Lecture 13: Make Files

Shaving Yaks

Before we start talking coding, let us begin with a story.

Once upon a time, I wanted to wash my car. Sadly, my hose was broken. Fortunately, that was not a problem, for hoses are cheaply sold at Home Depot. Sadly, the nearest Home Depot is across a toll bridge where one must pay a fee to cross. Fortunately, my neighbour has a toll-pass which gives free transit across the bridge. Sadly, if I were to ask my neighbour for the toll-pass, he will request that I return a cushion of his that I borrowed. I have the cushion. Unfortunately, my son broke the cushion and so to return the cushion I would need to fix the cushion. Furthermore, the cushion was stuffed with Yak hair. That was when I found myself down at the local zoo, shaving a yak.

Ha ha ha. You may also be familiar with the song “there’s a hole in my bucket” conducted between Henry and Liza. In this song, Henry’s woes are sung:

- A hole in my bucket,
- The stick is too long,
- The axe is too blunt,
- The stone is too dry.

As well as Liza’s replies:

- Then fix it,
- Then cut it,
- Then sharpen it,
- Then wet it.

Both the yak story and the ‘hole in my bucket’ song play on the idea of dependencies. In this case, dependencies of one bit of code upon another.

The former is more a story about whether you needed the dependencies in the first place. Perhaps it would be better just to pay the toll? The second, is a story about cyclic dependencies, which we will cover below.

Make

What is make and what is it for? Well by now you should roughly know this as make is a necessary tool for writing in C and C++ at Adelaide Uni. Make handles dependencies and compilation.

So, what are dependencies? Whenever one bit of code imports/includes/uses another a bit of code, we call that a dependency. Most programs written by students will ‘depend’ on stdio, iostream, string and other in-built C/C++ libraries. As these libraries are precompiled, the fact they are dependencies is largely irrelevant. However, in addition to standard libraries, once code reaches a certain size a non-terrible programmer will start breaking it up into separate files. This is very sensible from the point of usability. It is the same logic as having folders on a hard drive for different things, or even different ‘sets of photos’ etc. Keeping things which are ‘tightly logically bound’ separate from very different ‘tightly logically bound’ things makes sense. Code which fiddles with files probably should be kept separate from code which does maths.

So now that we have argued for separate files, how do we tell the compiler how to deal with said files? Well we could do this (if we were terrible):

```
gcc file1.c file2.c file3.c file4.c file5.c file6.c file 7.c file8.c
file9.c file10.c file11.c file12.c file13.c file15.c file16.c fi1e17.c
file18.c file19.c file20.c -o My_Exe
```

Why is this bad? Well firstly, we forgot file14 entirely and file17 has a one (1) instead of an 'L' in it. We could fix this type of problem by having a bash-script with this line hard-coded it in. Alternatively, we could be lazy and do this:

```
gcc *.c -o My_Exe
```

This has some obvious drawbacks. What if we have multiple programs accessing the same '.c' files? How could we store it so that the compiler does not pick up the wrong files but also does not have duplications of dependencies. Short answer, not possible.

However, these are fairly minor concerns. The larger concern is that programs with many dependencies take time to compile. Most programs you will write in university will be small and will not suffer from this, but even a program with twenty files will begin to take 'real time' to compile. Large programs (those constituting millions of lines of code) can take hours (or even days) to compile from scratch. The problem with the above approaches is that not all of these files need to be recompiled. A small change in one function may require no recompilation of any of the other files (depending on the dependencies). For example things written in 'main' usually don't require anything else to be recompiled. So, what if we wanted to only recompile things that need to be recompiled and skip everything else? How can we do this?

The answer to this question, was Make. Now, unfortunately Make suffers from the same problem that Bash suffers from. There is a certain level of hack-built-upon-hack in the bones of Make. It was written by an intern student and got passed around enough that it somehow became a standard tool. It has since been superseded by better compilation tools which you are more likely to use in industry. However, understanding how Make works will give you some indication as to how these other tools work and what they do. So… Make.

What does Make Do?

