C++ Files

(revision 2)

Prof. Allen Holliday

February 3, 2005

Introduction

In a C++ program, there are several ways to do file IO—Unix, C standard IO, and C++ streams. It's somewhat confusing to see a mixture of similar approaches, so we're going to focus on C++. This paper contains a description of the basics of C++ file IO.

Revision 1 clarifies some language in various places and significantly changed the End-Of-File section—replacing an obsolete means of detecting the end-of-file condition.

Revision 2 removes the use of character arrays for strings, addresses file open flag combinations, explains the *tellg/tellp* functions, explains why file position isn't the same as character index, and points out a possible platform-dependent required argument for the *ios::pos_type* constructor.

References

The primary reference for this paper is "The C++ Standard Library: A Tutorial and Reference" by Nicolai M Josuttis (Addison-Wesley, 1999, ISBN 0-201-37926-0).

C++ streams are mentioned in the texts for CPSC-121 and CPSC-131, but the treatment is brief and incomplete.

File Streams

Part of the C++ Standard Library is the IOStream Library, and one of the classes defined by the latter is the file stream or *fstream*. To use this class, your program must include its header file:

```
#include <fstream>
```

You may then declare objects for the files that your program will read and write:

```
fstream  file;
```

You may also declare objects for input-only or output-only files

```
istream  inputFile;
```

or:

```
ostream  outputFile;
```

Almost all file operations are performed by calling functions of the *fstream* class; there's an exception for the *string* type. The sections below describe those functions. In these descriptions, the word *file* is used to mean file stream.

Opening Files

Files are created or opened by the *open* function, which has two forms:

```
file.open(name);
file.open(name, flags);
```

The *name* argument is a traditional C string (char*); if *name* is declared as a C++ string, you should use the file stream's *c_str()* conversion function:

```
string   name;
.
.
.
file.open(name.c_str());
```

The *flags* argument may be any meaningful combination of the following six flags:

Flag	Meaning
ios::in	Open for reading (default for istream)
ios::out	Open for writing (default for ostream)
ios:app	Always appends at the end when writing
ios::ate	Positions at the end of the file after opening (ate is short for "at end")
ios::trunc	Truncates file (removes its former contents)
ios::binary	Does no replacement of special characters during reading or writing, as is done for text files.

These flags can be used in certain combinations, as defined in the following table. The *ate* or *binary* flags aren't shown, since they don't interact with any other flag; *ate* is the same as a seek to the end of the file and *binary* disables checking for special characters or character pairs.

Flags	Meaning
in	Reads (file must exist)
out	Empties and writes (creates file if it doesn't exist)
out | trunc	Empties and writes (creates file if it doesn't exist)
out | app	Appends (creates file if it doesn't exist)
in | out	Reads and writes; initial position is the beginning (file must exist)
in | out | trunc	Empties, reads, and writes (creates file if it doesn't exist)

Combinations not shown in this table, except for those that merely add the *ate* or *binary* flags, are not legal.

Text and Binary Files

The data in text files is limited to lines of ASCII characters; binary files contain bit patterns that have no such constraint. The most significant issue is that when test files are read, the file system interprets and may replace certain characters. In some file systems, the end of a line is represented by two characters, carriage return and line feed. When a file is accessed in text mode, the newline

character in a string is replaced with this pair during a write and the pair is replaced with a newline during a read. No such conversion is done for files accessed in binary mode.

Note that the distinction between text and binary is a matter of how the file is opened and not the nature of the file's contents. It's entirely possible to take a file that was created as a text file, and open and read it as a binary file. For example, general purpose file dump programs usually do this.

Reading Files

There are several functions that read data from a file. To read a single byte, you may say:

```
char  c;
.
file.get(c);
```

I highly recommend using the C++ *string*, because it avoids some problems with traditional C character arrays.

