Unit 5
File Handling and Exceptions

Files

All the programs we have looked at so far use input only from the keyboard, and output only to the screen. If we were restricted to use only the keyboard and screen as input and output devices, it would be difficult to handle large amounts of input data, and output data would always be lost as soon as we turned the computer off. To avoid these problems, we can store data in some secondary storage device, usually magnetic tapes or discs. Data can be created by one program, stored on these devices, and then accessed or modified by other programs when necessary. To achieve this, the data is packaged up on the storage devices as data structures called files. A file is a container for data. Think of a file as a drawer in a file cabinet which needs to be opened and closed. Before you can put something into a file, or take something out, you must open the file (the drawer). When you are finished using the file, the file (the drawer) must be closed. The easiest way to think about a file is as a linear sequence of characters.

Streams

Before we can work with files in C++, we need to become acquainted with the notion of a stream. We can think of a stream as a channel or conduit on which data is passed from senders to receivers. As far as the programs we will use are concerned, streams allow travel in only one direction. Data can be sent out from the program on an output stream, or received into the program on an input stream. For example, at the start of a program, the standard input stream "cin" is connected to the keyboard and the standard output stream "cout" is connected to the screen. In fact, input and output streams such as "cin" and "cout" are examples of (stream) objects. So learning about streams is a good way to introduce some of the syntax and ideas behind the object-oriented part of C++. The header file which lists the operations on streams both to and from files is called "fstream". We will therefore assume that the program fragments discussed below are embedded in programs containing the "include" statement #include As we shall see, the essential characteristic of stream processing is that data elements must be sent to or received from a stream one at a time, i.e. in serial fashion.

Creating a Sequential Access File

Steps to create (or write to) a sequential access file:

1. Declare a stream variable name:

ofstream fout; //each file has its own
stream buffer
ofstream is short for output file stream
fout is the stream variable name
Naming the stream variable "fout" is helpful in remembering that the information is going "out"
to the file.

2. Open the file:

fout.open("scores.dat", ios::out);
fout is the stream variable name previously declared "scores.dat" is the name of the file
ios::out is the steam operation mode (your compiler may not require that you specify the stream
operation mode.)

3. Write data to the file:

```
fout<<grade<<endl;
fout<<"Mr. Spock\n";
```
The data must be separated with space characters or end-of-line characters (carriage return), orthe data will run together in the file and be unreadable. Try to save the data to the file in the same manner that you would display it on the screen.
If the iomanip.h header file is used, you will be able to use familiar formatting commands with
file output.
```
fout<<setprecision(2);
fout<<setw(10)<<3.14159;
```

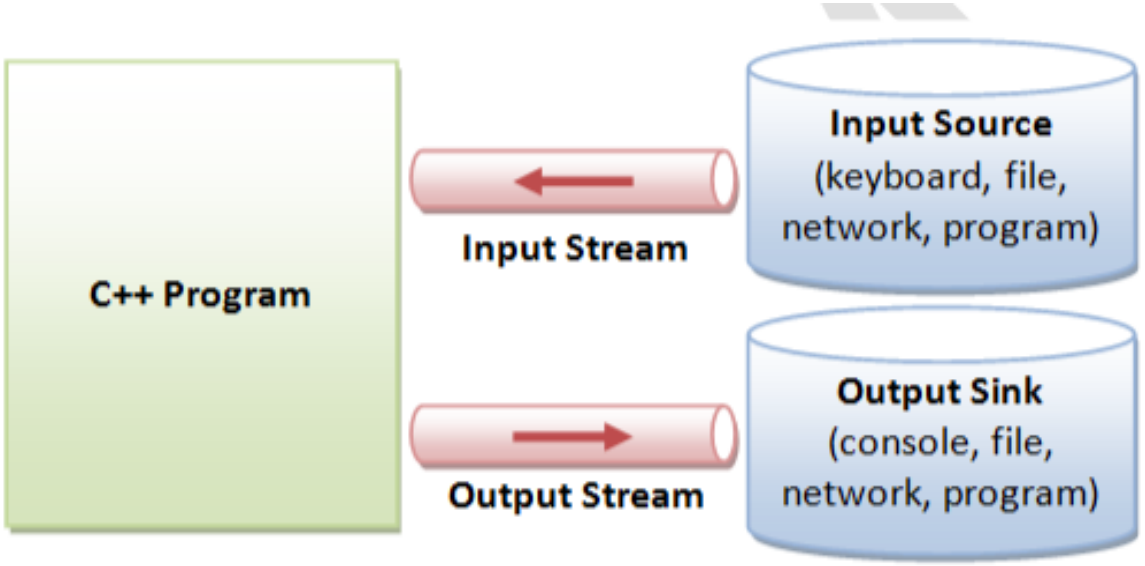
4. Close the file:
```
fout.close( );
```
Closing the file writes any data remaining in the buffer to the file, releases the file from the program, and updates the file directory to reflect the file's new size. As soon as your program is finished accessing the file, the file should be closed. Most systems close any data files when a program terminates. Should data remain in the buffer when the program terminates, you may lose that data.

