File I/O

Dr.Purushothama B R
Computer Science and Engg
National Institute of Technology Goa

C Versus C++ File I/O

- C++ supports the entire Standard C file system.
- If you will be porting C code to C++, you will not have to change all of your I/O routines right away.
- C++ defines its own, object-oriented I/O system,
  - which includes both I/O functions and I/O operators
- The C++ I/O system completely duplicates the functionality of the C I/O system
- And, renders the C file system redundant.
- While you will usually want to use the C++ I/O system,
  - You are free to use the C file system if you like.

Streams and Files

- It is necessary to know the difference between the terms streams and files.
- The C I/O system supplies a consistent interface to the programmer independent of the actual device being accessed.
- That is, the C I/O system provides a level of abstraction between the programmer and the device.
- This abstraction is called a stream and the actual device is called a file.

Streams

- The C file system is designed to work with a wide variety of devices,
  - including terminals, disk drives, and tape drives.
- Even though each device is very different,
  - the file system transforms each into a logical device called a stream.
- All streams behave similarly.
- Because streams are largely device independent,
  - the same function that can write to a disk file can also be used to write to another type of device, such as the console.

- There are two types of streams: text and binary.

Text Streams

- A text stream is a sequence of characters.
- Standard C allows (but does not require) a text stream to be organized into lines terminated by a newline character.
- However, the newline character is optional on the last line.
- (Actually, most C/C++ compilers do not terminate text streams with newline characters.)

Text Stream

- In a text stream, certain character translations may occur as required by the host environment.
  - For example, a newline may be converted to a carriage return/linefeed pair.

- Therefore, there may not be a one-to-one relationship between the characters that are written (or read) and those on the external device.

- Also, because of possible translations, the number of characters written (or read) may not be the same as those on the external device.

Binary Streams

- A *binary stream* is a sequence of bytes that have a one-to-one correspondence to those in the external device.
  - That is, no character translations occur.
- Also, the number of bytes written (or read) is the same as the number on the external device.
- However, an implementation-defined number of null bytes may be appended to a binary stream.
- For example, these null bytes might be used to pad the information so that it fills a sector on a disk.

Files

- In C/C++, a file may be anything from a disk file to a terminal or printer.
- You associate a stream with a specific file by performing an open operation.
- Once a file is open, information may be exchanged between it and your program.
- Not all files have the same capabilities.
- For example, a disk file can support random access while some printers cannot.
- This brings up an important point about the C I/O system:
- All streams are the same but all files are not.

<fstream> and the File Classes

- To perform file I/O, you must include the header <fstream> in your program.
- It defines several classes, including **ifstream, ofstream, and fstream.**
- These classes are derived from **istream, ostream, and iostream, respectively.**
- Note , **istream, ostream, and iostream are derived from ios.**
- So ifstream, ofstream, and fstream also have access to all operations defined by ios.
- Another class used by the file system is filebuf, which provides low-level facilities to manage a file stream.
- Usually, you don't use filebuf directly, but it is part of the other file classes.

Opening and Closing a File

- In C++, you open a file by linking it to a stream.
- Before you can open a file, you must first obtain a stream.
- There are three types of streams: input, output, and input/output.
- To create an input stream, you must declare the stream to be of class **ifstream.**
- To create an output stream, you must declare it as class **ofstream.**
- Streams that will be performing both input and output operations must be declared as class fstream.

Example

- ifstream in; // input
- ofstream out; // output
- fstream io; // input and outp

- Once you have created a stream, one way to associate it with a file is by using open( ).

- This function is a member of each of the three stream classes.

The prototype

- **void ifstream::open(const char *filename, ios::openmode mode = ios::in);**
- **void ofstream::open(const char *filename, ios::openmode mode = ios::out | ios::trunc);**
- **void fstream::open(const char *filename, ios::openmode mode = ios::in | ios::out);**
  - Here, filename is the name of the file; it can include a path specifier.
  - The value of mode determines how the file is opened.

Value of Mode

- It must be one or more of the following values defined by **openmode,**
  - which is an enumeration defined by ios (through its base class ios_base).

  - **ios::app**
  - **ios::ate**
  - **ios::binary**
  - **ios::in**
  - **ios::out**
  - **ios::trunc**
- You can combine two or more of these values by **ORing** them together.

