Build pipeline

Recall the build pipeline:

1. Preprocess
2. Compile
3. Link

Using g++ to preprocess and compile without linking

If you specify the -c flag to g++, it will preprocess and compile the specified .cpp files each into their own object file (extension is `.o`) containing their respective object code, but it will not link them into an executable program. Recall: object code is essentially machine code, nearly ready to be executed by your CPU.

Example: `g++ -c my_file.cpp`

This example will produce a file called `my_file.o`, which contains the binary object code compiled from `my_file.cpp`.

Once you do this for each of your .cpp files, you should have a bunch of object files. You can then link them into an executable program by listing them all with g++ (and no special flags).

Example: `g++ file1.o file2.o file3.o -o <name of executable>`

As always, the -o flag is optional—it just lets you specify the name that the executable should be given.

Why the heck would we want to do this? Well…

- Preprocessing & compilation are slow
- Linking is relatively fast

It shouldn't be surprising, then, that being able to separate these phases of the build pipeline can help us do things more efficiently.

For example, suppose:

- We have 1,000 .cpp files in our program.

- Compiling 1 .cpp file takes an average of 1 second. So compiling all 1,000 .cpp files takes 1,000 seconds, or about 17 minutes.
- Linking all of the object code into an executable program takes 20 seconds.

Suppose we compile the 1,000 .cpp files into 1,000 respective .o (object) files and build our executable. The whole process takes about 17 minutes and 20 seconds.

Suppose we then modify one of our 1,000 .cpp files, and then we want to rebuild our executable to reflect the changes. Consider:

- If we recompiled all 1,000 .cpp files and then relinked them all together (e.g., via `g++ file1.cpp file2.cpp … file1000.cpp`), it would take 17 minutes and 20 seconds again.
- If we only recompiled the one modified .cpp file into its respective object file, and then relinked it with all of the other existing 999 object files, it would only take 21 seconds (1 second to recompile the modified .cpp file, 20 seconds to relink).

Hence, it's much more efficient to preprocess and compile our .cpp files into object files separately.

Makefiles

Okay, so we've learned how to separate our code into arbitrarily many .h / .hpp and .cpp files. We've also learned how to make our build pipeline more efficient—compile the .cpp files separately into their own .o files, and then link them all together whenever any of the object files change (because their respective .cpp files changed).

However, building our code is now a several-step process. We have to run g++ with the -c flag for each of our .cpp files, and then run g++ on all of the generated object files to produce an executable. Furthermore, we have to carefully keep track of which .cpp files we've modified, so that we know which ones we have to recompile prior to relinking the program.

Question: is there a tool that will make the build process simpler? Answer: Yes. Enter **makefiles**.

Makefile overview

A makefile is a special kind of file, written in a special language that we'll colloquially refer to as "the Make language". All a makefile does is specify:

- Files that can be built / generated. These files are called targets
    - Example: my_file.o can be generated by compiling my_file.cpp
- Prerequisites of targets
    - Example: my_file.o will need to be rebuilt any time my_file.cpp changes. In addition, my_file.cpp might #include various header (.h) files. If any of those header files change, it's possible we'd have to rebuild my_file.o. Hence, we say my_file.o depends on my_file.cpp and all of the files it #includes (recursively—i.e., also the files that they include, and the files that they include, and so on…)
- The commands necessary to build targets from their respective prerequisites. These are called recipes. Each target has a recipe.

Once you have created a makefile to specify all of these things, you can run the `make` command from your terminal. Important: the makefile must be called either "makefile" or "Makefile", with no filename extension, and it must be present in your working directory when you execute the `make` command. For CS 162, just put your makefile in your project folder, which should also contain all of your .cpp and .h files.

Basic syntax of the Make language

Once you've created your makefile (which, again, must be called either "makefile" or "Makefile"), you can begin to define targets, prerequisites, and recipes.

