Lecture 03 Notes

Derek Harter

CSci 515 Spring 2014 *<2015-01-28 Wed>*

1 First Session (11 - 11:40)

1.1 Streams

The IOStream library is a new (object oriented) library, added with the C++ language, to support Input and Output to source and destination devices.

The source of input can be a keyboard, a file, or some other device. Likewise the destination of output can be to a file, to a terminal screen, or to some other device (for example you can send output into another C variable, like a string in memory).

A stream is a way of visualizing how data is transferred from the source to destination. A stream is inherently serial, the order in which you put things into the stream, is the order they will be received when they come out of the stream.

1.2 iostream header

You've already seen many examples of specifying the iostream header using

```
#include <iostream>
```

Iostream operators and objects are defined in the std namespace, thus you explicitly have to specify `std::` before using them, or include the

```
using namespace std;
```

directive.

In addition to iostream, if you want to do I/O to files, you need to include the fstream header. If you want to manipulate and format the data in/out of the stream, you need to include the iomanip header.

1.3 Standard Stream Objects

- cin, cout input from the standard input device, and output to the standard error device respectively. These are the keyboard and terminal, by default, but can be connected to others (like a file) by the OS, and program doesn't know or care.
- cerr send output to the standard error device, can be useful for separating error messages from normal output (and redirecting standard error to a different location). By default, standard error also goes to the terminal.
- clog also connects to the standard error output in a buffered manner. You don't need to be concerned with clog in this class.

1.4 Stream Output and Input

- using the << >> stream notation

```
cout << x << y << z;
cin >> x << y << z;
```

- Using member functions. The streams cout, cin, are objects, they have member functions. For example, and put and get single characters:

```
cout.put('A').put('\n');
cin.get(c);
```

- Example of reading a character at a time of input and echoing until EOF

```
int c; // use int, because char cannot represent EOF
while ( (character = cin.get()) != EOF)
{
  cout.put(character);
}
```

- Example of reading input a line at a time

```
const int SIZE = 80;
char buffer[SIZE];

cin.getline(buffer, SIZE);
```

- peek, putback and ignore can be used for low level I/O. We can ignore a number of characters, and we can peek ahead (without reading) or putback a character into the stream.

1.5 Formatted I/O, Precision and float output

- We can display integers in any base, using hex, oct, dec and setbase()
- For float/double types, use cout.precision(p) or setprecision(p) manipulator to set number of decimal places shown.
- For float/double, you can force fixed or scientific notation output using those output stream manipulators.

1.6 Field width

- For strings (both inputs and outputs) can limit output using cxx.width(w) or the setw(w) io manipulator.

1.7 Other Formatting

- Figure 15.12 in section 15.7 shows a lot of other useful manipulators, including to show as uppercase or lowercase, skipping white-space on input, show or don't show the base for integers, etc.

1.8 Left and right justify strings, padding

- use left and right manipulators to justify a sting output in a field.
- needs to be used in conjunction with setw(w) manipulator normally.
- Can pad out strings (instead of using whitespace when justifying) by using setfill('x')

1.9 Boolean output representation

- Can have booleans output as true/false (rather than 0/1) using boolalpha manipulator.

2 Second Session (11:45 - 12:30)

2.1 File Processing

For the most part, all of the stream I/O we have seen can be done to and from a file that you open and specify (instead of the standard input/output device). This is more complicated for a binary file we want to randomly access, but we will first look at opening a plain text stream, and reading/writing it sequentially.

2.2 Creating a Sequential File

- need to include fstream libray to open files for read/write
- At most basic, can open a file for output as

```
ofstream outFile("name-of-output-file.txt");
```

- And can open a file for input using:

```
ifstream inFile("name-of-input-file.txt");
```

2.3 Writing/Reading Data from a sequential File

- An open output stream file using sequential access, can be written

to using the name we just created. For example:

```
outFile << x << y << z;
```

- Likewise we can read input from a simple sequential file we opened:

```
inFile >> x >> y >> z;
```

- HINT: You need to be careful that you know what file you are opening and where it is located on your filesystem. I require that you always check whether the open of the file was successful after opening it:

```
ofstream outFile("name-of-input-file.txt");

// exit program if unable to create a file
if (!outFile)
{
  cerr << "File name-of-input-file.txt could not be opened, file not found error." << endl;
  exit(1);
}
```

- Also, never hardcode an absolute path to a file. Always use a relative path name.

- Once you successfully open a file for read/write, you can use any of the iostream methods we have talked about to format the input/output from/to the file.

2.4 Reading Lines of Data from a File

2.5 Reading comma separated values (CSV)

2.6 Random Access Files

- Create using `ios::binary` specifier

- use `seekp()` and write() read() member functions to get data in and out.

- `seekp()` need to specify a byte location to move to in file, need to calculate correctly or bad things will result.

- Advantages of binary files, more compact, faster to access (because of this)

- But overrated. Plain text files are human readable and editable. Storage is cheap, programmer time is expensive.

3 Third Session (12:40 - 1:40)

3.1 stdio.h, old school C I/O

- `printf()` function to do output and formatted output.

```
printf("x = %d, f = %f, c = %c\n", someInt, someFloat, someChar)
```

- Discuss table of format specifiers
- `scanf()` to do (formatted) input
- uses same table of format specifiers

3.2 File I/O with stdio.h

- use `fopen()` to create a file handle than can be used for reading and writing.

```
FILE *fileHandle;

fileHandle = fopen("file-name-to-open.txt", "r");  // or "w" for an output file to write to
```

- when done with file, always use `fclose(fileHandle);`
- can pass a file handle to `fprintf()` and `fscanf()` functions to read and write formatted input/output from/to files.

```
fprintf(outputFileHandle, "x = %d, f = %f, c = %d\n", x, f, c); // output values to plain text file
```

- again similar syntax for input using `scanf()`