COSC 2P95
IO

Week 4

Brock University

Principles of IO

There are several ways to accomplish Input/Output communications, but there are some common recurring themes.

- Input tends to involve reading some number of bytes into a buffer, tokenizing or otherwise splitting them, and then parsing them into useable data
- Output typically involves queueing up a sequence of bytes into a write buffer, which are then spit out at appropriate times
- More generally, there is typically a *stream* of bytes that may then be abstracted to derive additional organization or meaning
- Data may be either read/written in a human-readable text-based format (e.g. ASCII), or dumped as raw binary
    - When trying to write complex records as a byte stream, that's called *serialization*
- Some methods of data access may allow for a *cursor*, or otherwise random access

Principles of IO

And... that means?

Basically, we don't usually want to deal with high-level algorithms simultaneously with low-level byte transfers.

- Instead, we typically prefer to have some mechanism for providing most of those low-level features for us, unless there's a specific *need* to do things manually

Thus far, we've taken a look at `cin`, `cout`, `cerr`, and `clog`, but we haven't really gone into very much detail explaining them.

Formatted Stream Classes

C++ includes several levels of *stream classes* to help you with simple and convenient IO. The C++ reference shows that there is actually quite an elaborate hierarchy involved (`http://en.cppreference.com/w/cpp/io` or `http://www.cplusplus.com/reference/ios/`).

- You have *input streams* and *output streams*, as well as *input/output streams* (which may be used for both read and write access to the same stream)
- Streams may be attached to files
- String buffers can also be used for streams
  - e.g. to construct a complex string, piece by piece, and then retrieve it all at once
- Because streams are handled with classes, they can not only hold various state informations, but also be told to behave differently (e.g. outputting numbers in hexadecimal, instead of decimal)

`cin`, etc., are simply predefined named streams.

Output

We can handle output with an `ostream` (of which `cout` and `cerr` are examples)

- We've already seen how to use stream insertion (<<)
- We've covered displaying numbers as text, but there are far more formatting options available to us!
- `ostreams` may have their properties changed by either setting their internal *flags* (via `setf`; simply use | to combine flags), or by introducing *manipulators* via the stream insertion operator
- This includes things like defining the width allocated for fields, the level of precision for numbers, text justification, number systems to use, method of representing booleans, etc.

Output — References and Examples

Again, take a look here: `http://en.cppreference.com/w/cpp/io`
And also: `http://en.cppreference.com/w/cpp/io/basic_ios`
`http://en.cppreference.com/w/cpp/io/manip`
and: `http://en.cppreference.com/w/cpp/io/basic_ostream`

(Though largely the same, I'm also partial to
`http://www.cplusplus.com/reference/ios/` ,
`http://www.cplusplus.com/reference/library/manipulators/` ,
and `http://www.cplusplus.com/iomanip`)

Basically, we can format via *manipulators*, *flags*, or member functions.

I think this is best explained via demonstration.

Input

We've already used a named `istream`: `cin`.

In some ways, it's a little more complicated than output, since it needs to buffer, tokenize, and parse.

- There are additional functions provided for getting a single character/byte (`get`/`read`) , or an entire line (ignoring the common delimiters, like spaces)
- *However*, be aware that, typically, your bash terminal will provide *line-buffered input*
  - Your program may require only a single character at a time, but it will still keep perpetually waiting until you actually press enter
  - Note that this assumes a keyboard input for *standard input*, and that you haven't changed any terminal settings (e.g. via `stty`)

Howsabout an example?

Status Flags and Validation

When asked for a number, what happens when you answer *turkey*?

Outside of minor issues (like giving a number when asked, but outside of the specified range), there are two basic approaches to validating input:

- Read input into a string, and then verify each character, one-by-one
- After reading in, check the *state flags* of the istream
  - If they indicate a problem, you'll likely want to either discard input or rewind, followed by clearing the error bit and trying again

Also, you can tweak your input in similar ways to ostreams, ignore whitespace, etc. If desired, you can also explicitly check for EOF.

Example time!

String streams

Like many languages, C++ offers tools for IO-like actions, but based on streams to/from strings, rather than the console or files.

- For the most part, `stringstreams` (or `ostringstreams`/`istringstreams`) are used in the same way as any other stream

This one really isn't tricky. Let's just go straight to the example.

File IO

For the most part, dealing with file streams doesn't need to be more difficult than any other streams.

The biggest difference comes from how we (might wish to) access them.

- Instead of `istream`, `ostream`, or `iostream`, we'll be using `ifstream`, `ofstream`, or `fstream`, respectively
- Try to avoid opening files for both input and output unless you actually intend to use both
- When writing to files, consider calling `flush()` every now and then, and also `close()` at the end
  - In theory, the destructor will ensure that the file stream is closed anyway, but there's little benefit in gambling with write caches

Take a gander for different ways to open a file stream:
`http://en.cppreference.com/w/cpp/io/ios_base/openmode`
Example? Example.

Random File Access

Basic stream reading/writing is pretty good for many tasks. However, sometimes you need a bit more control.
Consider the task of trying to change a single byte within a file. Is there any need to read the entire file, just to dump it all back out again? In some cases, it may be useful to be able to jump from one place to another, or to return to a previous location. We can do all of these things.

- File streams have their own internal positions (effectively cursors)
  - If a file is opened for both input and output, then it has two such cursors
- You can request your current position, jump to certain defined points, or jump ahead (or backwards) by relative distances

It's been *so* long since we've looked at an example!

Delimiters

Generally, the default delimiters are adequate for our needs. However, since we'll sometimes want to do something unusual:

- You can use `skipws`/`noskipws` to indicate if leading whitespaces should be ignored
- `cin` doesn't make it pleasant to change its delimiter, though `getline` isn't too bad

IO in C

On occasion, you'll find yourself using the lower-level IO from C. Though generally (much) less pleasant, the provided functions do allow a few interesting tricks (including more elaborate, though more fragile, input filtering).

- Most of the differences are cosmetic. Things like buffering, delimiters, etc. are largely the same
  - A few of the C++ features (e.g. append) are C with a facade

printf and fprintf

The easiest way to produce text in C is via the *formatted print* statement.

It's a variadic procedure that first accepts a string, containing both text and replaceable symbols, followed by arguments to replace those symbols.

The fprintf statement works the same, but operates on a file descriptor. This can be dual-purposed to use stdout or stderr as the descriptor.

Of course, these functions used the old C-style strings (i.e. null character-terminated character arrays).

Unlike the C++ IO functions, this requires including the stdio.h header.

`scanf` and `fscanf`

Input is achieved via the *formatted scan* statement.

- It works in a similar fashion to `printf`, first starting with a string indicating what to expect, and then followed by arguments
- The arguments are pointers, so the function can simulate a ‘multiple return’

`sscanf` and `sprintf`

C also included functions for IO-resembling operations that acted on strings instead of true streams.

`sscanf` was used to extract tokens from within a string, while `sprintf` was for constructing strings based on supplied components.

—

Generally, you wouldn't use any of these throwback functions unless you wanted to make use of the formatted replacement strings as an unusual filter.

A comment on Bash

Because we're talking about both console and file IO, it's worth reminding that we can blur this line somewhat in Bash.

- We have redirection operators that we've partially touched on already:
  - > dumps output to a file
  - >> appends to an existing file
  - < uses the contents of a file to replace *standard input*
- Also, we've briefly discussed how we can *pipe* the output stream of one process into the input stream of another

Basically, the point is that, sometimes, Bash can take the place of both file IO and text processing/formatting.

Mind if I just cover one more quick example?

Questions?

Comments?

- Recipes?