UNIT 6

Exception Handing and Stream I/O

Exception handling

- Exceptions are runtime anomalies or unusual condition that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory space or disk space.
- The exception handling is a mechanism to detect and report an "exceptional circumstances "at runtime, so that appropriate action can be taken. It provides a type-safe, integrated approach, for coping with the unusual predictable problems that arise while executing a program.

Types of Exceptions

- Exceptions are basically of two types namely, *synchronous* and *asynchronous* exceptions.
- Errors such as "*out of range index*" and "*over flow*" belongs to synchronous type exceptions.
- The errors that are caused by the events beyond the control of program (such as keyboard interrupts) are called *asynchronous* exceptions.
- The exception handling mechanism in C++ can handle only synchronous exceptions.

The exception handling mechanism suggests a separate error handling code that performs the following tasks.

1. Find the problem (Hit the exception)
2. Inform the error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take the corrective action (Handle the exception)

The error handling code mainly consists of two segments, one to detect error and throw exceptions and other to catch the exceptions and to take appropriate actions.

C++ exception handling mechanism is basically built upon three keywords namely try, throw and catch.
1. The keyword **try** is used to **preface** a block of statements (surrounded by braces) which may generate exception. This block of statement is known as try block.
2. When as exception is detected, it is thrown using a **throw** statement in the try block.
3. A catch block defined by the keyword **catch,** catches the exception thrown by the throw statement in the try block and handles it appropriately.

```
try
{
. . . . . .
//block of statements which detects and throw an exceptions

throw exception;
}
catch(type arg ) //catch the exception
{
// Block of statements that handles the exceptions
}
```

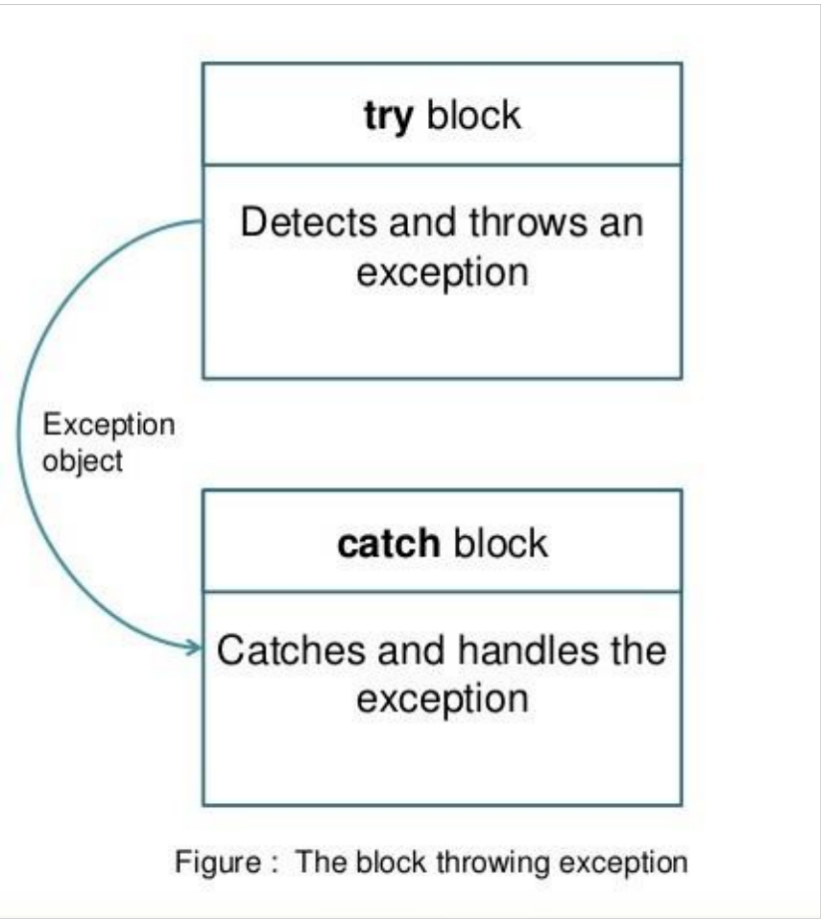


Figure : The block throwing exception

When the try block throws an exception, the program control leaves the try block and enters the catch statement of the catch block. If the type of object thrown matches the arg type in the catch statement, then the catch block is executed for handling the exception. If they do not match, the program is aborted with the help of *abort*() function which is executed implicitly by the compiler. When no exception is detected and thrown, the control goes to the statement immediately after the catch blocks That is ,catch block is skipped.

