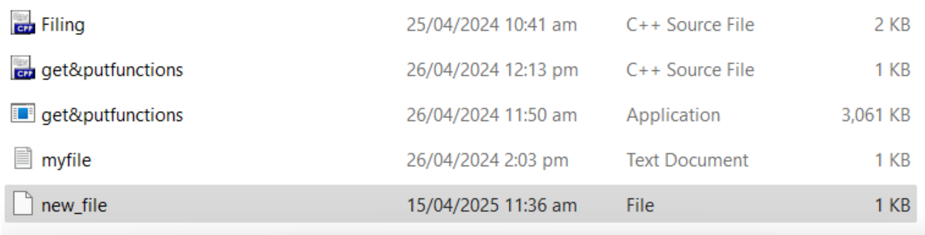
## Writing File- Example

```
#include<iostream>
#include <fstream>
using namespace std;
int main(){
fstream new_file;
new_file.open("new_file",ios::out);
if(!new_file)
{
cout<<"File creation failed";
}
else
{
cout<<"New file created";
new_file<<" Hello 2H-OOP [2025]";
cout<<"Writing performed sucessfully";
new_file.close(); // Step 4: Closing file
}return 0;}5
```

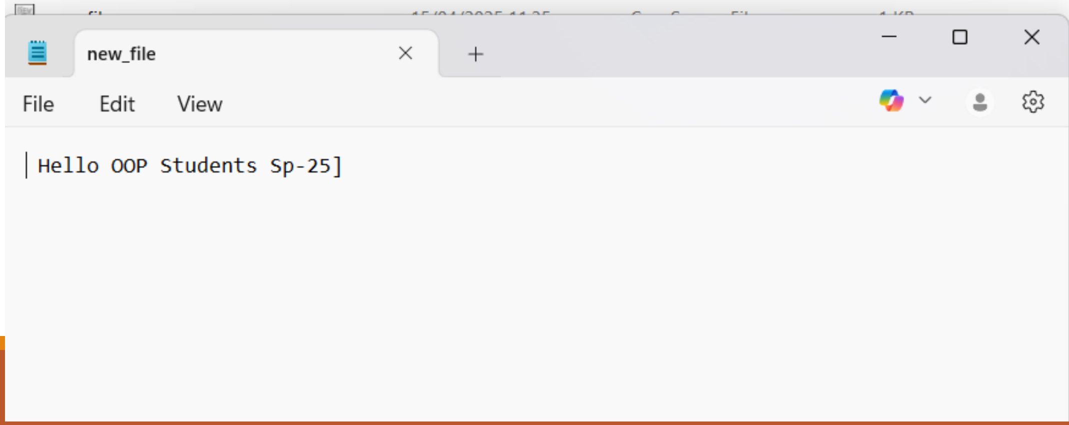


D:\SPRING 2025\OOP\Lecture

```
New file created
Writing performed sucessfully
--------------------------------
Process exited after 0.2655 seconds with return value 0
Press any key to continue . . .
```



| | | | |
|---|---|---|---|
| Filing | 25/04/2024 10:41 am | C++ Source File | 2 KB |
| get&putfunctions | 26/04/2024 12:13 pm | C++ Source File | 1 KB |
| get&putfunctions | 26/04/2024 11:50 am | Application | 3,061 KB |
| myfile | 26/04/2024 2:03 pm | Text Document | 1 KB |
| new_file | 15/04/2025 11:36 am | File | 1 KB |



```
Hello OOP Students Sp-25]
```

## formatted and unformatted IO

C++ provides both the formatted and unformatted IO functions.

In formatted or high-level IO, bytes are grouped and converted to types such as int, double, string or user-defined types.

In unformatted or low-level IO, bytes are treated as raw bytes and unconverted.

Formatted IO operations are supported via overloading the stream insertion (<<) and stream extraction (>>) operators, The << and >> operators are overloaded to handle fundamental types (such as int and double), and classes (such as string). You can also overload these operators for your own user-defined types.

