CSCI 240 Notes - Input and Output

This section of the Notes contains several topics on Input and Output used in various parts of the course.

---

cin

cin allows the user to enter values into variables from standard input (the keyboard or via I/O redirection).

It can accept *one* or more values, depending on how many << are used.

Its behavior is slightly different depending on the data type being read.

When you code

```
int i;

cin >> i;
```

cin attempts to read an integer from standard input (let's assume the keyboard). It waits until the user presses <Enter> and then attempts to convert the keystrokes entered into an int, and then store it in the variable whose name is supplied.

So if the user enters

123<Enter>

the characters '1', '2', and '3' are converted by *cin* to the integer 123 and this value is stored into the memory occupied by *i*.

You can read several values with one *cin*. Suppose you want the user to enter integer values:

```
cin >> int1 >> int2 >> int3;
```

You would separate the numbers with any whitespace character when entering the values.

cin and Input Errors

If the data supplied by the user does not match what is expected by the program, errors can occur. Sometimes you will get no indication of them, except that the program will not work as expected.

For example, if the user is supposed to enter a number, but enters:

2w3<Enter>

cin stops at the first **non-digit**, and converts just what has been entered up to there; in this case the '2'. So the value 2 would be stored. No error would be signaled. In this case of 2w3 - the w, the 3, and the \n are still "waiting" to be processed by subsequent input instructions - which are probably expecting something else - maybe a string representing someone's name. So the "w3" would be read as the name.

The Keyboard Buffer

Keystrokes (chars) entered by user are stored in a special area in memory called the keyboard buffer. *cin* gets chars from there.

Suppose user types _123<Enter> //the _ represents a space

```
So in kbd buffer is:   '_'   '1'   '2'   '3'   '\n'    //as chars
Or                      32    49    50    51    10     //as ASCII values
```

cin takes chars from the start (the left), **removing** them as it processes them. Typing **adds** chars to the end (the right).

cin for numeric values (float, double, int, long int, etc.) works like this for keyboard input:

- starts its work when user presses <Enter>
- scans past leading "white space" (blanks, tabs, newlines)
- stops on first non-valid char
- converts digits (and +,-,.) into specified internal numeric format, and stores result at address given by argument
- does not check for input errors; just stops on non-valid character

cin for a character will scan past leading whitespace and then take the next char from the keyboard buffer if there is one. If it is empty, it lets the user type chars until <Enter> is pressed. Then it gets the 1^{st} char in the buffer.

Reading Multi-Word Strings

We have used *cin* to read strings (both char arrays and the string class). However, it is flawed for most purposes because it stops reading when it hits any whitespace character. So if the string you want to read consists of words separated by spaces, you'll only get the first word. You'd have to code additional *cin*s or << to get additional words in the string.

For example, to read a first, middle, and last name, you could code:

```
cin >> fName >> mName >> lName;
```

But if you don't know how many words are in a line, you are in trouble.

You can use a "method of *cin*" to read a line of text (up to the \n character) in one operation. This is object-oriented terminology which will be covered late in the course. For now, just know how to do this:

```
cin.getline( strVar, maxlen );
```

Where

- strVar is a string class variable or char array
- maxlen is a number specifying the maximum number of characters to store (if you are storing into a char array, there must be room for a terminating null. So if you specify 80 as the max, only 79 chars from input will be stored plus the null).

If you want to stop inputting characters when you hit some other character, that character can be a third argument. For example, to stop on a '.', code:

```
cin.getline( str, 80, '.' );
```

Incidentally, there is another method that will read the next character in the keyboard buffer whether it is a whitespace character or not:

```
ch = cin.get();
```

Where ch is a char variable. Note that *cin.get()* <u>returns</u> the value value of the char it reads.

File Input and Output

You may know that you can run a normal program to write output to a file or read input from a file using *I/O redirection*. When you do this, the user of the program must type the name(s) of these files on the command line. Also, you are limited to one input and one output file.

However, you can also have a program itself connect to and read or write from one or more input and/or output files. The techniques for doing this are based on object oriented principles which are not covered until late in this course, but the techniques are nonetheless simple to use.

Files are *objects*. You can think of an *object* as a special *data type*. The data type has a set of special functions associated with it that only work on variables of the particular type. The functions are called *methods* (just a different terminology). We will discuss the methods *open()*, *fail()*, *getline()*, and *close()* below.

So - how do we do this?

1. First, you need to

```
#include <fstream>
```

2. Then, for each file you want to use, you must declare a variable. The variable type (or the object type) is either:

- ifstream (for input file stream)
- ofstream (for output file stream)

The word *stream* is used because files are treated as a stream or sequence of values, one after another.

```
ifstream inFile;

ofstream outFile;
```

3. Then you must "open" or connect the program to the file.

```
inFile.open( "test.txt" );
```

or

```
outFile.open( "output.txt" );
```

Because *inFile* is of type *ifstream*, the open method prepares the file to be *read*.

Because *outFile* is of type *ofstream*, the open method prepares the file to be *written*. (If there is already a file called output.txt it will be overwritten.)

There are some optional arguments to *open()* that are covered more fully in the textbook, Chapter 14. For example, you can arrange to *append* written data to an existing file.

4. Check that the *open()* succeeded. To do this, code:

```
if ( inFile.fail() )     // or outFile.fail()
  {
  cout << "open for test.dat failed";

  // exit the program
  exit(1);
  }
```

Note: on some systems, the *exit()* method may require <stdlib.h>. Alternately, if the open is in *main()* you could just *return*.

