File Input/Output Streams in C++

CS 16: Solving Problems with Computers I
Lecture #10

Ziad Matni
Dept. of Computer Science, UCSB
Announcements

• Homework #9 due today
• Homework #10 is out

• Midterm #2 is on Thursday, 5/18
  – That’s next week!

• Names on Homework Assignments!!!
Outline

Chapter 6 in textbook

• I/O Data Streams and File I/O
• An introduction to Objects
  – Member functions
• Handling File I/O Errors
Objects

• Objects are special variables that have their own special-purpose functions
  – There’s more to them than that, but more on this later…

• Example: String length can be gotten with

\[ \text{stringname.size()} \]

• These are called \textit{member functions}
I/O Streams

• I/O = program Input and Output
  – Delivered to your program via a *stream object*
  – A stream is a *flow of data*

• Input can be from the *keyboard (cin)* or a *computer file*
• Output can be from the *screen (cout)* or a *computer file*

• Input stream → Data flows *into* the program
• Output stream ← Data flows *out* of the program
Why Use Files?

4 Good Reasons:

• Files allow you to store data permanently and conveniently!

• Data output to a file lasts after the program ends
  – You can usually view them without the need of a C++ program

• An input file can be used over and over
  – No typing of data again and again for testing

• Files allow you to deal with larger data sets
File I/O

• **Read (input) from a file**
  – Usually done from beginning to the end of file (not always)
    • No backing up to read something again (but it’s OK to start over)
    • Similar to how it’s done from the keyboard

• **Write (output) to a file**
  – Usually done from beginning to end of file (not always)
    • No backing up to write something again (but it’s OK to start over)
    • Similar to how it’s done to the screen
Stream Variables for File I/O

You have to use “stream variables” for file I/O:

• Must be **declared** before it can be used

• Must be **initialized** before it can contain valid data
  – Initializing a stream means *connecting it to a file*
  – The value of the stream variable is really the filename it is connected to

• Can have their values changed
  – Changing a stream value means disconnecting from one file and then connecting to another
Streams and Assignment

- Streams use special built-in (member) functions instead of the assignment operator to change values.

- Example:

```java
streamObjectX.open("addressBook.txt");
streamObjectX.close();
```
• Input-file streams are of type `ifstream`

• Type `ifstream` is defined in the `fstream` library

• You must use `include` statement and `using` directives
  ```cpp
  #include <fstream>
  using namespace std;
  ```

• Declare an input-file stream variable with:
  ```cpp
  ifstream in_stream;
  ```

  Variable type       Variable name
Declaring An
Output-file Stream Variable

- Output-file streams of are type `ofstream`

- Type `ofstream` is defined in the `fstream` library
- Again, you must use the `#include` and `using` directives
  ```
  #include <fstream>
  using namespace std;
  ```

- Declare an input-file stream variable using
  ```
  ofstream out_stream;
  ```
Once a stream variable is declared, you connect it to a file

- Connecting a stream to a file means “opening” the file
- Use the *open* function of the stream object

```cpp
in_stream.open("infile.dat");
```

File name on the disk

*Must include a true path (relative or absolute)*
Using The Input Stream

• Once connected to a file, get input from the file using the extraction operator (>>)
  – Just like with cin

Example:

```cpp
ifstream in_stream;
in_stream.open(“infile.dat”);
int one_number, another_number;

in_stream >> one_number >> another_number;

in_stream.close();
```

The inputs are read from the infile.dat file separated by either spaces or newline characters.
Using The Output Stream

• An output-stream works similarly using the insertion operator (<<)
  – Just like with cout

Example:

```cpp
ofstream out_stream;
out_stream.open("outfile.dat");

out_stream << "one number = " << num1
  << " , another number = " << num2;

out_stream.close();
```

The output gets written in the outfile.dat file
The External File Name

• Must be the name of a file that the operating system uses

• Be compliant with naming conventions on your system
  – Example: Don’t call an input **text** file \texttt{XYZ.jpg}

• Make sure the path is true
  – If the file is local to your program, then no path is needed
  – Otherwise use either relative or absolute path names

Example: \texttt{inFile.open("../MyDirectory/inputFile_42.txt");}
Closing a File

• After using a file, it should be closed using the .close() function
  – This *disconnects* the stream from the file
  – Close files to reduce the chance of a file being corrupted if the program
    terminates abnormally

• *Example:* `in_stream.close();`

• It is important to close an output file if your program later needs
  to read input from the output file