Reading from a sequential-access file
To read data from a file you need one FileStream object and one StreamReader object. The StreamReader object accepts the FileStream object as its argument.
```
 FileStream fs;
StreamReader fr;
//create file stream object
fs = new FileStream("D:\\test.txt", FileMode.Open , FileAccess.Read );
//create reader objec
fr=new StreamReader(fs);
string content=fr.ReadLine() ;
while (!fr.EndOfStream )
{
Console.WriteLine(content);
content = fr.ReadLine();
}
//close FileStream object
fs.Close();
```

Updating a Sequential File
The data in many sequential files cannot be modified without the risk of destroying other data in the file.
If the name “White” needed to be changed to “Worthington,” the old name cannot simply be overwritten, because the new name requires more space.
Fields in a text file — and hence records — can vary in size.
Records in a sequential-access file are not usually updated in place. Instead, the entire file is usually rewritten.

Stream I/O
Streams
C/C++ IO are based on streams, which are sequence of bytes flowing in and out of the programs

(just like water and oil flowing through a pipe). In input operations, data bytes flow from an input source (such as keyboard, file, network or another program) into the program. In output operations, data bytes flow from the program to an output sink (such as console, file, network or another program). Streams acts as an intermediaries between the programs and the actual IO devices, in such the way that frees the programmers from handling the actual devices, so as to archive device independent IO operations.



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, which presents a consistent public IO interface.
To perform input and output, a C++ program:
1. Construct a stream object.
2. Connect (Associate) the stream object to an actual IO device (e.g., keyboard, console, file, network, another program).
3. Perform input/output operations on the stream, via the functions defined in the stream's pubic interface in a device independent manner. Some functions convert the data between the external format and internal format (formatted IO); while other does not (unformatted or binary IO).
4. Disconnect (Dissociate) the stream to the actual IO device (e.g., close the file).
5. Free the stream object.

File Input/Output (Header <fstream>)
C++ handles file IO similar to standard IO. In header <fstream>, the class ofstream is a subclass of ostream; ifstream is a subclass of istream; and fstream is a subclass of iostream for bidirectional IO. You need to include both <iostream> and <fstream> headers in your program for file IO.

To write to a file, you construct a ofsteam object connecting to the output file, and use the ostream functions such as stream insertion <<, put() and write(). Similarly, to read from an input file, construct an ifstream object connecting to the input file, and use the istream functions such as stream extraction >>, get(), getline() and read().

File IO requires an additional step to connect the file to the stream (i.e., file open) and disconnect from the stream (i.e., file close).

