COSC 2P95

Shh! This is a library! (nyuk nyuk!)

Week 8

Brock University

Organizing solutions

Hey, know what we totally haven't discussed at all yet?
Reusing code to avoid redundancy, improve readability, or better encapsulate information and behaviour.

- Nope. Not at all.

Yeah yeah, only kidding. But we still have more to learn on the subject. In particular, there are three things we still haven't addressed:

1. How to effectively separate solutions into multiple files (there's more we can do than just OO)
2. How to bundle up functional units into libraries, and make use of those libraries in the future
3. How to manage complicated projects, particularly for efficient compiling

Using multiple source files

Before we get to the fun stuff, let's first discuss *translation units* again.

- Each source file is compiled into a separate object file, which can then be *linked* by the linker. But how does linking actually work?
  - For example, for one translation unit to make use of code in another one, does the former *need* to include the header of the latter? Think carefully about it
- Two keywords that we probably won't need to use *too* often on a file-scope level are `extern` and `static`
  - You can use `extern` to declare (without needing to define) an *external* variable, which can be used to extend wider than a normal declaration (e.g. a file-scope variable within a function, or even in another translation unit). Of course, it still needs to be defined *somewhere*
  - A `static` function is hidden from the linker
- This is also probably a good time to learn how to use `namespaces`

(Example time. Of course)

Sample problem

Because we'll be looking at different variations of the same thing, it will help to have a base example.

- Normally, we'd like to demonstrate with data structures, graphics, IO, etc., but those tend to be templated, and we want simple headers for illustrative purposes
  - Basically, we're going to use *birds* and *dinosaurs* for our examples. Because of reasons

Let's look at two (totally independent) sample programs, which each rely on some (totally independent) reusable code.

(Yes, this means example-time again)

Reminder on compiling

There are two things we learned a while back that we should still remember:

- You can compile to *object files* (foregoing the linker) by using the `-c` flag
- You can then run g++ just on object files to automatially invoke the linker, and create the final executable

Include guards

There are certain combinations of `#include` lines we wouldn't be able to use.

- e.g. though `Velociraptor.h` gives us access to Dinosaur, we can't explicitly include the paleontology header
- What if we wanted to define an *Archaeopteryx* as being a combination of a velociraptor and a bird?
  - Again, we aren't permitted

The problem is that, when we have multiple translation units relying on the same headers, sooner or later we tend to define the same thing twice.j

However, this *shouldn't* need to be a problem. Since it's triggered by hitting the same pieces of code repeatedly, it's arguable whether there should really be a conflict.

The standard fix is an *include guard*, though there's a *pragma* (special preprocessor directives outside of the language standard) that's sometimes more popular. ... Example time?

Libraries

Okay, so now we can easily get everything compiled and running.

We can easily write different programs that make use of one or more parts of the different categories of support files.

Overall, (and especially thanks to our namespaces) we can easily see three different categories of support files, separate from the client code.

- What we're really talking about is *libraries*
  - We've been using libraries for weeks now, but this is the first time we've come this close to creating a proper one

Library usage has two basic parts:

- Bundling everything up into a single file, for easy reference later
- Linking an executable to one of those bundled files

That said, there are different ways to do that linking/distribution.

Static libraries

So far, we've already seen that we can compile individual translation units into object files. We can then explicitly link all of those object files to create an executable.

- A *static library* isn't very different from this. We bundle the object files into a single library, and then provide that library to be linked to, and included with, with the executable
- Under 'nix, such static libraries (or archives) commonly end with a `.a` extension

Of course, there are some downsides to static libraries. e.g.:

- If ten programs use the same static library, you now have 10 copies of that library
- Good luck upgrading the library once, to update all of thsoe programs
  - Of course, the counterpoint is that a program can confidently rely on a single specific version of a library

Let's take a look, shall we?

Dynamic libraries

Alternatively, we can create a library once, put it in a widely-accessible location, and then allow numerous applications to all reference that single library.

This is actually immensely common. Though you might not be familiar with shared object (`.so`), you've probably seen the Dynamic-Link Library extension: `.dll`.

- A single copy requires less hard drive space than a dozen copies
- It's far easier to update a single library, to improve all of those applications
  - (Hopefully those applications weren't relying on a specific version)
- The library needs to distributed as an additional file
- Because the library is separate and could be replaced, you don't know the absolute address of functions at compile-time. Because of this, *position-independent code* allows the library functions to be relocated

Note: important: Because you have a single copy of the library, it *needs* to be in an accessible location. Again, we should see this in action.