The syntax for defining a target, its prerequisites, and its recipe, is as follows:

```
<target name>: <prerequisite 1> <prerequisite 2> …
    <recipe>
```

In this case, <target name> is the name of the file that can be generated (the target), the prerequisites are the names of the files needed to generate the target, and the <recipe> is just a series of terminal commands (separated by newlines if there is more than one command—but there's often just one) used to generate the target from the prerequisites.

Important: The Make language is very picky about whitespace! Specifically, the recipe must be tab-indented. If your IDE / source code editor is configured to replace all tabs with spaces, this

might cause problems. Luckily, lots of IDEs and source code editors are smart enough to know not to do that when editing a makefile.

The first target in your makefile is called the default target. When you execute the `make` command, all it will do is try to build the default target. However, if you set up your prerequisites right, then this is all you need.

All files on your computer have timestamps, which record when they were last modified / generated.

Whenever the Make utility attempts to build a target file, it first checks to see if the file actually needs to be built. To do this, it looks at timestamps—if the target file already exists, and its prerequisites are up-to-date (they themselves don't need to be rebuilt), and the target file's timestamp is not older than any of its prerequisites' timestamps, then the target file does not need to be built, and so the Make utility will proceed to do nothing (it may say something like "nothing to do! Everything is up-to-date"). The target file will only be rebuilt if a) it doesn't currently exist, or b) it exists, but it's older than the modifications to one or more of its prerequisites.

For instance, suppose you specify your default (first) target as your executable, and you list all of the appropriate object files as its prerequisites, like so:

```
my_executable: file1.o file2.o file3.o
     g++ file1.o file2.o file3.o -o my_executable
```

Now, suppose you have separate targets for each of the .o files, like so:

```
file1.o: file1.cpp file1.h
     g++ -c file1.cpp
file2.o: file2.cpp file2.h
     g++ -c file2.cpp
file3.o: file3.cpp file3.h
     g++ -c file3.cpp
```

Suppose we have all of our .cpp and .h files, but we have never built our object files or executable. Suppose we run `make`. This will tell the Make utility to try to generate `my_executable`. It will then do the following:

1. It will notice that my_executable does not exist. It will try to build it.
2. However, it will also notice that the prerequisites (object files) don't exist either. So before it tries to build the executable, it will build the object files. This order is enforced by the fact that the object files are prerequisites of the executable—Make knows that the executable cannot exist without the object files.
3. Each of the three object files will be built according to their respective recipes
4. It will then build the executable via its recipe

Suppose we then modify file1.cpp, but we leave everything else alone. Suppose we then rerun `make`. Again, the executable is our default target, so that's where the Make utility will start. It will then do the following:

1. It will notice that my_executable depends on the three object files. In order to decide whether or not my_executable needs to be rebuilt, it first needs to decide whether the object files need to be rebuilt.
2. It will examine the three object files and their respective prerequisites. It will notice that file1.o depends on file1.cpp. Based on timestamps, it will notice that file1.cpp was modified <u>after</u> file1.o was generated, which means file1.o is outdated!
3. It will rebuild file1.o via its recipe (`g++ -c file1.cpp`)
4. It will acknowledge that file2.o and file3.o do not need to be rebuilt, since their prerequisites have not been modified
5. It will then go back to check my_executable. Now, file1.o has been rebuilt / modified in step 3. Indeed, according to timestamps, file1.o is newer than my_executable, which means the executable is outdated!
6. It will rebuild my_executable via its recipe

Hence, the Make utility works simply by specifying files and their prerequisites. It starts by recursively rebuilding prerequisites if necessary, and then attempting to rebuild the original target.

You can think of the Make process on a target like so:

1. For each of the target's prerequisites that have their own targets / recipes, process them (where "process" refers to the steps outlined in this very list—yes, this is recursive)
2. If the target doesn't exist, <u>or</u> if the target does exist but is older (by timestamps) than any of its prerequisites, build the target according to its recipe.