Value of Mode

- Including ios::app causes all output to that file to be appended to the end.
- This value can be used only with files capable of output.
- Including ios::ate causes a seek to the end of the file to occur when the file is opened.
- Although ios::ate causes an initial seek to end-of-file, I/O operations can still occur anywhere within the file.
- The ios::in value specifies that the file is capable of input.
- The ios::out value specifies that the file is capable of output.
- The ios::binary value causes a file to be opened in binary mode.

Note

- By default, all files are opened in text mode.
- In text mode, various character translations may take place, such as carriage return/linefeed sequences being converted into newlines.
- However, when a file is opened in binary mode, no such character translations will occur.
- Understand that any file, whether it contains formatted text or raw data, can be opened in either binary or text mode.
- The only difference is whether character translations take place.

Note

- The ios::trunc value causes the contents of a preexisting file by the same name to be destroyed, and the file is truncated to zero length.

- When creating an output stream using ofstream, any preexisting file by that name is automatically truncated.

Opening File for output

- The following fragment opens a normal output file.

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

- However, you will seldom see open( ) called as shown,
  - Because the mode parameter provides default values for each type of stream. As their prototypes show,
  - for ifstream, mode defaults to ios::in;
  - for ofstream, it is ios::out | ios::trunc; and
  - for fstream, it is ios::in | ios::out.

Open()

- If **open( )** fails, the stream will evaluate to false when used in a Boolean expression.
- Therefore, before using a file, you should test to make sure that the open operation succeeded.

```
if (!mystream) {
        cout << "Cannot open file.\n";
        // handle error
}
```

Open()

- Although it is entirely proper to open a file by using the **open( )** function,
  - Most of the time you will not do so because the **ifstream**, **ofstream**, and **fstream** classes have constructors that automatically open the file.
  - The constructors have the same parameters and defaults as the **open( )** function.

Most Common Way

- To open file for input

  ifstream mystream("myfile");

- You can also check to see if you have successfully opened a file by using the
  - **is_open( ) function**,
  - which is a member of fstream, ifstream, and ofstream.
- It has this prototype:

  bool is_open( );

It returns true if the stream is linked to an open file and false otherwise.

Example

```
if(!mystream.is_open()) {
        cout << "File is not open.\n";
          // ...
}
```

Closing a file

- To close a file, use the member function close( ).
- For example, to close the file linked to a stream called mystream,
- Use this statement:

  mystream.close();

- The close( ) function takes no parameters and returns no value.

Reading and Writing Text Files

- It is very easy to read from or write to a text file.
- Simply use the << and >> operators the same way you do when performing console I/O,
  - except that instead of using cin and cout,
  - substitute a stream that is linked to a file.

Example create short Inventory

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
        ofstream out("INVNTRY"); // output, normal file
           if(!out) {
                   cout << "Cannot open INVENTORY file.\n";
                   return 1;
            }
            out << "Radios " << 39.95 << endl;
            out << "Toasters " << 19.95 << endl;
            out << "Mixers " << 24.80 << endl;
            out.close();
            return 0;
}
```

Program to read the inventory file

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
        ifstream in("INVNTRY"); // input
        if(!in) {
                cout << "Cannot open INVENTORY file.\n";
                return 1;
        }
        char item[20];
        float cost;
        in >> item >> cost;
        cout << item << " " << cost << "\n";
        in >> item >> cost;
        cout << item << " " << cost << "\n";
        in >> item >> cost;
        cout << item << " " << cost << "\n";
        in.close();
        return 0;
}
```

Important

- When reading text files using the >> operator,
  - keep in mind that certain character translations will occur.
- For example, white-space characters are omitted.
- If you want to prevent any character translations,
  - you must open a file for binary access and use the different functions.
- When inputting, if end-of-file is encountered,
  - the stream linked to that file will evaluate as false.

Unformatted and Binary I/O

- While reading and writing formatted text files is very easy,
  - it is not always the most efficient way to handle files.
  - Also, there will be times when you need to store unformatted (raw) binary data, not text.
- When performing binary operations on a file, be sure to open it using the **ios::binary** mode specifier.

