COMP 161 - Lecture Notes - 15 - make and Makefiles

Spring 2014

In these notes we stop a moment to look at the program *make* and how it can be used to automate parts of the program build process. These notes are all adapted from the GNU *make* manual; you should read the introductory material there[1]

[1] http://www.gnu.org/software/make/manual/make.html

Why make?

Let's consider a basic program using our standard structure: *main_prog.cpp* contains the main procedure and utilizes three libraries *lib1*, *lib2*, and *lib3*. Each library has it implementation *cpp* file, its header *h* file, and another *cpp* file containing gTest tests. We might imagine the following build tasks:

- Compile and link the libraries with the main file
- Compile and link the libraries with the tests and the gTest main
- Compile and link one library with gTest
- Compile any of the *cpp* files to an object

The first few options require compiling many files when changes may only have been made to a one or two of those files. If you forget to compile one and use an old object, then you might see bugs where there are none. Wouldn't it be nice if the system just knew to recompile any library that has been changed? Well, *make* can do this. Using *make* also saves you from having to type long compile commands or from search back though your command history to find the right command.

In practice, programs can require significantly more source files than we're used to[2]. All of the compiling tasks that are annoyances with our small projects are nearly intractable problems with large software projects. Keeping track of dependencies between files in large programs is really a task best left to a program. This is what *make* and programs like *make* can do for us. The *make* program can save time and frustration. It's not hard to use and there is a wealth of information on the web. In these notes we'll look some basics and see how to use *make* for all those build tasks we talked about.

[2] Go explore the source for the game DOOM 3 https://github.com/id-Software/DOOM-3

Basic make

When all you want to do is compile a single file that does not need to be linked to any other file, then make can automatically determine a

g++ command for you. Let's say you want to compile an object file from a *lib1.cpp*. Then all you need to do is pass *make* the name of the file it should make, in this case *lib1.o*.

```
make lib1.o
```

When you run this command make comes up with the following command:

```
g++    -c -o lib1.o lib1.cpp
```

You should recognize this as the command for building *lib1.o* from *lib1.cpp*[3].

[3] notice the arguments are given in a different order than we're used to

Now what if you have a simple program that was entirely contained in a single source file *simple.cpp* and you wanted to build the executable *simple*? You could use the following make command:

```
make simple
```

This results in the follow compilation command:

```
g++     lib1.cpp   -o lib1
```

This again, is a fairly standard *g++* command for building executables.

What we've seen is that without any significant intervention on our part, *make* can manage some very basic compile time tasks. However, the commands it uses are perhaps too simple. What if we want the warnings given by *-Wall*? What if we need to link multiple files together or link to a library like *gTest*? What if we want the name of the made file to differ from that of the dependencies? For this we need to exert some control of what *make* does, and for that we must provide make with a script telling it what kinds of commands to use. This script is called a *Makefile*

Makefiles

To control the behavior of *make* you must create a file named *Makefile* and place it in your current working directory. If such a file exists, then when *make* is run it will look to that file for rules to use in making certain files. Rules have a very simple form:

```
target : dependencies
     rules
     ...
```

The *target* is the file to be made. The *dependencies* are the files needed to make the *target*. Finally, the *rules* are the commands used to turn

the *dependencies* in to the *target*. It is vital to note that the spacing before each rule **must be a tab**. You can't use multiple spaces or anything other than a single instance of the tab character.

Let's start with some rules for building some library objects for our pretend project

```
lib1.o : lib1.cpp
     g++ lib1.cpp -c -Wall

lib2.o : lib2.cpp
     g++ lib2.cpp -c -Wall

lib3.o : lib3.cpp
     g++ lib3.cpp -c -Wall
```

Now if you type *make lib1.o*, then make will find your *lib1.o* rule and use that rule. We've included the *-Wall* flag in our rule, so now we get warning reports as well as errors.

So far so good. Let's build some rules for making our test program. In this case, we want to build all our test objects, then link them together into a single text executable. We can then use the gTest filters to run individual tests if we like.

```
libTests : lib1_tests.o lib2_tests.o lib3_tests.o lib1.o lib2.o lib3.o
    g++ lib1_tests.o lib2_tests.o lib3_tests.o lib1.o lib2.o lib3.o \
-lgtest -lgtest_main -lpthread -o libTests

lib1_tests.o : lib1_tests.cpp
    g++ lib1_tests.cpp -c -Wall

lib2_tests.o : lib2_tests.cpp
    g++ lib2_tests.cpp -c -Wall

lib3_tests.o : lib3_tests.cpp
    g++ lib3_tests.cpp -c -Wall
```

Here we used a backslash in a recipe line to break up the line so that it fits on a page when printed. You shouldn't run in to many problems with this, but splitting lines like this doesn't always work as intended[4].