## Types of I/O

We can perform either formatted or unformatted I/O with file stream

Formatted output is carried out on streams via the stream insertion << and stream extraction >> operators

Character translations are performed between console window and files

## Formatted I/O

Formatted I/O can be performed by using extraction **>>** and insertion **<<** operators

All information is stored in the file in the same format as it would be displayed on the screen

When reading text files using the **>>** operator, certain character translations occur. For example, white-space characters are omitted

Formatted output converts the internal binary representation of the data to ASCII characters which are written to the output file. Formatted input reads characters from the input file and converts them to internal form.

## Advantages and Disadvantages of Formatted I/O

Formatted input/output is very portable. It is a simple process to move formatted data files to various computers, even computers running different operating systems, as long as they all use the ASCII character set.

Formatted files are human readable and can be typed to the terminal screen or edited with a text editor.

## Validate the file before trying to access

**Method 1:**

By checking the stream variable;

```
If ( ! Mystream)
{
Cout << "Cannot open file.\n ";
}
```

**Method 2:**

By using bool is_open() function.

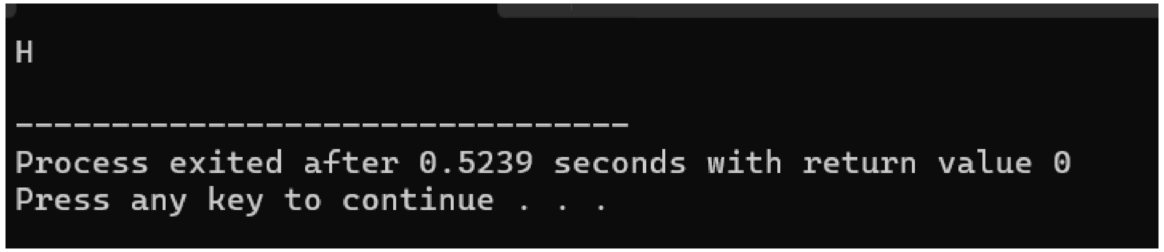
```
If ( ! Mystream.is_open())
{
Cout << "File is not open.\n ";
}
```

## To write to the file, use the insertion operator (<<)

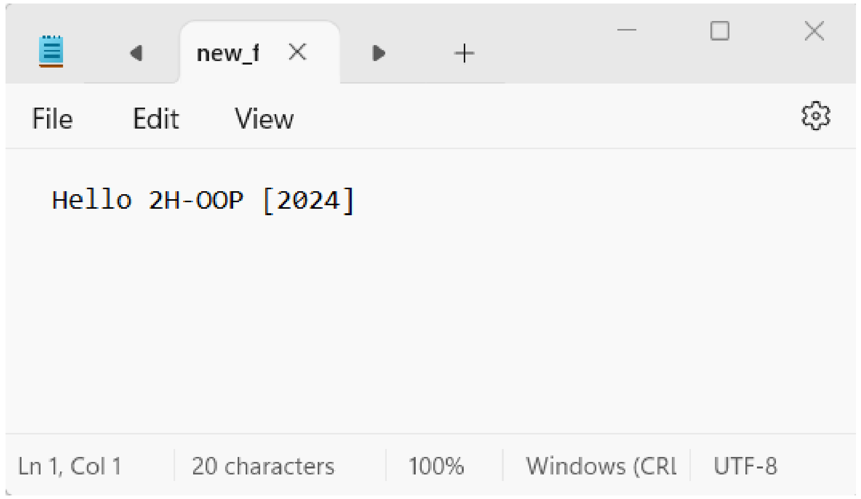
```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  // Create and open a text file
  ofstream MyFile("filename.txt");

  // Write to the file using insertion operator
  MyFile << "Files can be tricky, but it is fun enough!";

  // Close the file
  MyFile.close();
}
```

## Example for reading file

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
 ifstream in(“new_file");
 if(!in){
  cout << "Failed to open file." << endl;
   return 1; }
 char c[20];
   in >> c ;
   cout << c << endl;
 in.close();
 return 0;}
```