Don't lose your head(er)!

When distributing libraries as development tools (so, some shared objects, and pretty much any static libraries), don't forget to also include the header files!

Additional libraries

When compiling programs/libraries, just as you can create and include your own libraries, others have already written their own libraries you can include.
You can generally include them when compiling in a similar fashion to how you included your own.

Organization

I tried so hard, and got so far. But in the end, it doesn't even compile

Though an incredibly small example, even what we've seen so far can be a mild nuisance to coordinate.

As solutions get more and more complicated, management and compiling can become more tedious.

- If nothing else, determining what does or doesn't need to be compiled can be the difference between a few seconds, and several minutes (or longer)

Basically, we want to things in a new tool:

- The ability to easily control what happens to the current build (including removal of intermediate files)
- The ability to intelligently decide whether or not to include a file, based on whether or not it's part of the required dependencies, and if it's been changed since the last compilation

Make

Make is a tool that coordinates the building, cleaning, and even installation of projects

- Note that we don't really get proper project management yet; there are better tools for that

What we do is we define different possible *targets*, which can represent final products and/or intermediate dependencies, and then detail how to proceed to get every required target satisfied.

Makefiles

At the heart of the make took is a *makefile*.

If needed, you can have multiple makefiles; otherwise the make tool will look for makefiles in the sequence of `GNUmakefile`, `makefile`, `Makefile` (I'm partial to the latter).

Makefiles list tuples of *targets*, and the dependencies that need to be satisfied for those targets.

- Some targets are chosen from the command-line
  - e.g. `make clean`
- If a target requires additional files (e.g. object files or libraries) as dependencies, then those file can, themselves, also be targets
  - In fact, this is the normal way to use targets; they're files that make will look for, and it'll try to satisfy them to create those files. If the target never will be a file (e.g. *clean*), then it's a handy way to ensure that its corresponding block of commands will be executed
- The existence of a file may be sufficient to satisfy a dependency, but what if that dependency is newer than the target?
  - e.g. if a library depends on a header, and the header's changed, then couldn't the library be outdated?

Makefile targets

Though the make tool itself is extremely flexible, you still tend to see certain usages as common practice.

- When you run the make tool without any parameters, it attempts the *default target*
  - The default target is the first normal (doesn't start with a period) target found in the makefile
  - Style guides say to call it `all`, but you can call it whatever you like. Generally, it fully compiles an application (or library)
- A common target for installations is `install` (to be run after the first `make`). It traditionally copies the compiled executable to somewhere in the normal application path, and, if necessary, any additional libraries into the library folder
- A *clean* target is often included, so that, if a previous attempt at building was unsuccessful, or no longer appropriate for the current needs, all generated files can be removed

Makefile targets
(continued)

- Remembering that a target will be satisified once the corresponding file exists, if a target an a file happen to share the same name, that's bad
  - You can either give it another dependency using a name that really won't ever exist, or declare the target as a dependency of `.PHONY`
- Phony targets might be included as dependencies of other targets, if you want to include them as part of the build process
  - e.g. perhaps a *rebuild* first includes a *clean*
- It's not unheard of to include separate targets for compiling the entire project in either a *debug* form, or a *release* form
  - Typically, if you don't specify otherwise, release would be the default, and debug could be an extra option

Makefile recipes

For each target (that actually does anything), you then define a *recipe*. A recipe's really just a list of normal shell instructions, to reflect what you'd type if you were doing that portion manually.

- `#Comment with an octothorpe`
- If a recipe encounters an error (e.g. an actual compiler error, or even just trying to delete a file that doesn't exist), then the make process aborts
  - You can avoid that by prepending a `-` or `\` to the command
- Commands to be executed will normally be displayed as well
  - You can override that for a single command via `@`
  - You can turn it off entirely via `.SILENT:`
- We commonly use variables as macros for replaceable terms (like which compiler you're using, flags you might wish to add, etc.)
- For any line, dependency or whatever, if the line becomes too long, you can continue onto the next by ending the first line with a `\`.

Makefiles

Final thought

Honestly, there's no way we could cover everything for make files in one sitting. This is one of those *read the manual* things.
At the very least, I'd suggest either googling or checking out:
`http://www.gnu.org/software/make/manual/make.html`

And, of course, example time!

Questions?

Comments?

- Favourite pasta?