COSC 2P95

The Basics

Week 2

Brock University

Compilation

We'll get into this a little more when we discuss using multiple source files, but we need a cursory knowledge of how compilation works.

There are three particularly significant components involved in the compilation process:

- The preprocessor
  - Directives are given to a program, largely to do concatenations and substitutions, with minimal control logic included
  - Think of it as translating one form of source code into a different one
- The compiler
  - The compiler takes the amended *translation units*, and compiles them into *object files*
  - Like executable machine code, but still missing libraries, or connections to each other
- The linker
  - Connects object files, and libraries, to make an executable

There might also be an *assembler* involved

So, what are we writing?

We're writing the source files. For the most part, we usually won't worry about anything lower than that.

- In C, source files typically end with `.c`, while *header* files end with `.h`
- In C++, `.cpp` is the most common (though `.h` is probably still more popular than `.hpp`)
- Decent compilers are capable of accepting different *targets* (for cross-compiling), as well as following multiple *standards*

About source files...

The source files themselves are simply plaintext. It's also worth knowing:

- As hinted at earlier, programs can be broken up into multiple files, for logical separation and modularity
  - For C, this can be done however you feel appropriate
  - For C++, it's highly advisable to model solutions as you would for any other Object-Oriented language
- The result can be distributed as a library, or simply be an application
- Each file could contain any of: preprocessor directives, constant and variable declarations, external/static variables, procedures/functions, etc.

Also, you can include a `main` function, to represent the entry point into the application.

Compiler

For this course, we'll be using the GNU Compiler Collection (`gcc`).

- GCC actually supports a wide range of languages
  - It's not unheard of to 'make a compiler' by writing software to first translate another language into C, and then compile it
  - Specifically, we'll be compiling via the command `g++`

Of course, there are numerous other compilers, but GNU is pretty popular for Unix-derived systems.

`gcc` offers several advanced features (including stopping before the linker phase to invoke it manually at a later time, choosing different levels of compilation optimization, and targeting entirely different architectures), but we're not likely to need those any time soon.

Final thought on gcc and gpp

Please remember that you have a valuable resource at your disposal in the `man` pages.

- You aren't expected to remember how to include libraries, or turn more warnings on, or activate pedantic mode

The Preprocessor

Though we'll be learning more about this later, we probably can't avoid using the preprocessor at least a little.

Basically, the preprocessor can perform substitutions akin to running *macros*. For example, two important directives are:

- `#include`
  - Similar to, though distinct from, an *import* in Java or Python
  - Closer to copying and pasting code from other sources files into the current one, but without needing to actually include it
  - Good for forward declarations/function *prototypes*, constants, and typedefs
  - This facilitates the use of multiple source fils, but can also introduce its own conflicts
- `#define`
  - Primarily defines a symbolic replacement
    - One use is for *include guards* (more on this later)
  - Another is to define a literal to use as a subsitution for a term
    - Almost like a constant...
    - Please try to avoid this usage, when possible

I can haz exampleburger?

Yes! You can!

I think this is a good time for our first sample program.
How about something *completely* original?

```c
#include<stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello class!");
}
```

Hey! That was C!

Ya got me. That was C. Let's see just how different C++ can be!

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello class!");
}
```

Are you mocking me?

Perish the thought. Remember, C++ isn't just a C-like language. It's mostly C-compatible!
Still, let's look at how you'd write it in C++ (for real):

