Laboratory 8: *File It Away*

Overview: This week's lab focuses on the reading and writing of C++ character files. The ability to read data from and write it to secondary memory makes possible a tremendous amount of flexibility in C++ programs. Using files correctly involves the mastery of a number of small details which we'll practice today.

Objectives: Before you leave lab today you should understand, among other things:

a. how to declare, define and use file variables for reading and writing,
b. the difference between file variables and file names,
c. how streams can be passed as parameters to functions,
d. how different member functions of the **ifstream** class can be used,
e. how to check for the end of a file, and the end of a line in the file,
f. how an operation that reads a value can also be used to test for the end of a file.

1. Setting up for the lab

Please perform the following activities as you wait for the lab to formally begin:

a. Boot up the machine and log on to the CS Department server.

b. Put your lab diskette in the *A:* drive and make a subdirectory *LAB08* in the *LABS* directory.

c. Download the 7 files from the class Web site (http://s9000.furman.edu/chealy/cs11/lab08) to your new directory.

2. Reading numeric data with varying text file formats

The C++ fstream library provides the capability for the creation of character files which permit simple reading of data from files stored in secondary memory, and writing of a program's output to a similar file. Don't be thrown by the name "character" file. It turns out that all data is entered into a computer in character format. The system just converts numerical values automatically before they are stored in RAM.

It is typically very important to know something about the type of file that your program will be processing. However, programs often need to be able to process files which are formatted in different ways. A frequently-faced problem is that of reading data from a file when you don't know how many lines of data there are. To solve this problem, C++ provides the capability of testing for a special *end-of-file marker* that is automatically placed at the end of every file. The problem becomes even more difficult when you don't know how many values there are on each line of data either. We'll see how to solve both of these problems in our first activity.

a. Start up Borland C++ and open the file READINT1.CPP. Read through the file carefully to understand what it is doing.

Note that stream objects can be passed as parameters to functions just as other types of objects can be passed. They must be passed as reference parameters.

b. There are several data files included with the lab. The file *INTS1.DAT* has three lines each containing four integers. Run the program using this file to check the program's function. Jot down the output (in lines as they are displayed on the screen).

c. Now run the program using the file *INTS2.DAT* using 3 for the number of lines and 4 for the number of values per line. Again, record the results (in lines as they are displayed on the screen).

d. Run the program again using *INTS2.DAT* using 2 lines and 6 values per line. Write the program’s output here.

Fixing the program to make it general

You may not know it yet, but there is a big problem reflected in the results you have written down so far. (If you want an early hint as to what the problem is, take a look at the actual contents of the file *INTS2.DAT*.) A problem that you can see right now is that it is inconvenient to have to enter the exact number of lines and columns of data. What if you don't know? What if it changes from file to file? What if there is a different number of values on each line?

To deal with these problems, we will have to test for end of file each time a number is read in. The loop that performs this input will have to be modified so that it reads until the *end-of-file* marker (identified by the **EOF** constant in a C++ program) is reached. This is made easy by the **ifstream** member function **eof()**. This function returns *1* (*true*) when the end-of-file marker has been read, and *0* (*false*) otherwise.

e. Modify the *read_numbers* function so that it no longer has to collect any information from the user. The program should be capable of reading and writing any file of integers—no matter how many lines the file contains and no matter how many values there are per line.

f. Save the modified program and test it by running it with the original file *INTS1.DAT*. The output should look identical to the output you recorded earlier. If it does not, make corrections and try again.

g. Run your program again, this time using the file *INTS2.DAT*. Record the results (in lines as they are displayed on the screen). You should notice that the last output looks quite different from the output that was produced earlier from exactly the same file. Open the INTS2.DAT file. The output from your recent changes should contain the same number of lines and the same number of values per line as in the file. Do you understand why your output looks the way it does? √

3. Creating output files

Once you have mastered the details of file input, you'll find file output to be pretty straightforward. Next, we'll experiment with writing data to a file by creating a new function named *square_numbers* which will—as you probably could guess—read values from a file, square them, and write them back to a new file in exactly the same format. The original procedure *read_numbers* can then be used to verify your work by printing both files for you to check visually.

a. Close any open files in Borland C++. Open the file *READINT2.CPP* and study it. Make sure you understand what the program is supposed to do before continuing.

b. Modify the main program as needed. Specifically, add the code necessary to set up a file variable **fout** for output. The user of the program should be able to specify the file name that the variable will refer to.

c. Construct the function *square_numbers*. This procedure should read the numbers from **fin** one at a time, square them, and then write them to **fout** in exactly the same line format.

Use the function *read_numbers* as a model. Only relatively small modifications will be needed.

d. Save the modified program and test it by running it with the file *INTS1.DAT*. Recall that you recorded the contents of this file earlier in the lab. If the program doesn't work properly, make corrections and try again. Your output file should contain the same number of lines and the same number of values per line as *INTS1.DAT*. The values in each line should be the square of the values in *INTS1.DAT*. √

4. Reading and processing character data from files

The processing of character data is perhaps the most common computing activity there is. Just about everyone has used a word processor at some point. C++ allows for the reading, manipulation and writing of text files necessary for word processing. Our final activity for the afternoon will deal with files containing character data.

a. Close all open windows in Borland C++ and open the file *READCHAR.CPP*. Study the logic and ask me if you have any questions about it.

Note the use of the **get()** member function from the **ifstream** class. When **get()** is used, whitespace characters such as blanks, carriage returns and tabs are not ignored. Remember that when the extraction operation >> is used, all such characters are disregarded.

Note also that this problem could use a strategy similar to that used in the first part of today's lab. In general, C++ allows many ways to do the same thing.

b. Run the program using the file *TEXT.DAT* as input.

c. Modify the program so that it prints out the number of lines in the input file, and the number of times the letter 'e' appears. √

5. Finishing up

Close all windows and exit Borland C++.