Program:

```
#include<iostream>
using namespace std;
int main()
{
int a,b,x;
cout<<"Enter values of a and b"<<endl;
cin >> a>>b;
x=a-b;
try
{
if(x!=0)
{
cout << "Result (a/x)="<< a/x <<endl;
}
else
{
throw(x);        //throws an object
}
}
catch(int i)       //catches an exception
{
cout<< "Exception caught:DIVIDE BY ZERO"<<endl;
}
cout << "END";
return 0;
}
```

Output:

First Run:

```
Enter values of a and b
20
15
Result (a/x)=4
END
```

Second Run:

```
Enter values of a and b
10
10
Exception caught:DIVIDE BY ZERO
END
```

Multiple catch statements (Multiple Exception Handling)

It is possible that a program segment has more than one condition to throw an exception.
In such cases, we can associate more than one catch statement with a try (much like conditions in **switch** statement) as shown below.

```
try
{
//try block
}
catch(type1 arg)
{
//catch block1
}

catch(type2 arg)
{
//catch block2
}
. . . . .. . . . . . .
catch(typeN arg)
{
Catch block N
}
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last catch block for that try. When no match is found, the program is terminated.

It is possible that arguments of several catch statements match the type of an exception. In such cases the first handler that matches the exception type is executed.

Example of program where multiple catch statements are used to handle various types of exceptions.

Write a program that catches multiple exceptions. [PU:2016 spring]

```
#include<iostream>
using namespace std;
void test(int x)
{
	try
	{
		if(x==1)
		throw x;
		else if(x==0)
		throw 'x';
		else if (x==-1)
		throw 1.0;
		cout<<"End of try-block"<<endl;
	}
	catch(char c)
	{
	cout<<"caught a character"<<endl;
	}
	catch(int m)
	{
	 cout<<"Caught an integer"<<endl;
	}
	catch(double d)
	{
	 cout<<"Caught a double"<<endl;
	}
	cout<<"End of try-catch system"<<endl;
}
int main()
{
	cout<<"Testing multiple catches"<<endl;
	cout<<"x==1"<<endl;
	test(1);
	cout<<"x==0"<<endl;
	test(0);
	cout<<"x==-1"<<endl;
	test(-1);
	cout<<"x==2"<<endl;
	test(2);
	return 0;
}
```

Output:

```
Testing multiple catches
x==1
Caught an integer
End of try-catch system
x==0
caught a character
End of try-catch system
x==-1
Caught a double
End of try-catch system
x==2
End of try-block
End of try-catch system
```

Error and Exceptions

Error

- ✓ Errors are problems or mistakes that occur during the execution of a program. They can prevent the program from running correctly or cause unexpected behavior.
- ✓ Errors can be broadly categorized into two types:
  Compile-Time Errors: These errors are detected by the compiler while translating the source code into machine-executable code. They are caused by incorrect syntax, type mismatches, missing or incorrect declarations, and other issues that violate the rules of the programming language. Programs with compile-time errors cannot be successfully compiled.
  Run-Time Errors: These errors occur while the program is running. They are caused by unexpected conditions that arise during execution, such as division by zero, accessing invalid memory locations, or trying to perform operations that are not allowed. Run-time errors can lead to program crashes or incorrect results.
- ✓ It is not possible to recover from an error.

Exceptions

- ✓ Exceptions are runtime anomalies or unusual condition that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory space or disk space.
- ✓ Exceptions are basically of two types namely, *synchronous* and *asynchronous* exceptions.
  - ❖ Errors such as "*out of range index*" and "*over flow*" belongs to synchronous type exceptions.
  - ❖ The errors that are caused by the events beyond the control of program (such as keyboard interrupts) are called *asynchronous* exceptions.
- ✓ The exception handling mechanism in C++ can handle only synchronous exceptions.

C++ exception handling mechanism is basically built upon three keywords namely try, throw and catch.

1. The keyword **try** is used to **preface** a block of statements (surrounded by braces) which may generate exception. This block of statement is known as try block.
2. When as exception is detected, it is thrown using a **throw** statement in the try block.
3. A catch block defined by the keyword **catch,** catches the exception thrown by the throw statement in the try block and handles it appropriately.

