More about... **External File Stream Input and Output**

CS 162: Introduction to Computer Science II

Overview

Earlier, we learned how to use the `iostream` library to perform basic input from and output to standard input, standard output, and standard error. Now it is time to extend what we learned and see how it applies when we read from or write to external files.

The following sections focus on the basic file stream I/O facilities and touch upon some of the most popular features when dealing with external text and binary files. We begin by understanding how to get started reading and writing external files. We progress then to look at how different data types are handled and when we need to use care. It is important to remember that both Appendix A and B are designed for those just getting started using C++ and I/O.

Getting Started with File Steam Input and Output

Input and output is performed as a stream of bytes handled through external libraries. This applies to external files in the same way that it applies to standard input and output, except that we use the `fstream` library instead. This library supplies the operations necessary to convert objects to sequences of characters for output to external files and to convert sequences of characters to objects of various types for input from external files. We can use the `fstream` library in exactly the same manner as we use the `iostream` library. The extraction operator, `get`, `getline`, `read`, and `ignore` operations are available for reading from external files. The insertion operator and `put` function are available for writing to external files (along with all of the formatting controls). Therefore, how we use external files follows the same rules as when we use standard input and output. The only difference is in the process of attaching our objects to specific external files. External files simply represent a different stream of input and output.

The `fstream` library includes the operations of the `iostream` library and predefines a set of operations for handling reading and writing of built-in data

types when working with external files. The `fstream` library defines three new data types: `ifstream`, `ofstream`, and `fstream`. The following sections help us get started using the `fstream` library.

Preprocessor Directive

C++ provides an enhanced `fstream` library for reading and writing external files. To use it, we must include the following preprocessor directive:

```
#include <fstream>
```

This is necessary because file I/O is not built into the language. The good news is that when we include the `fstream` library, we no longer need to also include the `iostream` library. This is because the `fstream` library "is an" `iostream` library in addition to being able to work with files. By including this library, we gain access to all of the necessary operations to perform input and output for standard in, standard output, standard error as well as for external files. This includes the ability to work with wide character streams. Then, at link time, the necessary functions from the `fstream` and `iostream` libraries are included.

The `fstream` library places its identifiers (e.g., objects and functions) in the `std` namespace, just like the `iostream` library. This means that we must either perform input and output with files by qualifying the relevant operations with the namespace and the scope resolution operator (i.e., `std::`), or we declare that all identifiers from that library be used without such a prefix through the `using` keyword:

```
using namespace std;
```

Objects for File Input and Output

Remember with the `iostream` library, objects are defined for use when working with standard input, standard output, and standard error (i.e., `cin`, `cout`, `cerr`, and `clog`). Unlike the `iostream` library, there are no objects predefined in the `fstream` library for working with external files. This is because it wouldn't make sense to have objects already attached to external files, since those files are client and/or user specific. The `fstream` library has no idea what files are to be used, or the directory from which they belong. Therefore, it is up to the client

program to define objects for file input and output and then to attach these objects to specific external files.

To define objects for file input and output, we can use one of three possible data types provided by the `fstream` library: `ifstream`, `ofstream`, or `fstream`. When reading from files, we define objects of type `ifstream`. When writing to files, we define objects of type `ofstream`. And, when both reading from and writing to files, we define objects of type `fstream`. In the following example, we define three objects, one of each type.

```
ifstream file_in;   //for reading from files
ofstream file_out;  //for writing to files
fstream  file_io;   //for reading from and writing to
files
```

The same applies to wide character streams, as shown in the following example:

```
wifstream wfile_in;   //wide character input from files
wofstream wfile_out;  //wide character output to files
wfstream  wfile_io;   //wide character file input/output
```

The object names chosen for reading and writing external files are not reserved. In fact, they are specified by the client programmer and can be any unique identifier within the scope in which they are defined. However, it is useful to select names that represent the type of I/O being performed. When reading from a file, it is useful if "in" or "read" is part of the name. When writing to a file, it is useful if "out" or "write" is part of the name. This facilitates self documenting code and reminds us of the type of operations allowable. Remember, with input, we use extraction and with output we use insertion!