• The system will automatically close files if you forget
  *as long as your program ends normally!*
**Member Functions**

**Member function**: function associated with an object

- `.open()` is a member function of `in_stream` in the previous examples
  - `in_stream` is an object of class `ifstream`

- Likewise, a **different** `.open()` is a member function of `out_stream` in the previous examples
  - Despite having the same name!
  - `out_stream` is an object of class `ofstream`

For a list of member functions for I/O stream classes, also see:

Classes vs. Objects

• A class is a complex data type that contains variables and functions
  – Example: `ifstream`, `ofstream`, `string` are C++ classes

• When you call up a class to use it in a program you *instantiate* it as an object
  – Example:
    ```cpp
    ifstream MyInputStream;
    // MyInputStream is an object of class ifstream
    ```
Calling a Member Function

- Calling a member function requires specifying the object containing the function.
- The calling object is separated from the member function by the dot operator.
- Example: `in_stream.open("infile.dat");`

```
Calling object  Member function

Dot operator
```

5/9/17
Errors On Opening Files

• Opening a file can fail for several reasons
  – The file might not exist
  – The name might be typed incorrectly
  – Other reasons

• **Caution**: You may not see an error message if the call to open fails!!
  – Program execution usually continues!
Catching Stream Errors

- Member function `fail()`, can be used to test the success of a stream operation.
  - `fail()` returns a Boolean type (True or False).
  - `fail()` returns True (1) if the stream operation failed.
Halting Execution

• When a stream open function fails, it is generally best to stop the program

• The function `exit()`, halts a program
  – `exit(n)` returns its argument (n) to the operating system
  – `exit(n)` causes program execution to stop
  – `exit(n)` is NOT a member function! It’s a function defined in `cstdlib`

• Exit requires the include and using directives
  
  ```
  #include <cstdlib>
  using namespace std;
  ```
Using **fail** and **exit**

- Immediately following the call to open, check that the operation was successful:

```cpp
in_stream.open("stuff.dat");
if( in_stream.fail( ) )
{
    cout << "Input file opening failed.\n";
    exit(1); // Program quits right here!
}
```
Techniques for File I/O

- When reading input from a file **do not include prompts or echo the input**
- Unlike when you’re reading from a keyboard
  - The lines `cout << "Enter the number: ";
cin >> the_number;
cout << "The number you entered is << the_number;`

  become just one line when doing a file input read:

  `in_file >> the_number;`

- The input file must contain *just* the data that’s expected
  - So you have to be familiar with how the input file is organized
Formatting Output to Files

- Format output to the screen with:
  
  ```
  cout.setf(ios::fixed);
  cout.setf(ios::showpoint);
  cout.precision(2);
  ```

- Similarly, format output to a file using `out_stream` with:
  
  ```
  out_stream.setf(ios::fixed);
  out_stream.setf(ios::showpoint);
  out_stream.precision(2);
  ```
### Formatting Flags for `setf`

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ios::fixed</code></td>
<td>If this flag is set, floating-point numbers are not written in e-notation. (Setting this flag automatically unsets the flag <code>ios::scientific</code>.)</td>
<td>Not set</td>
</tr>
<tr>
<td><code>ios::scientific</code></td>
<td>If this flag is set, floating-point numbers are written in e-notation. (Setting this flag automatically unsets the flag <code>ios::fixed</code>.) If neither <code>ios::fixed</code> nor <code>ios::scientific</code> is set, then the system decides how to output each number.</td>
<td>Not set</td>
</tr>
<tr>
<td><code>ios::showpoint</code></td>
<td>If this flag is set, a decimal point and trailing zeros are always shown for floating-point numbers. If it is not set, a number with all zeros after the decimal point might be output without the decimal point and following zeros.</td>
<td>Not set</td>
</tr>
<tr>
<td><code>ios::showpos</code></td>
<td>If this flag is set, a plus sign is output before positive integer values.</td>
<td>Not set</td>
</tr>
<tr>
<td><code>ios::right</code></td>
<td>If this flag is set and some field-width value is given with a call to the member function <code>width</code>, then the next item output will be at the right end of the space specified by <code>width</code>. In other words, any extra blanks are placed before the item output. (Setting this flag automatically unsets the flag <code>ios::left</code>.)</td>
<td>Set</td>
</tr>
<tr>
<td><code>ios::left</code></td>
<td>If this flag is set and some field-width value is given with a call to the member function <code>width</code>, then the next item output will be at the left end of the space specified by <code>width</code>. In other words, any extra blanks are placed after the item output. (Setting this flag automatically unsets the flag <code>ios::right</code>.)</td>
<td>Not set</td>
</tr>
</tbody>
</table>
Creating Space in Output

- The **width** member function specifies the number of spaces for the next item
  - Applies *only to the next item of output*

**Example:**
- To print the digit 7 in four spaces and use
  ```cpp
  out_stream.width(4);
  out_stream << 7 << endl;
  ```
  Three of the spaces will be blank:

```
  7
```

- `.setf(ios::right)`
- `.setf(ios::left)`
- *default*
Not Enough Width?