File Output

The steps are:

1. Construct an ostream object.
2. Connect it to a file (i.e., file open) and set the mode of file operation (e.g, truncate, append).
3. Perform output operation via insertion >> operator or write(), put() functions.
4. Disconnect (close the file which flushes the output buffer) and free the ostream object.

```
#include <fstream>
.......
ofstream fout;
fout.open(filename, mode);
......
fout.close();

// OR combine declaration and open()
ofstream fout(filename, mode);
```

By default, opening an output file creates a new file if the filename does not exist; or truncates it
(clear its content) and starts writing as an empty file.

open(), close() and is_open()

```
void open (const char* filename,
 ios::openmode mode = ios::in | ios::out);
 // open() accepts only C-string. For string object, need to use c_str() to get the C-string

void close (); // Closes the file, flush the buffer and disconnect from stream object

bool is_open (); // Returns true if the file is successfully opened
```

File Modes

File modes are defined as static public member in ios_base superclass. They can be referenced
from ios_base or its subclasses - we typically use subclass ios. The available file mode flags are:

1. ios::in - open file for input operation
2. ios::out - open file for output operation
3. ios::app - output appends at the end of the file.
4. ios::trunc - truncate the file and discard old contents.
5. ios::binary - for binary (raw byte) IO operation, instead of character-based.
6. ios::ate - position the file pointer "at the end" for input/output.

You can set multiple flags via bit-or (|) operator, e.g., ios::out | ios::app to append output at the end of the file.

For output, the default is ios::out | ios::trunc. For input, the default is ios::in.

File Input

The steps are:

1. Construct an istream object.

2. Connect it to a file (i.e., file open) and set the mode of file operation.
3. Perform output operation via extraction << operator or read(), get(), getline() functions.
4. Disconnect (close the file) and free the istream object.

```
#include <fstream>
.......
ifstream fin;
fin.open(filename, mode);
......
fin.close();

// OR combine declaration and open()
ifstream fin(filename, mode);
```

Unformatted Input/Output Functions

put(), get() and getline()

The ostream's member function put() can be used to put out a char. put() returns the invoking ostream reference, and thus, can be cascaded. For example,

```
// ostream class
ostream & put (char c); // put char c to ostream
// Examples
cout.put('A');
cout.put('A').put('p').put('p').put('\n');
cout.put(65);
```

Stream Manipulators

C++ provides a set of manipulators to perform input and output formatting:

1. <iomanip> header: setw(), setprecision(), setbas(), setfill().
2. <iostream> header: fixed|scientific, left|right|internal, boolalpha|noboolalpha, etc.

Exception Handling

Exceptions

Exceptions are run time anomalies or unusual condition that a program may encounter during execution.

Examples:

- Division by zero
- Access to an array outside of its bounds
- Running out of memory
- Running out of disk space

Principles of Exception Handling: Similar to errors, exceptions are also of two types. They are as follows:

- Synchronous exceptions: The exceptions which occur during the program execution due
- to some fault in the input data.
- For example: Errors such as out of range, overflow, division by zero
- Asynchronous exceptions: The exceptions caused by events or faults unrelated
- (external) to the program and beyond the control of the program.

For Example: Key board failures, hardware disk failures

The exception handling mechanism of C++ is designed to handle only synchronous exceptions within a program. The goal of exception handling is to create a routine that detects and sends an exceptional condition in order to execute suitable actions. The routine needs to carry out the following responsibilities:

1. Detect the problem (Hit the exception)
2. Inform that an error has been detected (Throw the exception)
3. Receive error information (Catch the exception)
4. Take corrective action (Handle the exception)

An exception is an object. It is sent from the part of the program where an error occurs to the part of the program that is going to control the error

The Keywords try, throw, and catch
Exception handling mechanism basically builds upon three keywords:

- try
- catch
- throw

The keyword try is used to preface a block of statements which may generate exceptions.
Syntax of try statement:

```
try
{
statement 1;
statement 2;
}
```

When an exception is detected, it is thrown using a throw statement in the try block.
Syntax of throw statement