```
H

--------------------------------
Process exited after 0.5239 seconds with return value 0
Press any key to continue . . .
```

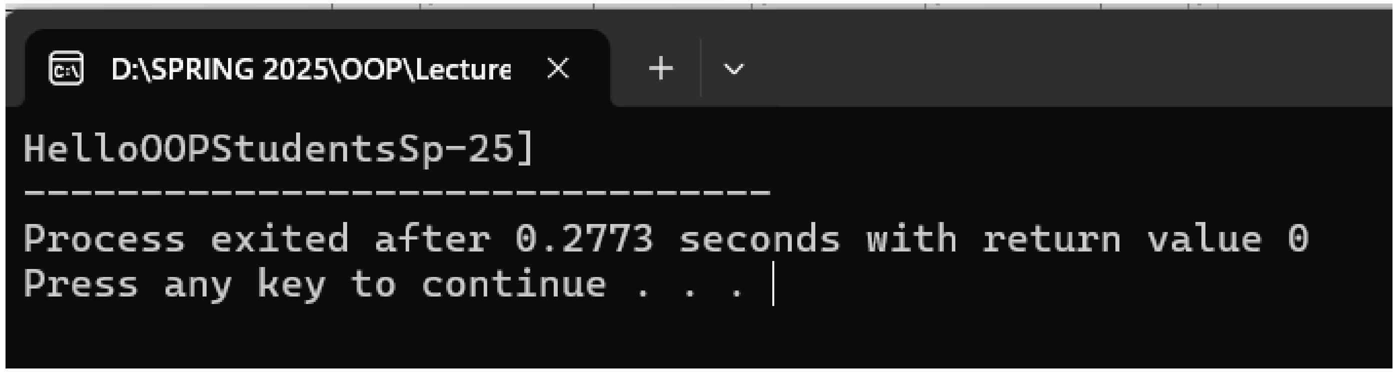


```
Hello 2H-OOP [2024]
```

Ln 1, Col 1 | 20 characters | 100% | Windows (CRL | UTF-8

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
        ifstream in(“new_file");
        if(!in){
         cout << "Failed to open file." << endl;
          return 1; }
        char c[20];
        while(!in.eof()){
          in >> c ;
          cout << c ;}
        in.close();
        return 0;}
```

## output



```
HelloOOPStudentsSp-25]
--------------------------------
Process exited after 0.2773 seconds with return value 0
Press any key to continue . . .
```

# Writing to a file

```
// open a file for writing
ofstream out("myfile");


// open a text file for writing
ofstream out("myfile.txt");
```

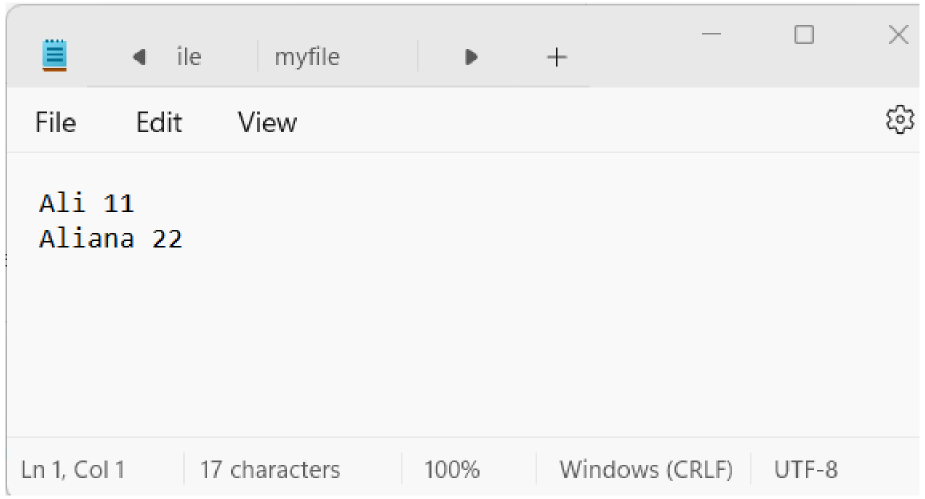
If file does not already exist then it is created

If the file cannot be opened or created, then ofstream object has the value ***false***