```
try
{
. . . . . .
//block of statements which detects and throw an exceptions

throw exception;
}
catch(type arg ) //catch the exception
{
// Block of statements that handles the exceptions
}
```

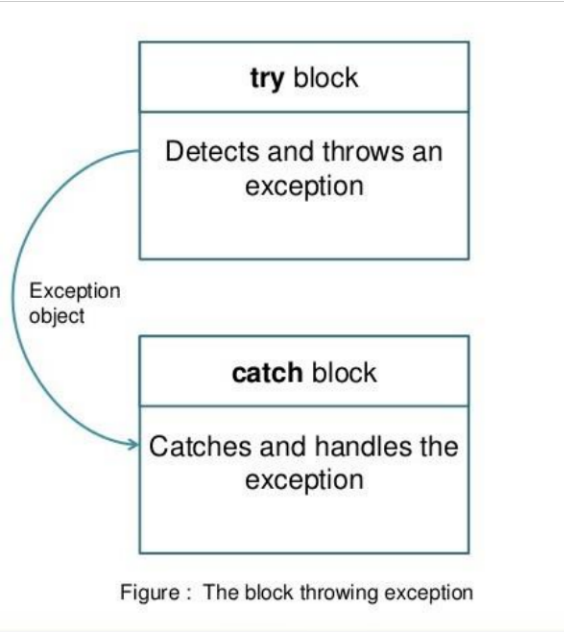


Figure : The block throwing exception

File Handling

- ✓ Many real-life problems handle large volumes of data and in such situations, we need to use some devices such as floppy disk or hard disk to store the data. The data is stored in these devices using the concept of files.
- ✓ A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files.

A program typically involves either or both of the following kinds of data communication:

1. Data transfer between the console unit and the program
2. Data transfer between the program and a disk file

- ✓ The I/O system of C++ handles file operations and uses files streams as an interface between the programs and files.
- ✓ The stream that supplies data to the program is known as input stream and one that receives data from the program is known as output stream.
- ✓ In other words, the input stream extracts(reads) data from the file and the output stream inserts (or writes) data to the file. This is illustrated in figure:

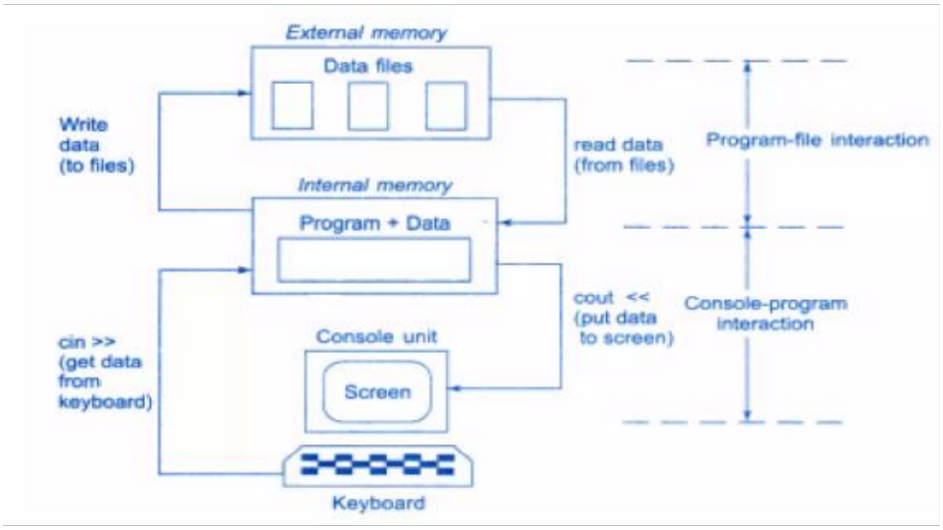


Figure: console-program-file interaction

The input operation involves the creation of an input stream and linking it with the program and the output file. Similarly, the output operation involves establishing an output stream with necessary links with the program and the output file.

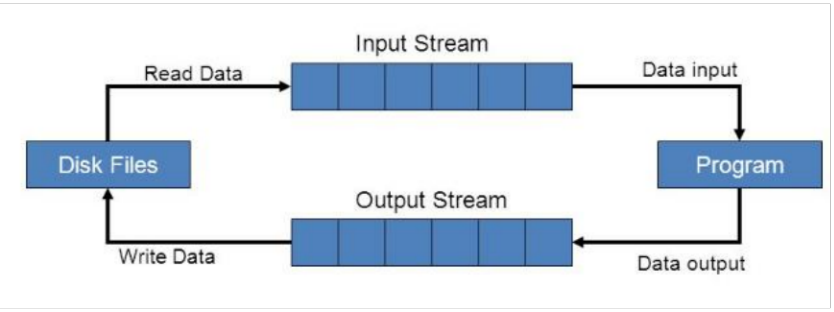
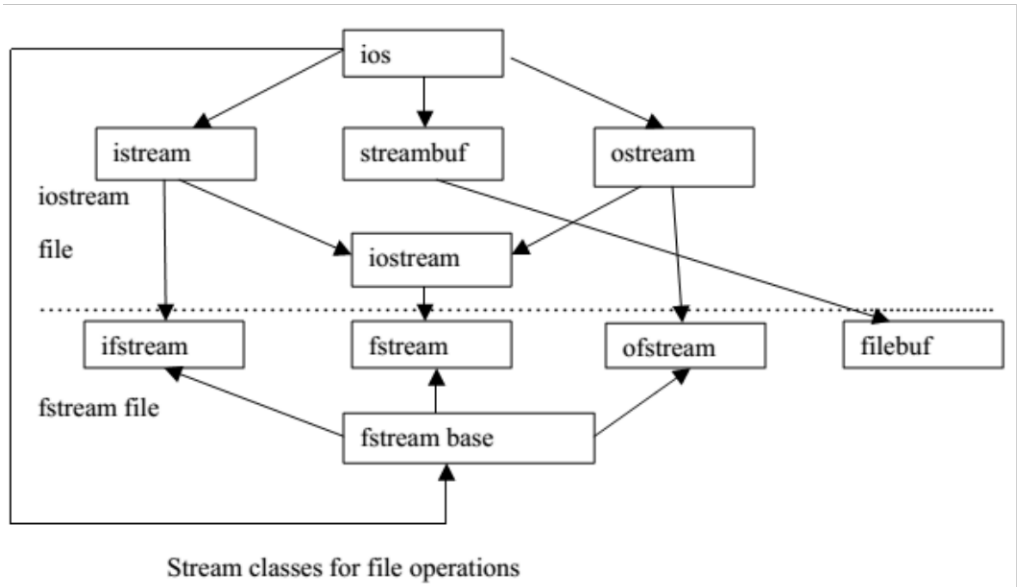


Figure: File input and output streams

Classes for file stream operations

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

Stream class hierarchy



Stream classes for file operations

The table below shows the details of file stream classes.

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**.

OPENING AND CLOSING A FILE

If we want to use a disk file, we need to decide the following things about the file and its intended use.

1. Suitable name for the file
2. Data type and structure
3. Purpose
4. Opening method

The filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with extension.