Make is all about date-stamps. A date-stamp tells you when a file was last modified and (usually) an unmodified file does not need to be recompiled. Actually, this isn't strictly true. There are two reasons to recompile a file. Number one is that it changed (hence date stamps). Number two is that something it relies upon (a dependency) changed. So, the game of Make is to keep track of what files have changed (which is an easy operation using the operating system) and which files depend upon which files… a less easy operation.

Some Basics

To use Make, you must have a file called explicitly 'Makefile' (no extension) in the directory you wish to compile things for. You call 'make' within that directory. Now, there is a way to make 'Make' (hur hur hur) less 'specific'. You can call Make manually and give it a file as an argument (we'll do this later). Suffice to say, if you have a Makefile and are in the correct directory, calling 'make' should do something.

A Minimalist Makefile

Looks like this:

```
my_exe: my_code.c my_code.h
	gcc my_code.c -o my_exe
```

This reads:

- Make a file called my_exe from my_code.c (the gcc command)
- Do this if someone types: make my_exe
- Only do something if my_exe is older than my_code.c or my_code.h. Otherwise… do nothing

The Parts

```
my_exe: my_code.c my_code.h
	gcc my_code.c -o my_exe
```

There are three parts to a Make instruction. The first (red) is the target, the second (blue) are the dependencies and the third (green) is the compilation instructions. Also, a warning: White Space matters.

The green part is just code. It is not limited to compilation instructions. You can put whatever Bash-related nonsense here that you want. A common example is the following:

```
clean:
	rm *.o my_exe
```

Many people have this as an optional make instruction to get rid of all the existing compilation (including .o object files which we will discuss below). However, you see this does nothing resembling compilation. The green section is very flexible.

The blue part (dependencies) is just that. It is a list of files, which if possessing older date-stamps than the target (the red part), will gatekeep whether the green code is executed or not. If all blue files are old, then nothing happens, if they are new, then green code happens.

The red part serves two purposes. Firstly, it allows Makefiles to have options. Consider below:

```
clean:
	rm *.o my_exe

my_exe: my_code.c my_code.h
	gcc my_code.c -o my_exe
```

This can be called as either ‘make’, ‘make clean’ or ‘make my_exe’. By default, ‘make’ (without arguments) does the first thing so as written currently will do ‘make_clean’. If the order were reversed it would do the ‘my_exe’. Otherwise if an argument is given it will look for something matching the argument and do that instruction (i.e. either clean or my_exe).

Default Make

Given make automatically chooses the first target if not given an argument, it makes sense to use this power sensibly. A good example would be a ‘how-to’.

```
nothing:
      echo "make clean: remove all files"
      echo "make bucket: compile bucket"
      echo "make build: compile all"
      echo "make deploy: move files around"
```

Or something to this effect.

Layered Dependencies

So far, nothing we have done is superior to a Lazy.sh script really. The only real difference is that if My_Exe is already compiled and we have not changed the code we do not recompile it. To make 'Make' (hur hur hur) have a real benefit we need to consider a layered example.

```
clean:
      rm *.o my_exe
my_exe: driver.o quack.o
      gcc driver.o quack.o -o my_exe
driver.o: driver.cpp
      gcc -c driver.cpp
quack.o: quack.cpp quack.h
      gcc -c quack.cpp
```

This Makefile assumes the existence of three C files, driver.cpp, quack.cpp and quack.h. To build it, we would call "make my_exe". However, if we look at my_exe's dependencies we find driver.o and quack.o. Moreover if we look at the compilation instructions, we see driver.o and quack.o again. At this point, Make will double check to see if these are files. If they are not, it will then check to see if it has another target with the specified names, which it does. Prior to compiling 'my_exe', Make will attempt to build both driver.o and quack.o.