## Example

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
 ofstream out(“myfile.txt");
if(!out){
cout << "File creation failed";     return 1;}
out << "Ali " << 11 << endl;
out << "Aliana" << 22 << endl;
out.close();return 0;
}
```

# Output

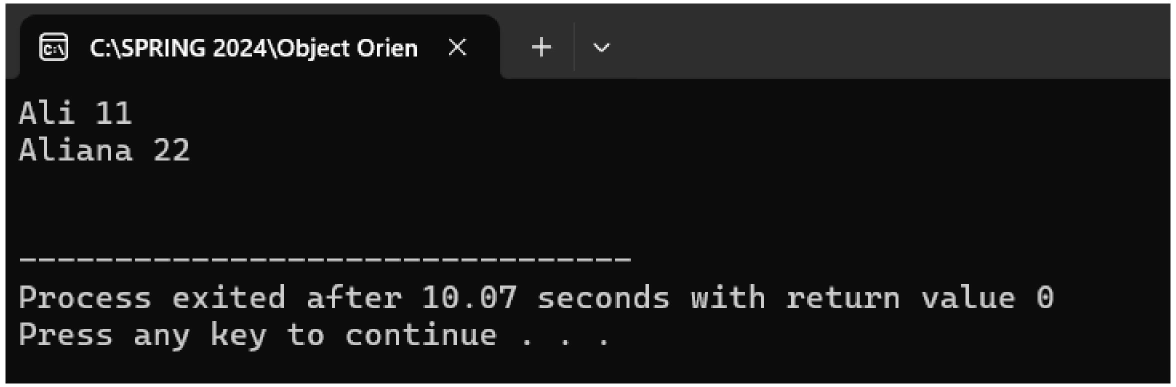


```
Ali 11
Aliana 22
```

## File Read through char array

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
		char str[12];
		ifstream f;
		f.open(“myfile.txt");
		while(f) //reading through file object
{
				f.getline(str,10);
				cout<<str<< endl;}
		f.close();}
```

## Output



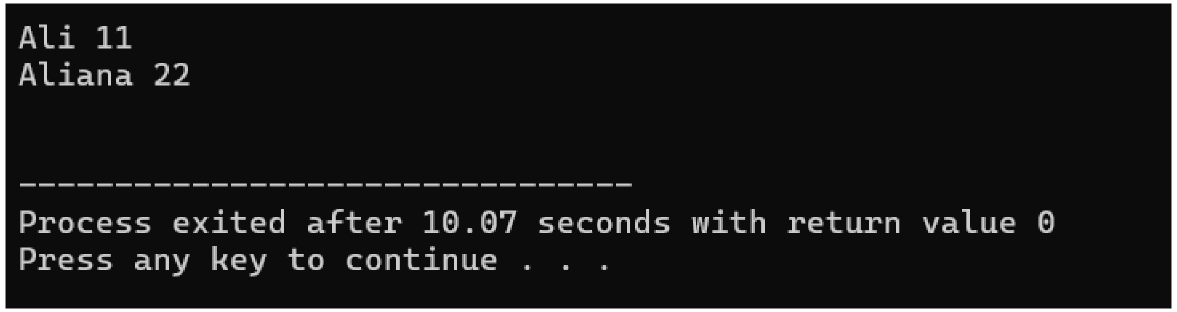
```
Ali 11
Aliana 22



--------------------------------
Process exited after 10.07 seconds with return value 0
Press any key to continue . . .
```

## File Read through string object

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
        string str;
        ifstream f;
        f.open("Test.txt");
        while(f){
                getline(f,str);
                cout<<str<<endl;    }
        f.close();}
```

## File Read through string object

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
		string str;
		ifstream f;
		f.open("abc.txt");
		while(getline(f,str)){

				cout<<str<<endl;
		}

		f.close();
}
```



```
Ali 11
Aliana 22



--------------------------------
Process exited after 10.07 seconds with return value 0
Press any key to continue . . .
```

## Unformatted I/O

- When we need to store unformatted (raw) binary data (not text) in a file, we can make use of the following set of functions
- When performing binary operations on a file, we open it using the **ios::binary** mode specifier
- Although unformatted file functions can work on text files, some character translations may still occur

## Unformatted I/O

Unformatted Input/Output is the most basic form of input/output.

Unformatted input/output transfers the internal binary representation of the data directly between memory and the file

## Unformatted I/O

- The unformatted output functions (e.g., put(), write()) outputs the bytes as they are, without format conversion.

- In unformatting input, such as get(), getline(), read(), it reads the characters as they are, without conversion.

## Advantages and Disadvantages of Unformatted I/O

- Unformatted input/output is the simplest and most efficient form of input/output. It is usually the most compact way to store data. Unformatted input/output is the least portable form of input/output. Unformatted data files can only be moved easily to and from computers that share the same internal data representation.
- Unformatted input/output is not directly human readable, so you cannot type it out on a terminal screen or edit it with a text editor.