Attaching File Stream Objects to External Files

To read or write external files, we must attach our file stream objects to specific external files. By doing so, we can use our file stream objects in place of `cin` with the extraction operator (>>) for input and in place of `cout`, `clog`, and `cerr` with the insertion operator (<<) for output.

There are two approaches for attaching file stream objects to external files. One is to specify the file name as the initial value of our objects; this implicitly

opens the file. The other is to explicitly open the file, by calling function `open` through a file stream object, passing in the file name. But what is a file name? It is an array of characters containing the exact name of the file and any necessary path information. It is system dependent whether the current working directory is the default path. The following example demonstrates how we can initialize a file stream object:

```
//attaches file_in to file emp.dat at initialization time
ifstream file_in("emp.dat");  //initializing file_in
...
```

There are times when we want to reuse our file stream objects and attach them to a different files. This is possible by opening the file at some later time. The following example demonstrates how we can attach a file to a file stream object later, after the object has been defined.

```
//attaches file_in to emp.dat at some later time
ifstream file_in;             //define a file stream
object
...
file_in.open("emp.dat");      //opening file emp.dat
...
```

Since the file name is just an array of characters, our programs can be more versatile by reading the desired file names from either the user or from some other external file. Then, we can supply these names as either the initial value of our file stream objects or through the `open` function. The following example demonstrates this approach:

```
//prompt the user for a file name
char fname[21];
cout <<"Please enter the file name: ";
cin >>fname;
ifstream file_in(fname);  //initializing file_in
...
```

With the `open` function, we can accomplish the same thing but separate the definition of our file stream objects from the file to which they are attached:

```
//prompt the user for a file name
```

```
char fname[21];
cout <<"Please enter the file name: ";
cin >>fname;
ifstream file_in;       //define the file stream object
file_in.open(fname);    //open the user specified file
...
```

Before we use any file stream object for input or output, we should make sure that we have successfully been able to attach it to the requested file. We do that by checking the value of that file stream object. If it is zero after opening the file, it has not been successfully attached. If it is non-zero, then we can proceed.

Once we have successfully attached our file stream objects to a specific file, we can begin to read from or write to that file. Since `file_in` is an `ifstream` object, by default the file is opened for input and the entire file is treated as the input stream for `file_in`. Therefore, we can use the extraction operator in conjunction with `file_in` to read data from that file. Doing so provides an elegant approach to performing file I/O which doesn't require that we specify the data types being read. We can also use `file_in` in conjunction with many of the functions we have examined, such as `get`, `getline`, `gcount`, `read`, `ignore`, `eof`, and `peek`. These provide alternate ways for reading in characters and character strings, determining how many characters have been read, skipping characters, and determining whether or not an end of file has occurred.

Closing External Files

Once we are done reading from or writing to an external file, we should close the file. This causes our objects to no longer be attached to the current file. There are two ways to close a file. Files are automatically closed when we reach the end of the lifetime of our file stream objects. Files can also be explicitly closed by calling the function `close` through a file stream object. The following demonstrates both approaches:

```
{
  ifstream file_in("emp.dat");  //define file object
  ..
  file_in.close();  //explicitly close emp.dat
  ...
  file_in.open("mgr.dat");  //attach a different file
  ...
} //implicitly close mgr.dat
```

Notice that once a file is closed, we can reattach a file to that file stream object. It may be either the same file or a different one. Also notice that we don't supply the file name in the call to `close`. This is because we can only close the file to which a file stream object is currently attached. Providing a file name would be meaningless, as only one file is ever attached at any given time.