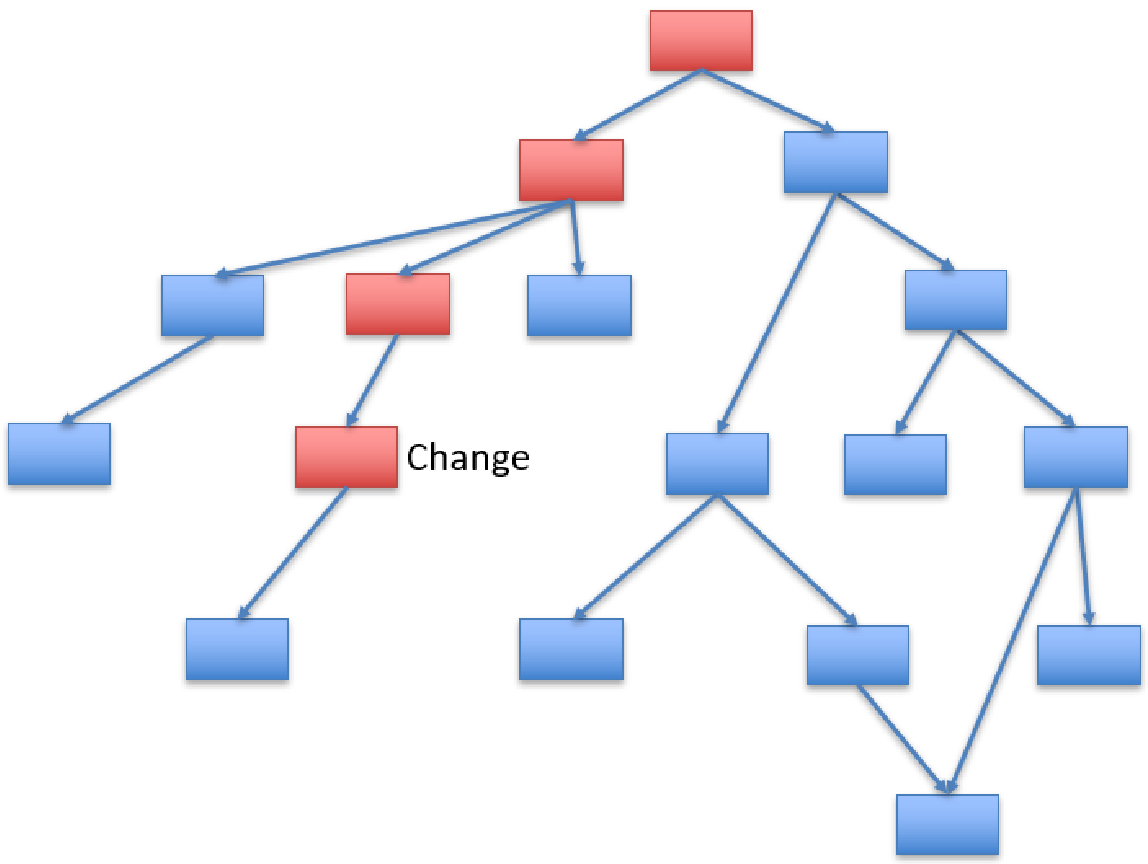
Let us assume that none of the required files exist (i.e. nothing has been compiled). Thus from scratch we would use the:

```
      gcc -c quack.cpp
```

Compile instruction. The '-c' creates a kind of interim compilation file. The file is compiled, but is not quite an exe and is certainly nothing like the original code. These object files need to be linked together prior to having a real exe. The generated file will be the old file + '.o' so this will produce a 'quack.o' file. Likewise driver.o will also be created and thus my_exe can be produced.

Now, again, this is broadly pointless… until we decide to change something. Let us change… quack.h. Now we run "make my_exe". First we need to check our dependencies. The first, driver.o is fine. It exists. We have a target for driver.o and so we need to check its dependencies, which have not changed, so we do nothing. Next we look at quack.o. Oh dear, quack.o is older than quack.h. So we need to recompile quack.o to proceed. We recompile quack.o and recursively go back to my_exe. Now that quack.o is new, it means that my_exe is older than its dependencies and we need to recompile my_exe. Then we are done. However, we have saved the arduous step of recompiling driver.o. Hooray! Job done. Make has finally done something very useful (save me time).