```
#include <iostream>

int main(int argc, char *argv[]) {
    std::cout<<"Hello class!"<<std::endl;
}
```

Should we explain that?

I think we probably should...

- `#include <iostream>` — we tell the preprocessor that we'll be using IO from the *Standard Template Library*
- `int main` — `main` actually has a return. After exiting, it tells the OS whether or not it exited on an error
- `int argc` and `char *argv[]` (or `char **argv`) — this is how we'll be able to pass in additional command-line parameters
- `cout` — an object wrapping around the *standard output* stream
- `<<` — a stream insertion (that returns a reference to the same stream)
- `endl` — a newline character
- `std::` — we wish to access something within another *namespace*

Values and types

C++ supports a pretty wide range of types. Generally, operations will require matching types, so be aware of typing, even when only dealing with values.

- `int`
  - `short int` (or `short`)
  - `int`
  - `long int` (or `long`)
  - `long long int` (or `long long`)
  - `sizeof(short)`≤`sizeof(int)`≤`sizeof(long)`
- `char` and `bool` are a little special
- Floating point
  - `float` — typically 32 bit
  - `double` — typically 64 bit
  - `long double` is also a thing, but less common
- Are these signed or unsigned? How do we define strings?

Using literals

More often than not, if you want to use a value explicitly, you can just type it.

However, there are some times when you might need to be less ambiguous:

- You can append suffixes like U, L, or sometimes LL to values to ensure that they're unsigned, long, long long, etc.
- If you want to ensure that a literal will be treated as a floating point value, make sure it has a decimal component
- You can express your integer values as:
  - Base 10 — by just typing the number
  - Octal — with a leading 0 (e.g. `x=013`)
  - Hexadecimal — with a leading 0x (e.g. `x=0x0B`)
  - Binary — as of the C14 standard, with a leading 0b (e.g. 0b00001011)

Casting and coercion

Suppose I have an `int k`, and I'd like it to have the contents of `double d`. Could I just say `k=d;` ?
Maybe this is a good time to explain the difference between *strongly-typed* languages, and *weakly-typed* languages...

Still, we can do the conversion explicitly with a cast: `k=(int)d;`

Pop quiz! What would you expect from: `2/3` ? What about `2/3.0` ? How about `2.0/3` ?

Variables and assignment

I think it's finally time to start declaring some variables!
A *variable* represents the label for a place in memory that can be used to hold data. Its symbol is associated with a *type*, which must be declared from the first creation of the variable.

- e.g. `int k;`
- or `int k=4,m,n;`
- or `char c=-'A';` (why is this a thing?)

This brings up our first official operator: *assignment*.
An assignment has a left-hand side (LHS), and a right-hand side (RHS). An RHS may be an expression or a variable, but an LHS can only be a variable.

Fun fact: an assignment is still an expression, and returns the value being assigned. This allows for *multiple assignments*! (example time!)

Scope and extent

This will be of somewhat limited use until we address functions and using multiple source files, but we still need to know how *scope* and *extent* work. (but first, do we remember *what* they are?)

- Block-level variables are *automatic* (`auto`) by default — they're allocated upon entering the block, and deallocated upon leaving the block. Describes *most* local variables
- A variable at the file-level (global-ish) scope will persist for the translation unit's duration of use. It's effectively *static*, but that's different from the `static` keyword (it actually has several uses)
- Now that we've already started to introduce namespaces, scope is already a bit more complicated. In a few weeks, we'll also introduce an *external* (`extern`) variable
- `volatile` is used to mark variables that might be accessed by external threads or modules
- You can also define a variable to be `register`, but this is only a suggestion to the compiler

Why's it called *cout*?

We'll get to proper IO later on, but first we need to understand what's going on so far:

- As mentioned, `cout` is a stream object for *standard output*
  - << defines a special *operator* on the class for stream insertion
- `cerr` behaves the same, but is bound to *standard error*
- `clog` is basically like a buffered version of cerr: good for dumping large volumes of text, but not really interesting to us
- `cin` is an input, and reads from *standard input*

Do we mind taking a quick glance at an example?
We need a bit more practice with Bash anyhoo.

Expressions and Operators

Of course, we know what an *expression* is: some sequence of values, variables, and operators that can be evaluated to return a value.

Which operators are we familiar with?

- `+ - * /`
- `%`
- `++ --`
- `& | ^`
- `<< >>` (can we see this being problematic?)
- `~ !` (are these the same thing?)
- `&& ||` (*totally* different from `&` and `|`!)
- `,` (are `a=b=3,c=4;` and `a=(b=3,c=4);` the same? EXPLAIN!)

Constants

There are three basic methods of defining constants:

- `const` — a keyword prepended to a variable declaration as a promise not to change it (don't forget to assign a value)
- By using the `#define` preprocessor directive, to make a substitution of the literal for the term
- `constexpr` — for constant expressions (variables/functions that can be evaluated at compile time)

Strings

We aren't going to get too much into strings yet.

- C provided a very minimal solution, supplemented by libraries
- C++ provides a proper `string` class as part of the Standard Template Library
  - Depending on the C++ standard, there might be some peculiarities for when it copies references, and when it perofrms a 'copy on write'
  - For now, just know that C++ strings are closer to Java and Python than C

Booleans

We've already talked about booleans (the `bool` type), but we haven't actually used one yet.
As with most data types in C++, the size of `bool` isn't defined by the standard (instead left up to compilers to decide), but 8 bits isn't uncommon. Think about the implications there...

Example time? I think it is.

Flow control

Conditionals

Most languages have some form of an *if* statement. C++ is no different.

- We have an `if` and an `else`, though no *elif* or *endif*
- Surround the boolean expression with parentheses
- You may follow the condition with a statement, or a block
  - (Watch out for semicolons... *mwahahahaha*)

Dangling elses

(Yes, that's really a thing)

One small thing to watch out for when using nested conditionals is the concept of a *dangling else*.

Consider the following code:

```
if (a)
    if (b)
        cout<<"Yes, a and b"<<endl;
else
    cout<<"Uh... help?"<<endl;
```

- Languages like, Ada and Bash script, use explicit terminators for conditionals (like *end if* and *fi*, respectively)

Ternary conditional operator

- ?

Switch statements

In C++, switches are used as a shorthand alternative for multiple else-if cases, instead matching up labels with the evaluated expression to test.

```
switch (dealie) {
case 1:
    cout<<"Only run this on 1\n";
    break;
case 2:
case 3:
    cout<<"Run this on 2 or 3\n";
    break;
case 4:
    cout<<"Only run this on 4, but continue to 5, too\n";
case 5:
    cout<<"I'm a 5\n";
    break;
default:
    cout<<"I like turtles\n";
    break; //because I can
}
```

Loops

Is there anything here we should go into more detail on?

- `for`
  - Typical `for` loops
  - `for (;;)`
  - `for (type c:s)` (range-based; only in more recent standards)
  - Could we write a single loop with two counters approaching each other?
- `while`
- `do ... while`
- `break` and `continue`

Labels and the `goto` statement

No.

Bitfields and Bitmasks

Suppose you wish to store some number of booleans. What data type would you use?

- Even `char` seems like a waste of space...
- `bool` is now an option, thanks to C++, but may or may not pack well

If you're dealing with *many* records (or saving many records to a disk/database), then it may make more sense to pack multiple booleans into the same integer value (e.g. 64-bit).

The bits can be set and extracted via bitwise operators, typically abstracted out to functions (with the bit position being provided as an index argument).

Questions?

Comments?

- Did you notice that *Brock Univeristy* is misspelled in the lower-left corner?