If we combine the above rules with our previous rules for the library implementation objects, then we have everything we need to

[4] http://www.gnu.org/software/make/manual/make.html#Splitting-Recipe-Lines

build our text executable. Let's say that you've compiled nothing. Then the command *make libTests* will first note that all the dependencies are missing and invoke the rule for each dependency automatically. Thus, in one simple command, we'd invoke seven compile commands. Now, what if you changed something in *lib2.cpp*? You need to recompile *lib2.o* and *lib2_tests.o*. You could run the make rules for each object, but re-running *libTests* will cause *make* to notice that *lib2.cpp* has changed. This in turn will cause *make* to rebuild those dependencies. This is a killer feature of *make*. It will, more often than not, keep your dependencies up to date.

Let's finish things out with the rules needed to build our main executable.

```
myprog : main_prog.o lib1.o lib2.o lib3.o
    g++ main_prog.o lib1.o lib2.o lib3.o -o myprog

main_prog.o : main_prog.cpp
    g++ main_prog.cpp -c -Wall
```

With the addition of these rules we have everything we need to use *make* in order to build all the various elements of our program. What's more, if we place the *myprog* rule at the top of the Makefile, then the command *make* will automatically run that rule. In general, *make* runs the first rule in the Makefile. If you wanted the test executable to be the default rule, then you'd place that first in the file and list all the other rules underneath it.

Phony Rules

Thus far all of our Makefile rules have built specific files by name. Sometimes we want logical rules, rules where the target isn't the name of a file but a description of a build or build related scenario. Put another way, sometimes we like to name rules for what the do, not what they produce. Such rules are called *phony* rules.

The classic phony rule is *clean*. The *make clean* rule is included in most standard Makefiles and is used to get rid of everything but the source code files. In our case this means deleting object files, emacs temporary files, and executables. We'll use a series of phony rules for this, a rule for each file type, and then have *make clean* call all the individual clean rules.

```
.PHONY : clean cleanobj cleanexe cleantemp

clean : cleanobj cleanexe cleantemp
```

```
cleanobj :
    rm *.o

cleanexe :
    rm myprog libTests

cleantemp :
    rm *~
```

First, notice that we By listing the sub-rules as dependencies for *clean*, we ensure that *make clean* invokes each of these rules. The sub rules all have no dependencies and run a simple *rm* command to get rid of some files. We could have written a single clean rule like this:

```
.PHONY : clean

clean :
     rm *.o *~ myprog libtests
```

Writing sub-rules gives you the option to clear out specific groups of files if you want. It also illustrates that we can chain phony rules together the same way that concrete rules get chained together.

Complete Makefile

```
myprog : main_prog.o lib1.o lib2.o lib3.o
    g++ main_prog.o lib1.o lib2.o lib3.o -o myprog

main_prog.o : main_prog.cpp
    g++ main_prog.cpp -c -Wall

libTests : lib1_tests.o lib2_tests.o lib3_tests.o lib1.o lib2.o lib3.o
    g++ lib1_tests.o lib2_tests.o lib3_tests.o lib1.o lib2.o lib3.o \
-lgtest -lgtest_main -lpthread -o libTests

lib1_tests.o : lib1_tests.cpp
    g++ lib1_tests.cpp -c -Wall

lib2_tests.o : lib2_tests.cpp
    g++ lib2_tests.cpp -c -Wall

lib3_tests.o : lib3_tests.cpp
    g++ lib3_tests.cpp -c -Wall

lib1.o : lib1.cpp
     g++ lib1.cpp -c -Wall

lib2.o : lib2.cpp
     g++ lib2.cpp -c -Wall

lib3.o : lib3.cpp
     g++ lib3.cpp -c -Wall

.PHONY : clean cleanobj cleanexe cleantemp

clean : cleanobj cleanexe cleantemp

cleanobj :
    rm *.o

cleanexe :
    rm myprog libTests

cleantemp :
    rm *~
```