You should use one of these forms of *getline*:

```
string   s;
.
.
getline(file, s);
getline(file, s, ':');
```

Notice that these are not functions of the *file* class. They are functions provided with—but not members of—the *string* class; they take an *fstream* or *istream* as an argument. There is no size argument—the string will be expanded as necessary to hold the entire line, which ends at a newline or a specified delimiter, just as with the previously described *getline* functions. There is a maximum size for a string but it's unlikely you'll encounter it (often it's 65,535).

You may also use the overloaded >> operator to read from a file:

```
file >> x >> y >> z;
```

The number of bytes read is determined by the types of the items being read into. When a file line contains several words separated by whitespace—spaces and tabs—only one word will be read by each >> operator.

Traditional C Character Arrays

I don't recommend using character arrays to hold text strings. They're more work and not as safe, because you must be careful about allocating enough memory (including the terminating null) for inputs of unknown size, and not letting strings run past the end of arrays, destroying the contents of other variables. However, if you do use them, there are file stream functions that you can use.

To read a line of ASCII text, up to some maximum number of characters, you may say:

```
char  s[10];
file.getline(s, sizeof(s));   //
```

The terminating newline is read but not stored in the character array. The next read operation will begin at the character after that newline.

To read a line of ASCII text, up to some maximum number of characters or a specified delimiter, you may say:

```
char  s[10];
file.getline(char*, sizeof(s), ':');
```

The terminating delimiter is read but not stored in the character array. The next read operation will begin at the character after that delimiter.

For either of these forms of the *getline* function, the *size* argument must include the null at the end of a traditional C string. There are two possible ways a *getline* may finish:

1. The specified number of characters, minus 1, has been read.

2. A newline or other delimiter has been seen.

In either case a null will be added after the last character read. An example may help clarify the first case. If the file contains the text "abcdef" and size = 3, the character array will contain the string "ab\0". Note that although three characters were specified, only two were read.

This is the same behavior as traditional C. The *size* argument specifies the size of the destination array, which should be the desired number of characters from the input plus one more for the terminating null that *getline* adds.

Binary (non-text) Data

To read a series of bytes that don't represent a line of ASCII text, you should say:

```
int   x[10];
.
.
file.read(x, sizeof(x));
```

or:

```
struct   s;
.
.
file.read(&s, sizeof(s));
```

This function doesn't terminate at a newline or any other delimiter and doesn't append a null after the data it reads. It is normally used for binary files.

End-Of-File Conditions

It's quite common for programs to loop through a file, repeatedly calling the *get*, *getline*, or *read* functions until there is no more data to read. Early non-ANSI/ISO C++ compilers provided the boolean function *file.eof()*, which indicated whether the last input operation failed because it attempted to read past the end of the file. The implementations where sometimes inconsistent and confusing. The *eof()* condition might be true after the last bytes of the file had been read, because the read operation was looking for some terminating character and encountered the end of the file instead.

Modern ANSI/ISO C++ IOstream libraries don't set the flag that the *eof()* function returns, making the function useless. The *fail()* function is the proper means for detecting the end-of-file condition. It returns *true* if an input operation doesn't succeed for any reason, which is usually either an illegal character, such as a letter when a digit is expected, or an end-of-file. It doesn't

have the *eof()* function's confusing behavior of returning *true* when the end-of-file is reached after successfully reading the remaining characters of a string.

Here's an example of the programming pattern for data types other than *string*:

```
while (1)
{
   file.getline(s, sizeof(s));
   if (file.fail()
   {
      break;
   }
}
```

The *string* example is only slightly different:

```
while (1)
{
   getline(file, s);
   if (file.fail()
   {
      break;
   }
}
```

In both examples, the check of *fail()* comes after the input operation; the only difference is the name of the input function itself.

Empty Lines, Whitespace, and End-Of-File

When *getline* is used to read strings from files that contain an empty line—those with nothing except a newline at their end—it passes back a string whose size is zero, which seems quite logical. When the >> operator is used, the behavior seems slightly different—it skips completely over those lines. If the remainder of the file consists only of empty lines, *fail()* will be *true*.

To understand this behavior, remember that *getline* is used to read entire lines, including any leading or trailing whitespace in the line. The >> operator is used to read words, not lines; it skips whitespace before words and stops at whitespace after words. The newline character at the end of any line is just more whitespace to skip over while looking for the next word. If nothing but empty lines remain in the file, the >> operator will fail to find a word.