Input File Stream

Reading from external files requires that we have either an `ifstream` or `fstream` object defined and attached to the desired file. A file stream object can only be attached to one file at any given time. Once opened, we can begin reading at the beginning of the file. Think of the file as simple a large "input buffer", where we have a file pointer that is initialized to the first character in the file. As we read, the file pointer progresses through the file until we reach the end of the file. The input operations behave as if we were performing standard input, but from a different input stream. We can use the file stream object to read from this stream instead of from standard input, up until we reach the end of the file. At that point, our input operations will begin to fail.

The following sections demonstrate the use of file stream objects for input and show a variety of ways of stepping through reading entire files.

Opening External Files for Input

When we open a file for input we can specify that we want the file to be opened for reading (which is the default for `ifstream` objects) and that the file is either text or binary. We can do this by explicitly specifying an additional argument in our open sequence. This can be done when we define `ifstream` objects or explicitly with the use of the `open` function. Table B-1 lists the modes for file input.

`ios::binary`	I/O is in binary rather that text (default is text)
`ios::in`	Open for reading (default for `ifstream` objects)
`ios::out`	Open for writing (useful for `fstream` objects)

Table B-1: Input File Stream Modes

The following represents the default condition for files that we open. This assumes that we are opening a file for input and that it consists of a text file. It is the same as if we had just used `open` with just the file name specified.

```
ifstream text_in;
text_in.open(fname,ios::in);     //this is the default
```

We can also open a file and specify that it is a binary file rather than a text file:

```
ifstream bin_in(fname, ios::binary); //a binary file
...
```

We can specify more than one mode simultaneously by 'ORing' the modes together. This can also be done implicitly as we define a file stream object or explicitly as we open a file.

```
//a binary file for input
ifstream bin_in(fname, ios::in|ios::binary);
```

This is especially useful when opening fstream objects for both input as well as output:

```
//a file being opened for both input and output
ifstream bin_in(fname, ios::in|ios::out);
```

Reading External Files

When a file has successfully been opened for input we can begin reading from the beginning of the file. This means that any use of the extraction operator, or the `get`, `getline`, `read`, `ignore`, and `peek` functions begin reading from the first character. Just as with standard input, we can use the extraction operator to skip leading whitespace and read in single characters, sequences of non-whitespace characters, integers, and reals. We can use the `get` function to read single characters when supplied with zero or one argument or sequences of characters when supplied with two or three arguments. The following demonstrates these for `file_in`:

```
char ch, str[81];
int i;
```

```
float f;

file_in >>i;   //read an integer from the file
file_in >>f;   //read a float from the file
file_in >>ch;  //read a single character from the file
file_in >>str; //read an array from the file
ch = file_in.get(); //read a single character

file_in.get(str, 81, '\n'); //read a line of text
file_in.get(ch);            //read a single character
file_in.getline(str,81,'\n');
```

Reading from a file stream follows all of the same rules that we learned with the iostream library. Client programs must be aware of the data types stored within the file to read that information back in a meaningful way. And, data written to a file must be able to be read back in, which means that whitespace must surround that data in a way that would facilitate subsequent reading.

When reading lines of text, we can either read the data a single character at a time, or we can read entire lines. If whitespace is important to our application, then we should read using the `get` or `getline` functions. If whitespace is simply used to separate fields, then we can use the extraction operator instead. If the `get` or `getline` functions are used, we must be aware of the delimiting character. The two or three argument versions of `get` and `getline` require that we extract any extra delimiting characters prior to using them. And, `get` leaves us positioned at the delimiting character after performing the read. `getline`, on the other hand leaves us positioned at the next character after the delimiting character. Therefore, remember to ignore, when necessary, delimiting characters that will prohibit subsequent input operations from being successful.