- Although the unformatted file functions will work on files opened for text mode,
  - some character translations may occur.
  - Character translations negate the purpose of binary file operations.

Characters vs. Bytes

- Before beginning our examination of unformatted I/O, it is important to clarify an important concept.
- For many years, I/O in C and C++ was thought of as byte oriented.
- This is because a char is equivalent to a byte and the only types of streams available were char streams.
- However, with the advent of wide characters (of type wchar_t) and their attendant streams, we can no longer say that C++ I/O is byte oriented.
- Instead, we must say that it is character oriented.
- Of course, char streams are still byte oriented and we can continue to think in terms of bytes, especially when operating on nontextual data.
- Equivalency between a byte and a character can no longer be taken for granted.

put( ) and get( )

- One way that you may read and write unformatted data is by using the member functions **get( )** and **put( )**.
- These functions operate on characters.
- That is, **get( )** will read a character and
- **put( )** will write a character.
- Of course, if you have opened the file for binary operations and are operating on a **char** (rather than a **wchar_t** stream),
  - then these functions read and write bytes of data.

Commonly used version

istream &get(char &*ch*);
ostream &put(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.
- 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(int argc, char *argv[])
{
                    char ch;
                    if(argc!=2) {
                              cout << "Usage: PR <filename>\n";
                               return 1;
                      }
                    ifstream in(argv[1], ios::in | ios::binary);
                   if(!in) {
                                cout << "Cannot open file.";
```

Contd…

```
        return 1;
    }

        while(in) {
                        in.get(ch);
                        if(in) cout << ch;
    }
    return 0;
}
```

End of File

- When the end-of-file is reached,
  - the stream associated with the file becomes false.
- Therefore, when **in** reaches the end of the file, it will be false, causing the **while** loop to stop.

Compact Way

- There is actually a more compact way to code the loop that reads and displays a file,

```
while(in.get(ch))
cout << ch;
```

- This works because **get( )** returns a reference to the stream **in**, and

- **in** will be false when the end of the file is encountered.

read( ) and write( )

- Another way to read and write blocks of binary data is to use C++'s **read( )** and **write( )** functions.

- Prototypes:

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

read( ) and write( )

- The **read( )** function reads *num* characters from the invoking stream and puts them in the buffer pointed to by *buf*.
- The **write( )** function writes *num* characters to the invoking stream from the buffer pointed to by *buf*.
- **streamsize** is a type defined by the C++ library as some form of integer.
- It is capable of holding the largest number of characters that can be transferred in any one I/O operation.

More get( ) Functions

istream &get(char *_buf_, streamsize _num_);

- The first form reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, a newline is found, or the end of the file has been encountered.

- The array pointed to by *buf* will be null terminated by **get( )**.

- If the newline character is encountered in the input stream, it is *not* extracted.
- Instead, it remains in the stream until the next input operation.

get()

istream &get(char *buf, streamsize num, char delim);

- reads characters into the array pointed to by buf until either num-1 characters have been read, the character specified by delim has been found, or the end of the file has been encountered.
- The array pointed to by buf will be null terminated by get( ).
- If the delimiter character is encountered in the input stream, it is not extracted. Instead, it remains in the stream until the next input operation.

int get( );

- The third overloaded form of get( ) returns the next character from the stream. It returns EOF if the end of the file is encountered. This form of get( ) is similar to C's getc( ) function.

getline( )

- Prototype:
  istream &getline(char *buf, streamsize num);

- This reads characters into the array pointed to by buf until either num-1 characters have been read, a newline character has been found, or the end of the file has been encountered.

- The array pointed to by buf will be null terminated by getline( ). If the newline character is encountered in the input stream, it is extracted, but is not put
  into buf.

getline()

- **istream &getline(char *buf, streamsize num, char delim);**

- reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, the character specified by *delim* has been found, or the end of the file has been encountered.
- The array pointed to by *buf* will be null terminated by **getline( )**.
- If the delimiter character is encountered in the input stream, it is extracted, but is not put into *buf*.

Detecting EOF

- You can detect when the end of the file is reached by using the member function **eof( )**,
- Which has this prototype:
  bool eof( );
- It returns true when the end of the file has been reached; otherwise it returns false.