## Get/Put Functions

The functions get() and put() reads and writes a single character to a file, respectively

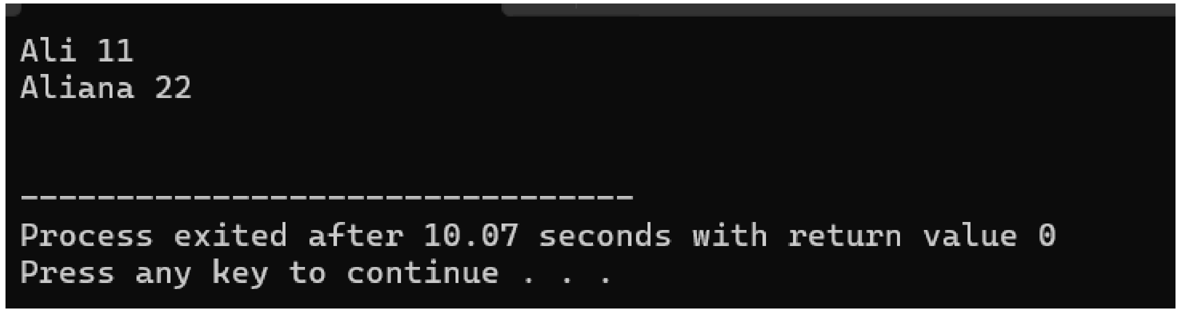
## get() Function

istream &get(char &*ch);*

The **get( ) function reads a single character from the invoking stream and puts that** value in *ch. It returns a reference to the stream*

## Example

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
 ifstream in(“myfile.txt");
          if(!in)   {
          cout << "Failed to open file" << endl;
          return 1;        }
          char c;
          while(in)        {
          in.get(c);
          cout << c;       }
 return 0;
}
```



```
Ali 11
Aliana 22



--------------------------------
Process exited after 10.07 seconds with return value 0
Press any key to continue . . .
```

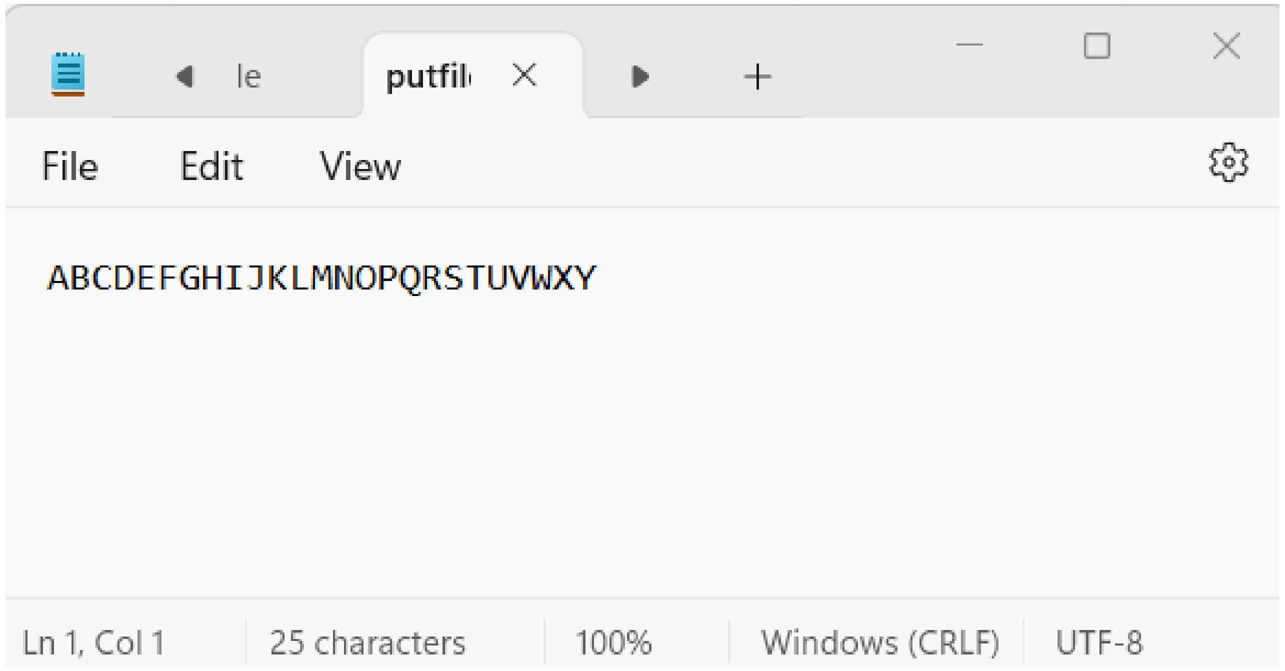
## put() Function

ostream &put(char *ch);*

*The* ***put( ) function writes ch to the*** stream and returns a reference to the stream

## Example

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
ofstream o(“putfile.txt");
 if(!o){
          cout << "Failed to open file" << endl;
          return 1;          }
          for(int i = 65; i < 90; i++){
          o.put(i);}
          o.close();
 return 0;
}
```



```
ABCDEFGHIJKLMNOPQRSTUVWXY
```