Below is a visual example of a dependency tree. The file at the top is like the main.c or driver.c file. Each file is dependent on the files connected below it. The main advantage of Make is nicely illustrated because changing the file marked ‘Change’ will only ever result in recompilation of the red boxes but leave the blue boxes intact. One thing you can see however is that there is an example in the bottom right where two parts of the tree are dependent on one file. In this case, both arms would be recompiled. This is why changing versions of C can be a headache, because standard dependencies might change requiring recompilation of everything dependent on them. Consider what would happen if someone changed stdio for example?



Custom Makefile

As mentioned before, the file need not be called Makefile. To do so, you can use the ‘-f’ flag.

```
> make -f <my_weird_make_file> <parameters>
```

i.e.

```
> make -f QuackMake my_exe
```

Make Shortcuts

So the next bit is where we see what happens when you give an intern responsibility for a programming task…

$@ - Target Shortcut

If you are feeling lazy, you can auto-replace the 'target' of a Make instruction using the $@… thing.

```
my_target: my_dependency
	something something $@
```

Make will automatically identify $@ as a Make shortcut and replace it with 'my_target' (i.e. the target).

$^ - Prerequisite Shortcut

```
my_target: my_dependency1 my_dependency2
	something something $^
```

Make will automatically identify $^ as a Make shortcut and replace it with 'my_dependency1 my_dependency2' (i.e. the target). This can be handy… for some compilation instructions… sometimes. Such as:

```
my_exe: file1.c file2.c file3.c
	gcc $^ -o $@
```

Before you get too excited, remember that '.h' files are a thing and this does not play readily with '.h' files.

$< - First Prerequisite

As $^ but only the first dependency.

```
my_exe: file1.c file2.c file3.c
	gcc $^ -o $@
```

'*' Wild Card

```
my_target: my_*.c
	gcc $^ -o $@
```

So, this will include all '.c' files as dependencies (at least all which start with 'my_'). It is similar to * in Bash. Again… this is Lazy, but also getting into dangerous territory. What if there is an extra '.c' file in the directory, what will happen then?

Multiple Targets

It is also possible to have multiple targets with the same rule:

```
target1 target2:
	# Do stuff
```

The rule will run for each target in the target list. This means it makes no sense to have both compiled in a single go because they will be compiled twice. Instead by judiciously using $@ you could potentially make both using the same command (seems unlikely to me really).

% Wildcard

Now we are getting into the weeds. So the % wildcard can be potentially useful if you have a series of similarly named files which have similar compilation instructions. Consider:

```
my_target_%.o: my_%.c
      gcc $^ -o $@
```

This will check to see any files matching the pattern 'my_target_??????.o' then look for the corresponding 'my_?????.c' and then use that to do the compilation. A super lazy Makefile might look like:

```
%.o: %.c
      gcc $^ -o $@
```

Which would sort of compile everything available? If there is a particular file we don't want to fall prey to this simplified compilation instruction we can specify a literal one too.

```
%.o: %.c
      gcc $^ -o $@
driver.o: driver.c other_thing.c other_thing2.c
      gcc $^ -o $@
```

Now the %.o target will only run if it cannot find an alternative target (which it will find) and thus not run, instead running the specific target (driver.o).

Macros

So now that we have this symbol heavy gibberish Makefile (it is not that bad really, but we are going down a dangerous path), what if we want to make it even more opaque? What if suddenly we decide we want to change from **gcc** to **clang**!

We can use macros (which are basically like shell variables).

```
CC=gcc
my_exe: main.c file1.o, file2.o
      $(CC) $^ -o $@
file%.o: file%.c file%.h
      $(CC) -c $<
```

At runtime (Make runtime that is), all the $(CC) will be replaced with 'gcc'. Hooray! (BTW you can also use ${} instead of $() which is not confusing at all).

Parting Words

Well… you can probably tell why 'automated Makefiles' is a thing. Use many environments and they hide all of this from you, with good reason. There is so much that could go wrong, and a program can somewhat deduce dependencies without the user as long as it knows which files to 'include'. However, knowing what it is doing under the hood is also very useful, especially when it is not doing what you expect.