<u>Simple makefile example using our baseball player struct / file separation demo:</u>

We have the same three files for this baseball player demo as we did in the file separation lecture: main.cpp, baseball_player.cpp, and baseball_player.h. We also have our makefile, which looks like this:

```
1 run_baseball_players: main.o baseball_player.o
2     g++ main.o baseball_player.o -o run_baseball_players
3
4 main.o: main.cpp
5     g++ -c main.cpp
6
7 baseball_player.o: baseball_player.cpp baseball_player.h
8     g++ -c baseball_player.cpp
```

Our default target is our executable, which I've called "run_baseball_players". In order to build the executable, we first need to build main.o and baseball_player.o, so I've listed them as the executable's prerequisites.

In order to build the .o (object) files, we need the respective .cpp files. Also, baseball_player.o depends directly on baseball_player.h, so I've listed it as a prerequisite as well—if we modify baseball_player.h, then the Make utility will know to rebuild baseball_player.o. There are more robust ways to recursively find all prerequisites of a C/C++ program, but that's beyond the scope of this course. Feel free to do independent research if you're interested.

Here's what our working directory looks like:

```
[alex@alex-desktop tmp]$ ls
baseball_player.cpp  baseball_player.h  main.cpp  makefile
```

Since our executable is our default target, all I need to do to build it is simply run `make` in the terminal:

```
[alex@alex-desktop tmp]$ make
g++ -c main.cpp
g++ -c baseball_player.cpp
g++ main.o baseball_player.o -o run_baseball_players
```

Notice that, when I do this, it proceeds to build main.o, then baseball_player.o, then the executable. (the order in which the object files are built does not matter, but it's important that the object files are both built before the executable, which is enforced in the makefile since the object files are prerequisites of the executable).

Here's what our working directory looks like now—notice that the object files have been generated, as has the executable:

```
[alex@alex-desktop tmp]$ ls
baseball_player.cpp  baseball_player.h  baseball_player.o  main.cpp  main.o  makefile  run_baseball_players
```

Suppose I then modify main.cpp. In this case, I just opened it in vim and immediately saved & quit. This updates the timestamp of the file. However, main.o has not yet been updated / rebuilt, so it still reflects the old object code of main.cpp. When I run make, then, it will recognize:

1. main.o is a prerequisite of the executable
2. main.cpp is a prerequisite of main.o
3. main.cpp has been modified more recently than main.o, so main.o is outdated
4. It will then rebuild main.o
5. And then it will rebuild (relink) the executable

See here:

```
[alex@alex-desktop tmp]$ make
g++ -c main.cpp
g++ main.o baseball_player.o -o run_baseball_players
```

Notice: It does not rebuild baseball_player.o because it doesn't have to! baseball_player.cpp has not been modified, so baseball_player.o is still up-to-date.

Constants in makefiles

You can define constants / variables in makefiles. Convention is that they're all caps, snake case (like preprocessor constants), unless they're used for "internal" makefile purposes (we won't cover those, but convention for them is lowercase snake case). Usually, you'd define constants at the very top of your makefile. The syntax is simply:

`SOME_CONSTANT_NAME = <some string value>`

Once you've defined a constant, you can use it in your targets, prerequisites, and / or recipes like so:

$(SOME_CONSTANT_NAME)

Here are some common constants in makefiles that you may find useful:

1. The compiler command to use for all compilations (e.g., `COMPILER = g++`).
2. Flags to provide to the compiler for all compilations, like warning flags, or -g to insert debugging metadata into your object files to assist valgrind / gdb (e.g., `CFLAGS = -g -Wall`)
    a. Note: These are usually different from the linker flags, which are supplied to the recipe that links the executable. We probably won't use any linker flags for this course, but compiler flags might come in handy
3. The name of the executable (e.g., `EXE = assignment2`)