Examples:

Input.data
Test .doc
student
For opening a file, we must first create a file stream and then link it to the filename. A file stream can be defined using the classes ifstream, ofstream and fstream that are contained in the header file fstream. The class to be used depends upon the purpose that is, whether we want to read data from the file or write data to it. A file can be opened in two ways.

1. Using constructor function of the class
2. Using member function **open()** of the class

- ✓ The first method is useful when we use only one file in file stream.
- ✓ The second method is used when we want to manage multiple files using one stream.

Opening files using constructor

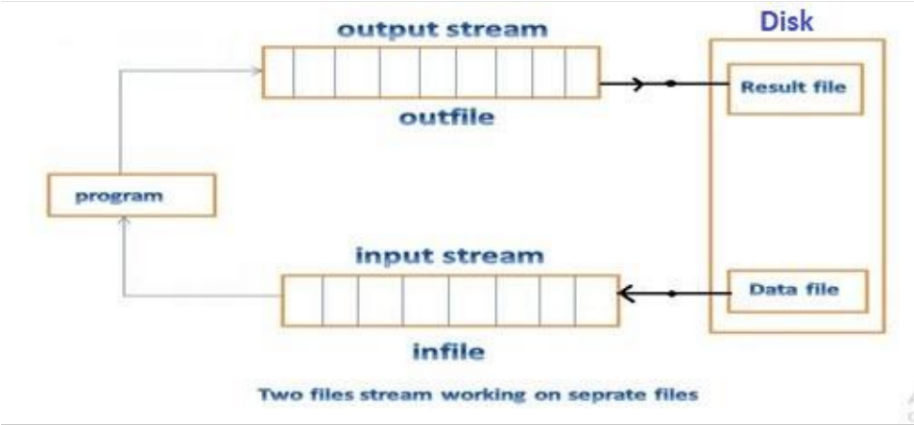
We know that a constructor is used to initialize an object while it is being created. Here, a filename is used to initialize file stream object. This involves the following steps:

1. Create a file stream object to manage the stream using the appropriate class. That is to say, the class **ofstream** is used to create the output stream and the class **ifstream** to create the input stream.
2. Initialize the file object with the desired filename.

   For example, the following statement opens a file named "results" for output.

   ofstream outfile("results"); //output only

   - ✓ This creates **outfile** as an **ofstream** object that manages the output stream. This object can be any valid C++ name such as o_file,myfile,fout.
   - ✓ This statement also opens the file **results** and attaches it to the output stream **outfile**. This is illustrated in figure below.



   Similarly, the following statement declares **infile** as an **ifstream** object and attaches to it to the file data for reading input.

ifstream infile("data"); //input only

The program may contain statements like
outfile<<"Total";
outfile<<sum;
infile>>number;
infile>>string;

We can also use the same file for both reading and writing data as shown in figure. The program would contain the following statements.

Program 1:

. . . . . .

. . . . . .

ofstream outfile(“salary”); //creates outfile and connects ”salary” to it

. . . . . .

. . . . . .

Program2