Reading Until End of File

End of file also works the same as we learned with the iostream library. We can use the returned value of `get` or the extraction operator in a conditional expression. Alternatively, we can call the `eof` function through an object of the `ifstream` class (e.g., `file_in.eof()`) to determine if the previous input operation failed. `eof` returns `true` if the previous input operation failed (such as through an end of file) and `false` otherwise. It is especially important to realize that if the previous input operation did not fail, `eof` will not return `true` even when we believe that we have read the last item. This is because `eof` simply

returns whether or not a mode-bit has been set from a previous read; it does not examine the contents of the input stream. Therefore, we should always read prior to using this function. Otherwise, this function is meaningless! This is especially important when working with files.

The following example demonstrates the use of end of file to read and echo the entire contents of a file:

```
//main.cpp  (ExB01)
#include <fstream>
using namespace std;

int main() {
  char s[81];
  char fname[]="emp.dat";
  ifstream file_in(fname); //open the file
  ...

  //read a string and then check if an eof occurred
  if (file_in) {                  //did open succeed?
    while (file_in.get(s, 81) && !file_in.eof()) {
      cout <<s <<endl;            //echo the string
      file_in.get();              //flush delimiter
    }
    file_in.close();              //explicit close
  }
  return (0);
}
```

When an end of file is encountered, subsequent input operations will fail. In some situations, such as when we use `get` with no arguments, `-1` is returned when an end of file has occurred. Remember that if too many input operations are attempted prior to checking for end of file, the end of file bit may be reset and therefore undetectable by the `eof` function. Therefore, it is safest to check for end of file after each input operation when an end of file is a possibility. And, if we don't first read before checking for end of file, we may find that we get the last item in the file displayed twice!

Examples of Reading External Files

We can read external files in many ways. The following example demonstrates how we can read and echo an entire file one character at a time as well as a line at a time.

```
//main.cpp  (ExB02)
#include <fstream>
using namespace std;

int main() {
  char ch, s[81];
  char fname[]="emp.dat";
  ifstream file_in(fname); //open the file,  ...

  //read character by character until an eof occurred
  if (file_in) {                  //did open succeed?
    while (file_in.get(ch)) {     //fails on end of file
      cout <<s <<endl;            //echo the character
    }
    file_in.close();              //explicit close
  }

  file_in.open(fname);            //reopen the file
  ...

  //read a string and then check if an eof occurred
  if (file_in) {                  //did open succeed?
   while (file_in && file_in.getline(s, 81)
             && !file_in.eof()) {
      cout <<s <<endl;            //echo the string
    }
    file_in.close();              //explicit close
  }
  return (0);
}
```

Output File Stream

We use the output file stream to create or append data to external files. As with input, this requires that we start by defining file stream objects and then attaching those objects to the desired external files. Once successfully opened, we can use the insertion operator to write single characters, integers, reals, and strings out to the file.

Writing to external files requires that we have either an `ofstream` or `fstream` object defined and attached to the desired file. As with input, a file stream object can only be attached to one file at any given time. By default, when a file is opened for output the entire contents of the file is destroyed. Think of the file as simple a large "output buffer", that begins as empty by default. The good news is that we can specify when opening our files that we want to append data without destroying its original contents. These will be discussed in the following sections.

Opening External Files for Output

When we open a file with an `ofstream` object, by default we are opening a file and treating it as an empty file. If the file is not empty, the file will still be treated as if it is empty, destroying its previous contents. If this is not the desired behavior, we can open a file with an additional argument specifying an output mode.

When we open a file for output we can specify that we want the file to be open for writing (which is the default for `ofstream` objects) and that the file is either text or binary (text is the default). We can do this by explicitly specifying an additional argument in our open sequence. This can be done when we define an `ofstream` object or explicitly with the use of the `open` function. Table B-2 lists the modes that correspond to output files.