• What if the argument for width is too small?
  – Such as specifying `cout.width(3);`
    when the value to print is **3456.45**

• The entire item is always put in output
  – If too few spaces are specified, as many more
    spaces as needed are used
  – In the example above, the value is still printed as if the
    `cout.width(3);` was not there.
Unsetting Flags

• Any flag that is set, may be unset
• Use the `unsetf` function
  – Example:

```cpp
cout.unsetf(ios::showpos);
```

causes the program to stop printing plus signs on positive numbers
Manipulators

• A function called in a nontraditional way

• Manipulators, in turn, call member functions
  – May or may not have arguments

• Used after the insertion operator (<<) as if the manipulator function call is an output item
The `setw` Manipulator

- `setw` does the same task as member function `width`
  - `setw` calls the width function to set spaces for output
  - Found in the library `<iomanip>`

- Example: `cout << "Start" << setw(4) << 10`  
  `<< setw(4) << 20 << setw(6) << 30;`

  produces: Start  10   20   30

  2 Spaces   4 Spaces

  - The 1st `setw(4)` ensures 4 spaces between "Start" and 10, INCLUSIVE of the spaces taken up by 10.
  - The 2nd `setw(4)` ensures 4 spaces between 10 and 20, INCLUSIVE of the spaces taken up by 20.
  - The 3rd `setw(6)` ensures 6 spaces between 20 and 30, INCLUSIVE of the space taken up by 30.
The **setprecision** Manipulator

- **setprecision** does the same task as member function **precision**
  - Found in the library `<iomanip>`

- Example:
  ```cpp
  cout.setf(ios::fixed);
  cout.setf(ios::showpoint);
  cout << "$" << setprecision(2)
       << 10.3 << endl
       << "$" << 20.5 << endl;
  ```

  produces:
  
  $10.30
  
  $20.50

- **setprecision** setting stays in effect until changed
Appending Data to Output Files

• Output examples we’ve given so far create new files
  – If the output file already contained data, that data is now lost

• To append new output to the end an existing file use the constant ios::app defined in the iostream library:
  
  ```cpp
  outFile.open("important.txt", ios::app);
  ```

• If the file does not exist, a new file will be created

• Other member functions include those that return where in the output file (or input file) the next data will be
  – Helps with customizing read and writing files
  – To be used carefully!
Entering File Names for I/O Files

• Users can also enter the name of a file to be read/written
  – As an input read by cin

• You can use regular C++ strings for the filenames, but ONLY if you ensure that you are compiling with C++ version 11 (or later).

• OTHERWISE, you’ll have to use C-strings
  – WARNING!!!! PAY ATTENTION TO THIS!!!

• Textbook has details on how to use C-strings for filenames
Note on Compiles using C++ ver. 11

• To make sure you compile using an updated version of C++ (ver. 11), do this (in Linux):

\[
g++ \text{ prog.cpp} -o \text{ prog} \ -\text{std=c++11}
\]

• If you want your compiler to give you warnings (as well as error msgs of course), do this (in Linux):

\[
g++ \text{ prog.cpp} -o \text{ prog} \ -\text{std=c++11} \ -\text{Wall}
\]
These Compile Statements are Getting Looooong!

• Wait until you have to compile these long statements over and over again for a multitude of C++ files!

• **Makefiles** to the rescue!
  – Next week, we’ll discuss how to use this valuable Linux tool
In-Class DEMO!

*Program does the following:*

- Reads inputs from an input file
  - User will enter filename for the program

- Does some computations with the read inputs

- Writes the results as outputs to an output file
  - User will enter filename for the program
To Dos

• Homework #10

• THU: More on I/O formatting & Strings in C++
  – Read rest of Chapter 6 in textbook

• TUE: Arrays
  – Read Chapter 7 in textbook
</LECTURE>