. . . . . .

. . . . . .

ifstream infile(“salary”); //creates infile and connects salary to it

. . . . . .

. . . . . .

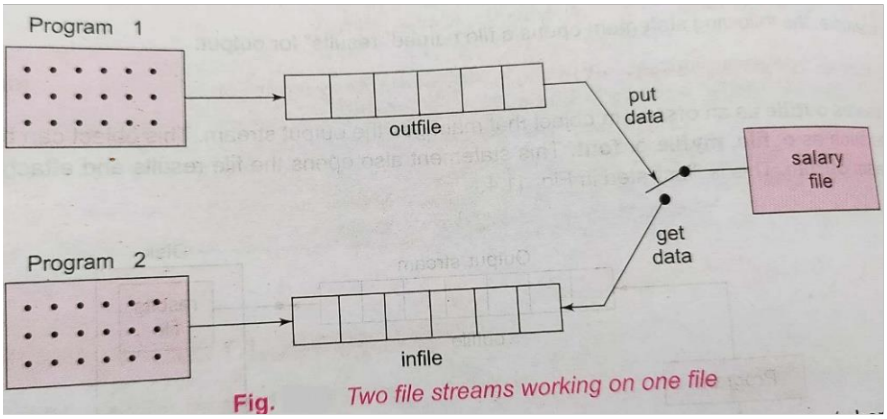


Fig. Two file streams working on one file

The connection with a file is closed automatically when the stream object expires (when the program terminates). In the above statement, when the program1 is terminated, salary file is disconnected from **outfile** stream. Similar action takes place when program 2 terminates.

Instead of using two programs, one for writing data(output) and another for reading data(input) ,we can use a single program to do both operations on the file. Example

. . . . . .

. . . .

outfile.close(); //Disconnect salary from outfile

ifstream infile(“salary”); // and connect to infile

. . . . . .

. . . . . .

infile.close(); //Disconnect salary from infile

Although we have used a single program ,we created two file stream objects, **outfile** (to put data into the file) and **infile(**to get data from the file). Note that the use of a statement like

outfile.close();

disconnects the file salary from the output stream **outfile.** Remember, the object outfile still exists and the salary file may again to be connected to **outfile** later or any other stream. In this example, we are connecting the salary file to **infile** stream to read data.

Program below uses a single file for both writing and reading to data. First it takes data from the keyboard and writes it to the file. After writing is completed, the file is closed. The program again opens the same file and reads the information already written to it and displays the same on the screen.

Working with Single File

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
        ofstream outfile("ITEM");
        char name[20];
        cout<<"Enter item name:";
        cin>>name;
        outfile<<name<<endl;
        float cost;
        cout<<"Enter the item cost:";
        cin>>cost;
        outfile<<cost<<endl;
        outfile.close();
        ifstream infile("ITEM");
        infile>>name;
        infile>>cost;
        cout<<"Item name:"<<name<<endl;
        cout<<"Item cost:"<<cost<<endl;
        infile.close();
        return 0;
}
```