Ln 1, Col 1 | 25 characters | 100% | Windows (CRLF) | UTF-8

## TASK

Q: Design a C++ OOP program for a student marksheet system using file handling. The system should include a Student class that stores details like name, roll number, and marks in 3 subjects.

Use put() to write student data to a file and get() to read from the file. Demonstrate the complete flow with user input and file output.

| Feature | `get()/put()` | `read()/write()` |
|---|---|---|
| Works on | One character | Multiple bytes |
| Used with | Text files | Binary files |
| Usage | Reading names, lines, chars | Saving/loading whole structs or arrays |
| Format | Human-readable | Binary format |

## Read/Write Functions

- The functions **read()** and **write()** are similar to get() and put() except that we can read and write entire blocks of bytes (e.g. character array) in a file

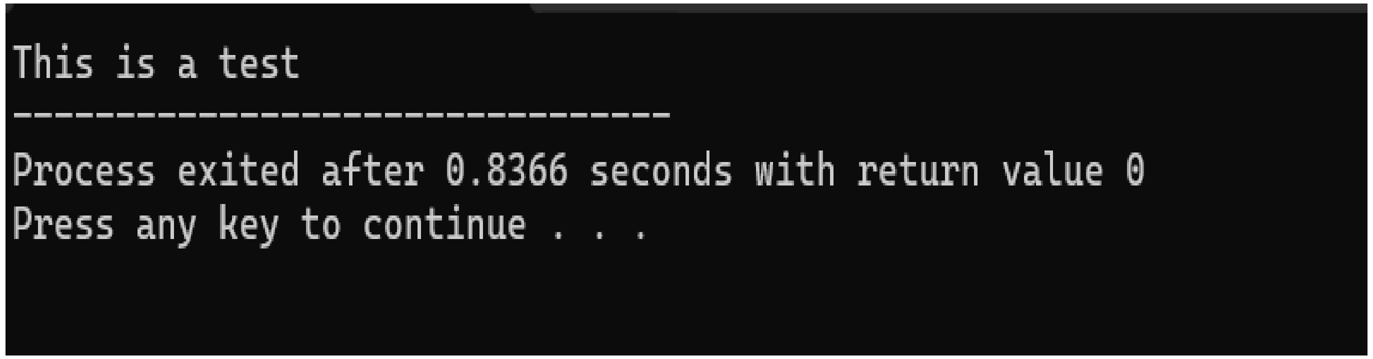
- **istream &read(char *buf, int num);**
- **ostream &write(const char *buf, int num);**

## Example

```
// Writing block (array) of characters
char chW[20] = {'T', 'h', 'i', 's', 'i', 's', 'a', 't','e','s','t'};
o.write(chW, 20);

// Reading block (array) of characters
char chR[20];
i.read(chR, 20);
for(int i = 0; i < 20; i++)
{
        cout << chR[i];
}
```

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
ofstream in("Test.txt");
if(!in){
                cout << "Failed to open file" << endl;
                return 1;       }
        char chW[20]="This is a test";
                in.write(chW, 20);
        return 0;
}
```

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){
fstream in("Test.txt");
                    char chR[20];
          in.read(chR, 20);
          for(int i = 0; i < 20; i++){
                    cout << chR[i];}
          return 0;
}
```



```
This is a test
--------------------------------
Process exited after 0.8366 seconds with return value 0
Press any key to continue . . .
```