Here's our updated demo with some constants:

```
COMPILER = g++

# -Wall means "warn all"---it enables all compiler warnings.
# -g inserts debugging metadata into the EXE, which helps valgrind and gdb
#       display / track line numbers of errors, etc
CFLAGS = -g -Wall
EXE = run_baseball_players

$(EXE): main.o baseball_player.o
    $(COMPILER) main.o baseball_player.o -o $(EXE)

main.o: main.cpp
    $(COMPILER) $(CFLAGS) -c main.cpp

baseball_player.o: baseball_player.cpp baseball_player.h
    $(COMPILER) $(CFLAGS) -c baseball_player.cpp
```

Advanced makefile features (not required for CS 162, but potentially helpful if you're interested)

There are lots of advanced features of makefiles. If you use these appropriately features, you could end up with a very robust makefile that can work for many different C++ projects right out of the box!

Auto dependency generation

Suppose a.cpp includes a.h, which includes b.h, which includes c.h. In some sense, a.cpp depends on c.h (it transitively includes it, after all). Ideally, whenever we modify c.h, it would be nice if our makefile was smart enough to rebuild a.o.

One way we could do this is by specifying the a.o target like so:

```
a.o: a.cpp a.h b.h c.h
	g++ -c a.cpp
```

Now that c.h is a prerequisite of a.o, a.o will be rebuilt whenever c.h is modified (or whenever b.h, a.h, or a.cpp are modified). However, this can get really messy. You may have a file that includes 10 other files, each of which includes 10 other files, and so on…

Luckily, g++ and the Make utility together provide ways of automatically and recursively discovering all of the prerequisites of a .cpp file (all of the headers it includes, all of the headers those headers include, and so on…). Here's a tutorial.

Auto .cpp file discovery—one makefile target to rule them all

It can get cumbersome having to remember to write targets / recipes for all of your object files. Luckily:

1. There are terminal commands, like `find`, that can automatically discover .cpp files in a given directory (even recursively)
2. You can run terminal commands from within a makefile to discover those .cpp files
3. You can use makefile pattern substitution to convert those .cpp file names to respective .o file names, which gives you a list of all .o targets. Here's the relevant documentation (see patsubst).
4. You can create a "generic" .o makefile recipe that works for every .o target by simply compiling its respective .cpp file. If the list of all .cpp files are discovered by a shell command in steps 1 & 2, then you can have a relatively short makefile that can build extremely large projects. To do this, you need to use Make pattern rules. Here's the relevant documentation.

Organizing your files

A makefile target does not technically have to be a file name, but rather a file path. Usually, it's a relative path.

A common strategy is to:

1. Store your .cpp files in a src/ directory
2. Store your .h / .hpp files in an inc/ directory. Important: To tell g++ where the header files are located so that #include directives work properly, you have to specify a -I argument. See the g++ man pages.
3. Store all of your .o files in an obj/ directory, or a .obj/ directory (files / folders prefixed with a period are "hidden" on *nix systems (Mac / Linux))
4. Store the executable in a bin/ directory

Order-only prerequisites

You can specify order-only prerequisites of a target. This is a special kind of prerequisite. The Make utility will ensure that the order-only prerequisites exist prior to building the target, but it will not care about their timestamps (an order-only prerequisite being newer than the target will not trigger Make to rebuild the target). A common use case is to specify folders as order-only prerequisites. For instance, if the target .o file is supposed to go in the obj/ folder, you better make sure the obj/ folder exists first. However, a folder's timestamp is updated any time its contents are updated, and you don't want that to affect when other files are rebuilt (e.g., updating obj/a.o, which updates the timestamp of the obj/ folder, should not trigger Make the rebuild all other object files). Making the folder an order-only prerequisite solves this problem.

(and yes, you can define targets / recipes for folders. The recipe is usually just `mkdir <folder>`)

See here.

And much more!

Check out the GNU Make documentation if you're interested. There are lots of cool features.