output:

```
Enter item name:CD-ROM
Enter the item cost:250
Item name:CD-ROM
Item cost:250
```

Opening files using Open()

The function open() can be used to open multiple files that use the same stream object. For example, we may want to process a set of files sequentially. In such cases, we may create a single file stream object and use it to open each file in turn. This is done as follows:

```
file-stream-class stream-object;
stream-object.open("file name");
```

Example:

```
ofstream outfile;
outfile.open("DATA1");
. . . . . .
. . . . . .
outfile.close();
outfile.open("DATA2");
. . . . . .
. . . . . . . . .
outfile.close();
. . . . . .
. . . . . .. . .
```

The previous program segment opens two files in sequence for writing the data. Note that the first file is closed before opening the second file. This is necessary because a stream can be connected to only one file at a time.

Working with Multiple Files

Write a C++ program to illustrate reading and writing into multiple files.

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
	ofstream fout;
	fout.open("country");
	fout<<"USA"<<endl;
	fout<<"Japan"<<endl;
	fout<<"Nepal"<<endl;
	fout.close();
	fout.open("capital");
	fout<<"Washington"<<endl;
	fout<<"Tokyo"<<endl;
	fout<<"Kathmandu"<<endl;
	fout.close();

	//Reading the files
	char  line[50];
	ifstream fin;
	fin.open("country");
	cout<<"contents of country file"<<endl;
	while(fin)
	{
		fin.getline(line,50);
		cout<<line<<endl;
	}
	fin.close();
	fin.open("capital");
	cout<<"contents of capital file"<<endl;
	while(fin)
	{
		fin.getline(line,50);
		cout<<line<<endl;
	}
	fin.close();
	return 0;
}
```

The output of program would be:

```
contents of country file
USA
Japan
Nepal

contents of capital file
Washington
Tokyo
Kathmandu
```

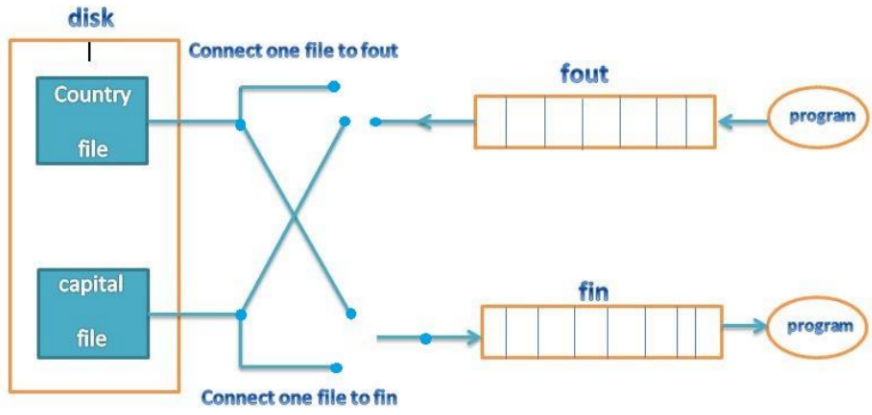


Fig: Streams working on multiple files

DETECTING END-OF-FILE

Detection of end-of-file condition is necessary for preventing any further attempt to read data from the file.
This was illustrated in program by using the statement
while(fin);
An ifstream object, such as fin, returns a value 0 if any error occurs in the file operation including the end-of-file condition. Thus, the while loop terminates when fin returns a value zero on reaching the end-of-file condition.

There is another approach to detect the end-of-file condition. Note that we have used the following statement in the program.