## eof() Function

**bool eof( );**

You can detect when the end of the file is reached by using the member function **eof( )**

It returns *true* when the end of the file has been reached; otherwise it returns *false*

## Example 1

```
int main()
{
 ifstream i("myfile.txt");

 if(!i)
 {
		cout << "Failed to open file" << endl;
		return 1;
 }
 char c;
 for(int x = 0; x < 50; x++){

		i.get(c);
		cout << c;
 }

 return 0;
}
```

**myfile.txt contains text: "Hello"**

***Output:***

```
Helloooooo
```

# Example 2

```
int main() {
 ifstream in("myfile.txt");
 if(!in)
 {cout << "Failed to open file" << endl;
             return 1;
 }           char c;
 for(int x = 0; x < 10; x++)
 {if(in.eof())
                          break;
             in.get(c);
             cout << c;
 }return 0;
}
```

## File pointers

Every file maintains two pointers called

i) get_pointer (in input mode file) and

Ii) put_pointer (in output mode file)

which tells the current position in the file where reading or writing will takes place.

These pointers help attain random access in file.

That means moving directly to any location in the file instead of moving through it sequentially.

For example:

For example, if you have to modify a value in record no 21, then using random access techniques, you can place the file pointer at the beginning of record 21 and then straight-way process the record. If sequential access is used, then you'll have to unnecessarily go through first twenty records in order to reach at record 21.

## The seekg(), seekp(), tellg() and tellp() Functions

Random access is achieved by manipulating seekg(), seekp(), tellg() and tellp() functions.

The seekg() and tellg() functions allow you to set and examine the get_pointer, and the seekp() and tellp() functions perform these operations on the put_pointer.

The seekg() and tellg() functions are for input streams (ifstream) and seekp() and tellp() functions are for output streams (ofstream).

## tellg() and tellp()

The functions tellg() and tellp() return the position, in terms of byte number, of put_pointer and get_pointer respectively, in an output file and input file.

## Random Access

We can move the pointer (while reading or writing) to a specific position in a file

**istream &seekg(int *offset, origin);***

**ostream &seekp(int *offset, origin);***

Where origin can be any of the three following options

## Origin Options

```
ios::beg        // Beginning-of-file position
ios::cur        // Current location position
ios::end        // End-of-file position
```

fin.seekg(30); // will move the get_pointer (in ifstream) to byte number 30 in the file(starts from beg)

fout.seekp(30);// will move the put_pointer (in ofstream) to byte number 30 in the file

- It automatically points at the beginning of file, allowing us to read the file from the beginning.

```
fin.seekg(30, ios::beg);  // go to byte no. 30 from beginning of file
fin.seekg(-2, ios::cur);   // back up 2 bytes from the current position
fin.seekg(0, ios::end);    // go to the end of the file
fin.seekg(-4, ios::end);  // backup 4 bytes from the end of the file
```

## seekg() Function

- The **seekg( )** function moves the associated file's current pointer *offset the number* of characters from the specified *origin*

- The **seekg()** function is member of the *ifstream* class and is called through an *ifstream* object

- The function only moves the pointer ahead, the reading operation should then be performed through some other function

## Example

```
// Starting from the beginning, move position pointer five characters further

i.seekg(5, ios::beg);
char chR[20];
i.read(chR, 20);
for(int i = 0; i < 20; i++)
{
        cout << chR[i];
}
```

## seekp() Function

The **seekp()** function moves the associated file's current pointer *offset the number* of characters from the specified *origin*

The **seekp()** function is member of the *ofstream* class and is called through an *ofstream* object

The function only moves the pointer ahead, the writing operation should then be performed through some other function

## Example 1

***// Starting from the beginning, move position pointer two characters forward… then perform writing with put()***

```
char ch = 'K';
i.seekp(2, ios::beg);
i.put(ch);          // writes K at third position in the file
```

## Example 2

*// Starting from the end, move position pointer three characters backwards… then perform writing with put()*

```
char ch = ‘J’;
i.seekp(-3, ios::end);
i.put(ch);            // writes J at fourth from last position
```

## Further Examples in Dev File