- throw (excep);
- throw excep;
- throw; // re-throwing of an exception

A catch block defined by the keyword 'catch' catches the exception and handles it appropriately. The catch block that catches an exception must immediately follow the try block that throws the exception
Syntax of catch statement:

```
try
{
Statement 1;
Statement 2;
}
catch ( argument)
{
statement 3; // Action to be taken
}
```

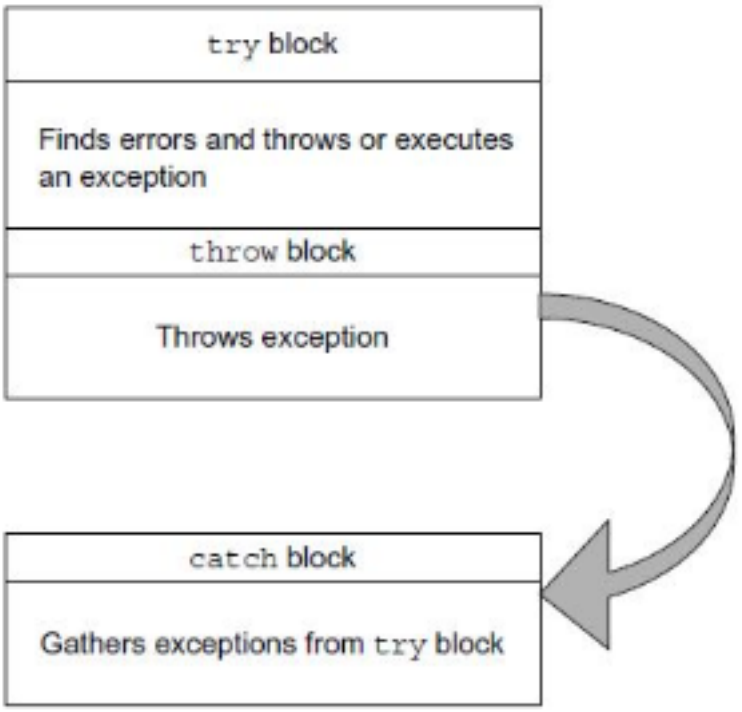
When an exception is found, the catch block is executed. The catch statement contains an argument of exception type, and it is optional. When an argument is declared,

the argument can be used in the catch block. After the execution of the catch block, the statements inside the blocks are executed. In case no exception is caught, the catch block is ignored, and if a mismatch is found, the program is terminated.