5. Now you can read and write information from/to the file. For example:

```
inFile >> num;

outFile << "Hello";
```

5a. Reading strings with embedded spaces presents some problems, since an *ifstream* will stop at the first space. You can use the *getline()* method (function) to read such strings - see the example below. Alternately, you can use multiple input operations if you know how many "words" you want to read:

Example: if your input looks like this:

Chicago IL

```
inFile >> str1;

inFile >> str2;
```

then str1 will have "Chicago" and str2 will have "IL". This will be true even if there are extra spaces or even newline chars between the two words.

6. Close the files. On some systems (especially for input files) this may not be necessary, but it is considered good programming practice:

```
inFile.close();

outFile.close();
```

Example: the program below copies the contents of one file to another. It reads and writes strings. When a string is read from a file, the newline character is not put into the string data item in the program, so when we write each string out to the second file, we have to add it back in. Also, we use the regular console *cout* to echo the data to the screen (in addition to writing it to the file).

Notice that there are two versions of the read-write loop. The first uses a sentinel value to signal end-of-input (a line with the single character 'x'); the second tests a special return value from getline:

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
ifstream in;
ofstream out;

char s[80];

//Note the two \\ needed to specify one \ in the string.
//Otherwise, the \t would be taken as a tab and the \o would be an undefined
//escape sequence

in.open( "c:\\temp\\original.dat" );

if (in.fail())
  {
  cout << "open for original.dat failed";
  // exit the program
  exit(1);
  }

out.open("c:\\temp\\newCopy.dat");


//read-write loop version 1:
//assumes that the last line in the file is "x" This code will NOT
//copy the "x" line

in >> s;

while ( strcmp(s, "x") != 0 )
  {
  cout << s << "\n";

  out << s << "\n";

  in >> s;
  }


//read-write loop version 2:
//uses the function getline().  See note below. This version will copy
//the last line (the "x") since it reads data until all of the
//records have been read from the file.

while ( in.getline(s, 80) )
  {
  cout << s << "\n";

  out << s << "\n";
  }

in.close();
out.close();

return 0;
}
```

Note: version 1 above stops reading chars at any whitespace. So a line consisting of several words separated by spaces would require several reads (in >> s). *getline()* reads a whole line up to a newline character (or other delimiter which can be specified as an optional third argument) so you can easily read such a line.

So *getline()* reads one line from the file *in* and stores it into *s*. The function returns:

- *true* when it *can* read a line and
- *false* when it *can't* read a line (because there are no more lines)

So the return value is used to exit the loop upon end-of-file.

I/O of Structures - Block I/O

To read and write structures to disk, you could read and write the individual data members in order. However, this is inefficient and awkward. It is much neater and more efficient to input and output a whole structure at a time, with one operation.

Assume you have a structure defined and created in your program. Let's say the structure name is *Person* and the structure variable is person:

```
struct Person
  {
  char name[40];
  int id;
  };


Person person;
```

Supply a

```
#include <fstream>     // Note: just fstream (not if- or of- stream)
```

Writing a structure

1. Declare the file and open it for output:

```
ofstream outfile;

//Note the "ios::binary"

outfile.open( "people.txt", ios::binary );
```

Don't neglect to check for failure. Use the *fail()* method of ofstream:

```
if( outfile.fail() )
{
...
}
```

2. Populate *person* with data:

```
strcpy(person.name, "Joe");

person.id = 1234;
```

3. Write the structure to disk:

```
outfile.write( (char *)&person, sizeof(Person) );
```

Note the typecast of *&person* to a *char* *. *&person* is just an address, but the write method needs a *char* * argument. *&person* doesn't say what kind of data the address is pointing to, but the function insists that it be a pointer-to-char. So the typecast does that.

So the first argument tells *write* <u>where to find the data</u> to write, but notice that it doesn't say "this is a Person thing". It just says where the Person begins.

The second argument tells *write* <u>how many bytes</u> to write out from that beginning address. In this way, *write* will output the correct number of bytes for just one Person.

4. Do steps 3 and 4 as often as needed to create a file of Persons.

5. Close the file:

```
outfile.close();
```

Reading a structure

Assuming you have written a file of Person structures with a program based on the information in the previous section, you can write a program to read these structures back from disk in a similar way.

1. Declare the file and open it for input:

```
ifstream infile;

// Note the ios: stuff

infile.open("people.txt", ios::binary);
```

Don't neglect to check for failure. Use the *fail()* method of ifstream:

```
if (infile.fail())
{
...
}
```

2. Read the data until end of file:

```
infile.read((char *)&person, sizeof(Person));

while (infile)
  {
  // do whatever with this person: print it or ...

  infile.read((char *)&person, sizeof(Person));
  }
```

Note that the value of *infile* is tested as the loop exit condition. *infile* will be true after a successful read and false when end-of-file is reached.

3. Close the file:

```
infile.close();
```

Building an Array of Structures from input

You can declare an array of structs and then fill it with structs read from disk.

1. Declare the array:

```
Person people[80];
```

2. Open the file as usual, but the read commend will look like this:

```
infile.read((char *)&people[i], sizeof(Person));
```

Of course, the subscript *i* will need to be initialized to 0 and will need to be incremented after each read so that the next read puts the next struct into the next array element.