COSC 2P95

Introduction

Week 1

Brock University

Lectures and Labs

- Lectures are Fridays, from 3pm–5pm (TH245)
- There are two lab sections
  - Lab 1 is Tuesdays, from 4pm–6pm
  - Lab 1 is Fridays, from 9am–11am

Labs will be conducted in MCD205, *booting into Linux*.

Course Sections

Unlike most courses at Brock, the two lecture sections actually correspond to differences in focus (and assumed background).

- If you are taking this course with MATH 2P40 as the prerequisite, then Section 01 is suggested for you.
- If you're using COSC 1P03 as the prerequisite, then Section 02 is suggested.
- If you're using an APCO course as the prerequisite, then neither is specifically more advisable than the other.

Lecture content is identical between the two courses, but (as we'll see on the next slide) evaluation is handled differently. Also, the two sections have their own webpages.

- Section 01 is administered through Sakai
- Section 02 is on the main COSC website

Evaluation

Section 01 (MATH)

Labs $\times$ 10	10%
Lab Exercises $\times$ 10	50%
Exam	40%

Section 02 (COSC)

Labs $\times$ 10	10%
Assignments $\times$ 3	54%
Exam	36%

Refer to the course outline for additional rules for the labs and exam.

Overview of Topics

Across the term, we'll touch on several of the following subjects:

- An introduction to C/C++
- Types, enums, operators, and expressions
- Flow control
- Procedures and functions
- Timing, threading, processes, and interprocess communication
- Streams and IO
- Compiling, make, and libraries
- Data structures and memory allocation
- Object orientation
- Templates

What is C++?

(Surely you saw this question coming)

This is actually a very difficult question to jump straight into. Why? Because the answer (arguably) is:

> *A twisted corruption and defilement of C*

Helpful, eh?

What is C++?

No, really, let's try to answer this one!

C++ was a language created by Bjarne Stroustrup as a modification of the C language. Primarily, it added some basic convenience features, as well as the very obvious addition of Object Orientation*.

This, of course, means we need to discuss C a bit.

* Note: What one describes as "Object Orientation", another could describe as "a nonsensical and inconsistent mess of preprocessor directives and structs". You know, tomato-tomato (as if anyone *actually* pronounces it *tomato*!)

What is C?

(Ritchie don't hurt me, don't hurt me, no more)

C is often described (inaccurately) as a low-level language.
It's true that C is a much lower-level language when compared to C#, Java, Python, Ada, etc. However, it really served as an early example of a *portable* high-level language.

Do we know what *portable* means?

Architectures and Assembly

(And why we'd like to avoid them!)

There are lots of reasons to not want to write programs in assembly.

- You get comparatively little for the same number of lines of code you write
- You need to explicitly write nearly everything out
- It's very difficult to read
- etc. etc. etc...

But there's one thing that assembly started to introduce over writing in machine code: the code was somewhat portable.

> *An instruction, or set of instructions, written once, can then be used to generate machine code for multiple devices, with multiple architectures/instruction sets.*

Consider something like a simple `add` statement. Maybe it's 16 bits, maybe it's 32. Who knows what the precise bit pattern is to invoke it? But all you need is to be able to spell `add`.

What does this have to do with C?

This same principle extends to C, and quite fantastically.

There are multiple *standards* of C (primarily based on the year in which they were formally released), but beyond that, with the right compiler, code written once can be targeted at numerous different platforms.

Compare how this works in Python? In Java?

Okay, but what *is* C?

C is a statically typed language created mostly by Dennis Ritchie.

- Yes, it was the language that came "after B" (from Bell Labs)
- Its history is rooted in use for telecommunication switching, and then the development of UNIX
  - Ever wonder why *switch* statements always seemed so odd? That's why

If you're interested in more of the history, your best bet would probably be to look up Ritchie and Stroustrup themselves.

So, that helps us understand C++?

Kind of?

C was a language intended to help control telephone hardware, that was slowly and incrementally adapted to eventually be used for the creation of an entire operating system.
As software design progressed, C started to become more and more cumbersome (not to mention mildly unpredictable).
Structs and function pointers could help, but that's not enough for a true evolution of software design.

So, C++ is a kludge?

Again, kind of?
A new language was created as a new iteration of the original. An *increment* of C, you could say. Like $C + 1$.

- As mentioned earlier, the most significant addition with C++ is the introduction of (a form of) Object Orientation
- This allowed developers to model systems with modern design patterns
  - which, of course, also fostered the development of even newer design patterns

Yes, it's a kludge. But it's a kludge that largely hides its kludginess (totes a word) from developers. That's enough.

So, why are we talking about C so much?

Ask any bona fide C++ aficionado about how to use C++, and there's a good chance that the first thing they'll say will be *"don't think of C++ as C with Object Orientation"*. And that's the right opinion to have as a seasoned professional.

To learn, though? We're going to be treating C++ as an expansion of C. (And, honestly, all popular languages have their own preferred patterns and style guides. Absolutely none of them are particularly useful until after you've learned the basics)

What else?

Our development platform for this course will be Linux. The labs have Red Hat installed, though really any Linux environment should really work comparably for our purposes. This includes sandcastle.
With the right additional software added, you could also use cygwin or MobaXterm in Windows.
Either way, part of this course will include using the Bash shell (Linux/Unix command-line). By the end, you should be at least reasonably comfortable doing basic tasks without a GUI.

Final thought on Bash

I think we're mostly good to wait until the first lab for this, but are we all at least familiar with the basic appearance/usage of Bash?

Do we know how to find help on commands?

Do we have any idea what sorts of interesting things we can do with processes and piping?

Do we know what `~` is? `*`? `??` `\`? `cat`? `alias`? `touch`? `mkfifo`? `grep`?

(Actually, I'm genuinely asking. If possible, it would be good to get an idea of how familiar you all are, or aren't, with the environment you'll be using. I'm happy to fill in any gaps in the first lab)

Editor, compiler, IDEs, etc.

We'll be using the Gnu compiler (gcc/g++) for this course. That means we'll mostly just be using a plaintext editor of some sort.
Feel free to use an IDE's editor as a text editor if you prefer, but when we get to things like libraries and *make*, you'll definitely need to use the command-line entirely for compiling.

Questions? Comments?

We'll start actually learning how to write programs next week.
But for now, I'm thinking we must have questions?
Also, this is the very first offering for this course (ever), so if there's anything you'd like to see done differently, please don't hesitate to make suggestions.