`ios::app`	Append to an existing file
`ios::ate`	Open and seek to the end of file
`ios::binary`	I/O is in binary rather that text (default is text)
`ios::in`	Open for reading (especially for `fstream` objects)
`ios::out`	Open for writing (default for `ofstream` objects)
`ios::trunc`	Truncate the file to have zero items

Table B-2: Output File Stream Modes

The following represents the default condition for files opened for output. This assumes that we are opening a file for output, that it consists of a text file, and that it contains no data. It is the same as if we had just used open with just the file name specified.

```
ofstream text_out;    //this is the default
text_out.open(fname,ios::out|ios::trunc);
```

We can also open a file and specify that it is a binary file rather than a text file:

```
ofstream bin_out(fname, ios::binary); //a binary file
...
```

We can specify more than one mode simultaneously by 'ORing' the modes together. This can also be done either when we define a file stream object or when we explicitly open a file.

```
//appending at the end of a binary file
ofstream bin_out(fname, ios::app|ios::binary);
```

Writing External Files

Data is written to external files in exactly the same way that it is written to standard output. This means that nothing is automatically written; everything we need written to the file must be inserted into that output stream. For newlines, we must explicitly insert the `endl` into the output stream or write the character `'\n'`.

It is important to note that text files only contain characters. That means if we write an integer or a real to a text file, it will be converted to a character (or sequence of characters) when it is written. It will still look like a number, but the number `10` will be stored as the character `1` and the character `0`.

The following example demonstrates how we can write various data types to the end of an existing file:

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

int main() {
  //open a file for output and to append
  ofstream file_out("test.dat", ios::app);

  if (file_out) {  //if the open was successful
    file_out <<"This line is written to Test.dat!";
```

```
    file_out <<"...with this attached!" <<endl;
    file_out <<1001;
    file_out <<endl;
    file_out.close();
  }
  return (0);
}
```

The first time the file is opened, it will contain the following information:

```
This line is written to Test.dat!...with this attached!\n
1001\n
```

If we ran this program a second time, we would be appending the same set of text to the end of the file:

```
This line is written to Test.dat!...with this attached!\n
1001\n
This line is written to Test.dat!...with this attached!\n
1001\n
```

Notice how important it is to write newlines after the last item written to the file. If we had not, subsequent reads would not be able to detect the difference between 1001 and the second `This`. Therefore, it is wise to consider writing a newline at least after the last data item written to a file.

Designing Functions to Read and Write External Files

When designing functions to read and write our external files, we have three choices for using our file stream objects. We could use globals, locals, or pass file stream objects to functions. Doing the latter allows us to open a file in one function and manipulate its contents from another function, without unnecessary side effects. The following sections discuss each of these approaches and demonstrate the use of file stream arguments through a complete example.

Using File Stream Objects in Functions

Functions that use global file stream objects allow one function to open a file and another to read and/or write that file. But, this inherently ties one function to

another and may cause unwanted side effects. It also reduces the maintainability of our software and may unnecessarily clutter our global namespace.

Another approach is to define local file stream objects within our functions. As locals, the file stream objects must be reattached before reading and/or writing to files and then the files are implicitly closed at the end of that file stream object's lifetime regardless of whether or not we explicitly close the file. Therefore, local file stream objects have limited application. They do not allow us to open a file in one function and then access that file in any other function.

The best solution is to pass file stream objects to our functions. When we do so, it is vital that we pass them by reference to avoid making local copies of our file stream objects within each function. Pass by reference gives functions an alias to the calling routines actual file stream object. As such it does not cause files to implicit close at the end of our functions. If we used pass by value by mistake, our functions would be given a complete copy of the actual file stream object; the argument is then treated as a local that has a lifetime that is limited to the function being called.

The following prototype statements demonstrate how functions can accept file stream objects by reference.

```
bool open(char fn[], ifstream &in);  //open fn for input
bool open(char fn[], ofstream &out); //open fn for output
```

The following is the implementation of these functions:

```
bool open(char fn[], ifstream &in){  //open fn for input
  return (in.open(fn));  //return false if open fails
}
bool open(char fn[], ofstream &out){ //open fn for output
  return (out.open(fn)); //return false if open fails
}
```