```
if(fin.eof()!=0)
{
exit(1);
}
```

eof() is a member function of ios class. It returns a non-zero value if the end-of-file(EOF) condition is encountered and a zero otherwise. Therefore, the above statement terminates the program on reaching the end of file.

More About Open (): FILE MODES

The function open() can take two arguments.

General form:

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

The second argument mode(called file mode parameter) specifies the purpose for which the file is opened.
In previous examples we open the files without providing second argument , this is possible because prototype of these class member functions contain default values for the second argument and therefore they use the default values in the absence of actual values. The default values are as follows.
ios::in for ifstream functions meaning open for reading only
ios::out for ofstream functions meaning open for writing only.

The file mode parameter can take one(or more) of such constants defined in the class ios. Table lists the file mode parameter and their meanings.

Parameter	Meaning
ios::app	Append to end-of-file
ios::ate	Go to end-of-file on opening
ios::binary	Binary file
ios::in	Open file for reading only
ios::nocreate	Open fails if the file does not exist
ios::noreplace	Open fails if the file already exists.
ios::out	Open file for writing only
ios::trunc	Delete the contents of the file if it exists

Functions for manipulation of File pointers

We can move the file pointers in the desired position which means we can control the movement of file pointer ourselves. The file stream classes support the following functions to manage such situations.

seekg() Moves get pointer(input) to a specified location
seekp() Moves put pointer(output) to a specified location
tellg() Gives the current position of the get pointer
tellp() Gives the current position of the put pointer

For example, the statement:

infile.seekg(10);
moves the file pointer to the byte number 10.Bytes in file are numbered beginning from zero. Therefore ,the pointer will pointing to the 11^{th} byte in the file.

Sequential input and output operations

The file stream class supports a number of member functions for performing the input and output operations on files. One pair of functions, put() and get() ,are designed for handling a single character at a time. Another pair of functions, write() and read() are designed to write and read blocks of binary data.

Write () and read () functions ()

The functions **write ()** and **read()** handle the data in binary form. This means that the values stored in the disk file in same format in which they are stored in internal memory.

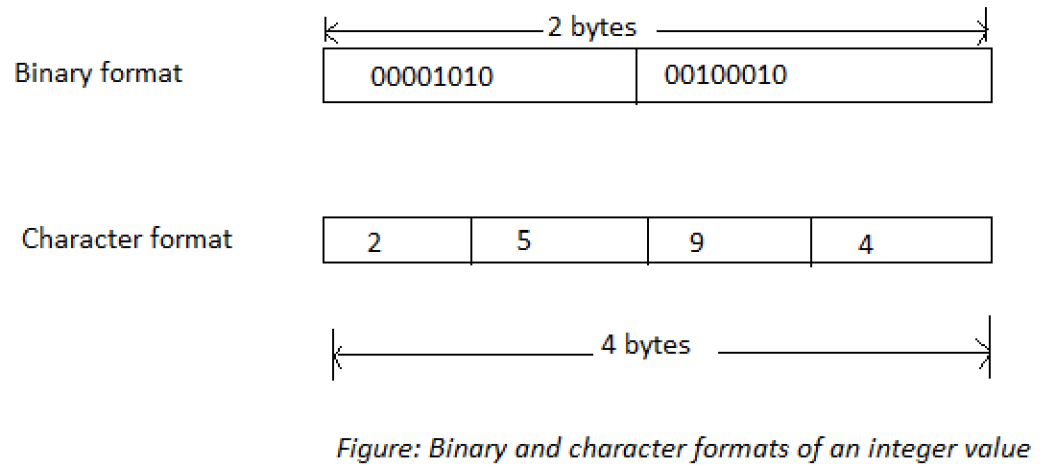


Figure: Binary and character formats of an integer value

Figure shows how an **int** value 2594 is stored in the binary and character formats. An int takes two bytes to store its value in the binary form, irrespective of its size. But a 4-digit int will take four bytes to store it in character form.
The binary format is more for storing the numbers as they are stored in the exact internal representation. There are no conversions while saving the data and therefore saving is much faster.