Writing Files

There are fewer options for writing to files. To write a single character, you may say:

```
char  c;
file.put(c)
```

To write a series of bytes that don't represent a line of ASCII text, you should say:

```
char  x[10];
.
.
file.write(x, sizeof(x));
```

There's no write function for strings that corresponds to the *getline* functions for reading. To write a string, you should use the overloaded << operator:

```
string   s;
file << s;
```

Positioning (Seeking) In Files

Normally, a program opens a file and reads or writes it in sequential order. Sometimes, you need to access a file in random fashion, reading or writing data at different locations in the file, perhaps going back and forth from place to place. Every open file has two internal pointers, one for reading and one for writing, that indicate where the next access will start. They are expressed as offsets from the beginning of the file. Their initial values are zero, and each one is incremented by each read or write's actual size.

To access a file in random fashion, you can change the value of a pointer with the file's seek functions. There are two functions, each with two forms, for changing these pointers:

```
file.seekg(absolutePosition);
file.seekg(offset, relativePosition);

file.seekp(absolutePosition);
file.seekp(offset, relativePosition);
```

The last letter of the function name, g or p, is short for Get (read) and Put (write). The absolutePosition argument must be declared as:

```
ios::pos_type  absolutePosition;
```

Some compilers may require a constructor argument, such as:

```
ios::pos_type  absolutePosition(0);
```

The *offset* argument can be any integer type, and may be positive or negative. The *relativePosition* argument specifies the starting point for the offset.

Constant	**Meaning**
ios::beg	Position is relative to the beginning of the file
ios::cur	Position is relative to the current position in the file
ios::end	Position is relative to the end of the file

The position argument is not the same as the character index. Some file system implementations, such as MS-DOS and its descendants, replace certain characters with character pairs. For example, the single *newline* character is stored as the *carriagereturn/linefeed* pair. The string "ABC\nD" would be look like this in memory

0	1	2	3	4
A	B	C	\n	D

But it would be stored in a file as:

0	1	2	3	4	5

A	B	C	\r	\l	D

The D character's index is still 4, but its file position is 5.

There are two functions for finding the values of the current positions:

```
absolutePosition = file.tellg();
absolutePosition = file.tellp();
```

They can be called before a read or write to determine the file position of the item being read or written.

Closing Files

There's only one simple function to close a file when you're finished with it:

```
file.close();
```

Errors and Stream States

There are several boolean functions that indicate whether an error has occurred and what the state of the file stream is. These functions return the setting of several of the file's internal state flags.

Function	**Meaning**
file.good();	File stream is good; no unusual condition or error has occurred
file.eof();	An end-of-file condition has been encountered. Warning: modern ANSI/ISO C++ IOstream libraries don't set the flag that this function returns.
file.fail();	An error has occurred; includes fatal errors covered by bad();
file.bad();	A fatal error has occurred.

In general, a fatal error is one that can't be corrected by retrying the operation. For example, a disk error during a read or write. A non-fatal error is one that may be correctable by retrying the operation, usually with some argument changed. For example, if you want to create a file only if it doesn't already exist, you might first try to open it in *ios::in* mode to see if it does. A *true* from *file.fail()* is the desired result; it tells you there's no file that you'd be overwriting. You would then open the file in *ios::out* mode, after a *file.clear()* to clear the error flag.

A word of caution: when the internal flags are set, they remain set until they are explicitly cleared. Closing the file doesn't clear the flags, only this function does:

```
file.clear();
```

There's a common error that's made when using one fstream to handle multiple files, one at a time. A typical sequence of operations might be:

1. Open one file
2. Read the file until its end
3. Close the file
4. Open the next file

5. Read the file until its end
6. Close the file
7. Repeat for the remaining files

The error is forgetting to clear the flags before step 4. Reaching the end of the first file sets the *fail* flag and *fail* is still set when the open of the next file is attempted, so the open also fails.