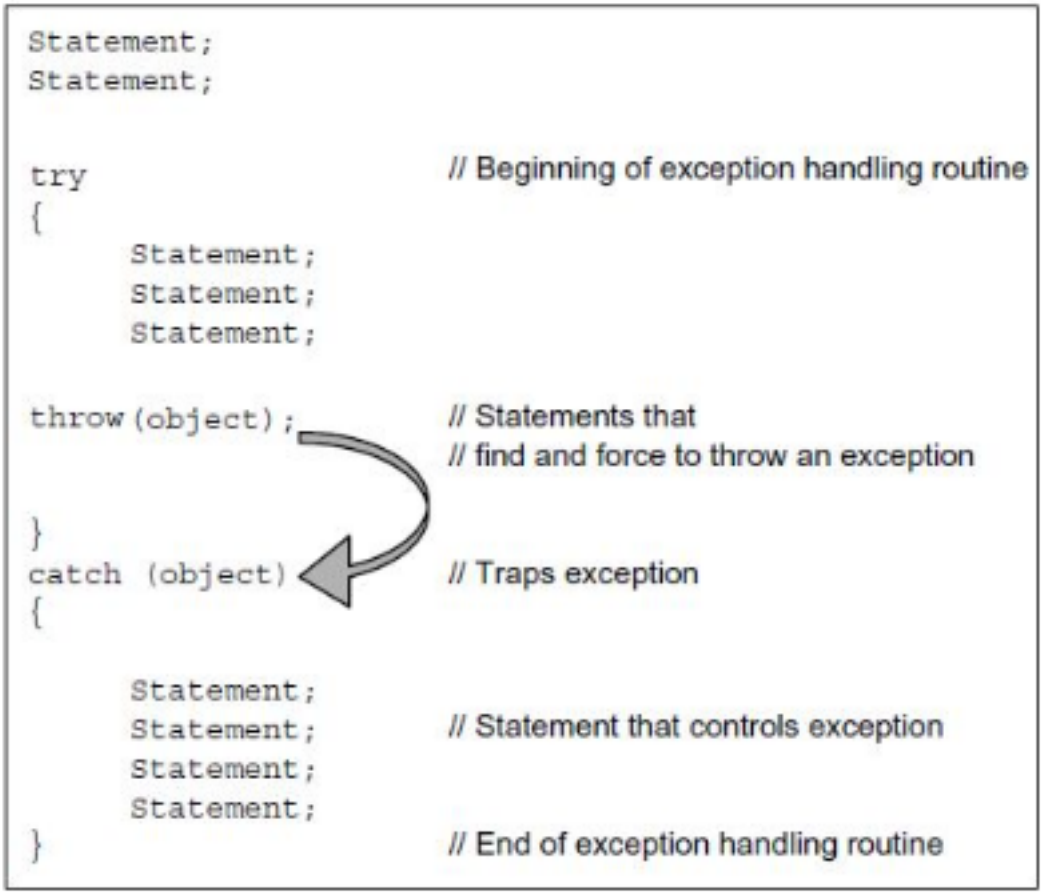
Guidelines for Exception Handling

The C++ exception-handling mechanism provides three keywords; they are try, throw, and catch. The keyword try is used at the starting of the exception. The throw block is present inside the try block. Immediately after the try block, the catch block is present.

Figure shows the try, catch, and throw statements.



As soon as an exception is found, the throw statement inside the try block throws an exception (a message for the catch block that an error has occurred in the try block statements). Only errors occurring inside the try block are used to throw exceptions. The catch block receives the exception that is sent by the throw block. The general form of the statement is as per the following figure.



When the try block passes an exception using the throw statement, the control of the program passes to the catch block. The data type used by throw and catch statements should be same; otherwise, the program is aborted using the abort() function, which is executed implicitly by the compiler. When no error is found and no exception is thrown, in such a situation, the catch block is disregarded, and the statement after the catch block is executed.

```
/* Write a program to illustrate division by zero exception. */
#include <iostream>
using namespace std;
int main()
{
int a, b;
cout<<"Enter the values of a and b"<<endl;
cin>>a>>b;
try{
if(b!=0)
cout<<a/b;
else
throw b;
}
catch(int i)
{
```

```
 cout<<"Division by zero: "<<i<<endl;
}
return 0;
}
```
Output:
```
Enter the values of a and b
2
0
Division by zero: 0
```
```
/* Write a program to illustrate array index out of bounds exception. */
#include <iostream>
using namespace std;
int main() {
int a[5]={1,2,3,4,5},i;
try{
i=0;
while(1){
if(i!=5)
{
cout<<a[i]<<endl;
i++;
}
else
throw i;
}
}
catch(int i)
{
cout<<"Array Index out of Bounds Exception: "<<i<<endl;
}
return 0;
}
```
Output:
```
1
2
3
4
5
Array Index out of Bounds Exception: 5
```
```
/* Write a C++ program to define function that generates exception. */
#include<iostream>
using namespace std;
void sqr()
{
int s;
cout<<"\n Enter a number:";
```

```
cin>>s;
if (s>0)
{
cout<<"Square="<<s*s;
}
else
{
throw (s);
}
}
int main()
{
try
{
sqr();
}
catch (int j)
{
cout<<"\n Caught the exception \n";
}
return 0;
}
```

Ouput:

```
Enter a number:10
Square=100
Enter a number:-1
Caught the exception
```