```
infile.read((char *) &V, sizeof(V));
outfile.write((char*) &V, sizeof(V));
```

These functions takes two arguments. The first is the address of the variable V, second is the length of variable in bytes. The address of the variable must be cast to type **char* (i.e pointer to character type).**

Program illustrates how these two functions are used to save an array of float numbers and then recover them for display on the screen.

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

int main()
{
        float height[4]={175.5,152.5,67.25,160.7};
        ofstream outfile;
        outfile.open("Binary");
        outfile.write((char*)&height,sizeof(height));
        outfile.close();    //close the file for reading

        for(int i=0;i<4;i++)    //clear array from memory
        {
                height[i]=0;
        }
        ifstream infile;
        infile.open("Binary");
        infile.read((char*)&height,sizeof(height));
        for(int i=0;i<4;i++)
{
        cout<<height[i]<<endl;
}
infile.close();
return 0;
}
```

Reading and writing a class object

One of the shortcomings of I/O system of C is that it cannot handle user-defined data types such as class objects. Since the class objects are the central elements of C++ programming. It is quite natural that the language supports features for writing to and reading from the disk files objects directly. The
binary input and output functions **read ()** and **write ()** are designed to do exactly this job. These functions handle the entire structure of an object as a single unit, using computer internal representation of data. For instance, the function write () copies a class object from memory byte by byte with no conversion. One important point to remember is that only data members are written to the disk file and member functions are not.
Program illustrates how class objects can be written to read from the disk files. The length of the object is obtained using the **sizeof** operator. This length represents the sum of lengths of all data members of object.

```
#include<iostream>
#include<fstream>
#include<iomanip>
using namespace std;
class Inventory
{
	private:
	char name[10];
	int code;
	float cost;
	public:
		void readata()
		{
			cout<<"Enter name:";
			cin>>name;
			cout<<"Enter code:";
			cin>>code;
			cout<<"Enter cost:";
			cin>>cost;

		}
		void writedata()
		{
		cout<<"Name:"<<name<<endl;
		cout<<"Code:"<<code<<endl;
		cout<<"Cost:"<<cost<<endl;
		}
};
int main()
{
	Inventory item[3];
	fstream file;
	file.open("stock.dat",ios::in|ios::out);
	cout<<"Enter details for three items"<<endl;
	for(int i=0;i<3;i++)
	{
		item[i].readata();
		file.write((char*)&item[i],sizeof(item[i]));
	}
	file.seekg(0);
	cout<<"Details of item stored in file are:"<<endl;
	for(int i=0;i<3;i++)
	{
		file.read((char*)&item[i],sizeof(item[i]));
		item[i].writedata();
	}
	file.close();
	return 0;
}
```

Previous old Questions from this chapter

1) What is exception handling? Discuss briefly.[PU:2005 fall]
2) What is exception? Explain the method of exception handling in C++?[PU:2019 fall]
3) What is exception? Define the type of exceptions. Explain about Exception handling mechanism in C++.[PU:2013 spring]
4) What is exception? What is the syntax for exception handling in C++.Write a program that catches multiple exceptions.[PU:2016 spring]
5) Explain about all Exception Handling constructs. With suitable example explain multiple exceptions handling in C++.
6) Write a short notes on:

- Exception handling[PU:2009 spring][PU:2014 spring][PU:2014 fall] [2018 fall]
- Exception mechanism[PU:2017 spring]

Model questions for File handling

- Discuss stream class hierarchy. How a file can be open in C++? Explain with suitable examples and syntax. Write a program to write information of 10 employee in file and display their details in console.
- Write a program to store detail of a book (id, author, price) in a file. And display the stored. Information from file. Use the object-oriented approach for programming.
- Write a program to illustrate reading and writing into multiple files.
  [Hint use open () method]
- Create a class employee with data members empid, name, address, salary and read the information of 10 employees and write to the file named “empinfo” and finally read information from file and display on the screen.