Makefile Tutorial

At the end of the gcc tutorial I discussed the usage of the -c flag to compile source files separately, then using a second call to gcc to combine the output of the -c operations into a single executable. This concept is used most often in the construction of makefiles.

What is a Makefile?

A makefile is a specially formatted text file that a unix program called 'make' can interpret. Basically, the makefile contains a list of requirements for a program to be 'up to date.' The make program looks at these requirements, checks the timestamps on all the source-files listed in the makefile, and re-compiles any files which have an out-of-date timestamp.

Variables and Comments

Variables and comments are two features of a makefile that are not required but make life much easier. If you are reading this tutorial with the intention of compiling your own source code, you should already understand the usefulness of variables, and if you have ever tried to re-use code you wrote a year ago, you understand the usefulness of comments. Variables in makefiles do not have types, and are typically interpreted as strings. Here are some commonly used variable names, with declarations:

```
CFLAGS = -g -Wall

CC = g++
```

These variables allow you to quickly change the behavior of your makefile. These variables will be used in later examples. To use the value stored in the variable, type

```
${varname}
```

and the text you assigned to that variable will be substituted into that place. Comments are nothing more than appending a '#' in front of a line, such as

```
#makefile for creating helloworld.exe
```

A Note About File Extensions

You will see that all the files in this tutorial have the extension .cc. '.cc' is the accepted file extension for c++ when using gcc. You may remember from the gcc tutorial that if your file extension is .cc, then you can call gcc on your files instead of g++ because you are using a recognized extension. Feel free to use .cp or .cpp but be sure that you are only making calls to g++ in that case, or you will get command line errors. Please also note that file names in the following makefiles reflect the fact that the example files end in .cc. If your files end in .cpp, be sure that files mentioned in your makefile match the filename exactly.

Dependencies

If you have ever read an online tutorial for writing makefiles, you will find that many of them come up short when explaining dependencies. Think of a dependency as a rule that must be adhered to. Here is the structure:

```
[name of rule] : [list of other rules, separated by
spaces] [list of source files, separated by spaces]

[TAB]command to execute in the event that the rule is
violated.
```

There are a few things to note here. First, the name of the rule on the left is just a name. In many makefiles, the rule name may be an exact match of a file name in your directory, but make does not actually look at that file. Take this example:

```
checkFormainDoto: main.cc

     g++ -c main.cc

checkFormainDotEXE: checkFormainDoto

     g++ main.o -o main.exe
```

What this says is that to have an up-to-date main.exe file, we need to check for main.o first. If make finds that main.cc has a different time stamp than the last time make was run, then it will complete the tabbed-in command on the line following the main.o rule. Then, when that rule is satisfied, make should find that the outcome of the checkFormainDoto rule has changed since the last time make was run and then run the tabbed-in code for creating make.exe. The format you will see in a typical makefile (and the format you should use and get used to reading) is:

```
main.o: main.cc

     g++ -c main.cc

main.exe: main.o

     g++ main.o -o main.exe
```

Or, for a more interesting usage:

```
main.o: main.cc

     g++ -c main.cc

help.o: help.cc

     g++ -c help.cc

main.exe: main.o help.o

     g++ main.o help.o -o main.exe
```

The reason these rules are called dependencies is that one rule, such as the main.exe rule, can *depend* on the status of another rule or file. Dependencies can become very complicated very quickly. More advanced topics outside of this tutorial allow a programmer to create makefiles for massive projects with little effort.

Another thing to notice in the format of a dependency is the requirement of a tab before the resulting command. A tab MUST BE USED. Be careful to realize that a sequence of spaces, or a space and then a tab are not the same thing as a single tab. After listing the requirements for a dependency, every line that follows which has a leading [tab] character will be interpreted as a command that will contribute to satisfying that rule. Note that a dependency can have multiple tabbed-in commands to satisfy it.

Getting Started

Let's start with a simple hello world example. Right click on the previous link, choose save target as, and save it to your working directory. Next, you will want to open up a text editor. I prefer emacs, keyboard-driven editor, or vi, another keyboard-driven editor. I find these programs very useful when I am working with the command line so that everything happens from a single window. If you are not comfortable with either of these programs, or you are on a windows workstation (emacs is available for windows), you can use any mouse-and-keyboard-driven editor, such as Wordpad on windows, or kate on linux machines. Jedit is another cross-platform text editor that you can download from the web. Use the editor to create a new file in the same directory as helloworld.cc. Name it makefile, and enter the following text:

```
helloworld: helloworld.cc

    g++ -o helloworld helloworld.cc
```

If you copied-and-pasted, and you have unexplained errors, delete the whitespace in front of 'g++ helloworld' and insert a tab. Be sure there is no white space in front of the first line. Save the file, go to your directory on the command line, and type:

```
> make
```

You should see an automatic call to g++ to compile your program. Try running make again. You should see something like

```
> make

make: 'helloworld' is up to date.
```

This is because every dependency in the makefile is satisfied, so no work needs to be done. Now, open up helloworld and change the message printed to the screen. When you run make again, you will see that it runs g++ again because it detected that helloworld.cc had changed since the last time make ran, and the helloworld dependency needed to be satisfied for the program to be up to date.

A More Interesting Example

For this example, we will use frog.h, frog.cc, and main.cc. Note that if you are using the same folder as you used for helloworld, you will need to save helloworld's makefile to a different location or overwrite it, since make looks for a file called makefile when it runs. It is possible to use the -f [filename] option to specify a different makefile name, but this happens so rarely, I recommend you don't get into the habit of doing it. This is a complete example, using variables and comments.

```
    #makefile for compiling the frog project

    #Author:  Adam Anthony



    CC=g++

    CFLAGS=-g -Wall

    RM=/bin/rm -f

    #for Windows, use RM=del



    #calling 'make all' will ensure that every feature of
the program is compiled
```

```
all: main

#create the object file for frog.cc

frog.o: frog.h frog.cc

     ${CC} ${CFLAGS} -c frog.cc


#create the object file for the main file

main.o: frog.h main.cc

     ${CC} ${CFLAGS} -c main.cc

#create the executable

main: main.o frog.o

     ${CC} ${CFLAGS} -o main main.o frog.o

# rule for cleaning files generated during
compilations.  Call 'make clean' to

#use it

clean:

     ${RM} *.o main
```

Again, play around with changing the files, and see what happens. Also, try

```
> make clean
```

to force make to delete all your compiled code, so that make will compile every file in your project on the next run.

An Even More Interesting Example

This example shows how you can direct make to compile different groups of files. Typically Make will find the first dependency in the file, satisfy all dependencies and sub-dependencies that it requires, and exit. It is possible to set up a compilation directory where all of your source files can be kept, controlled by a single make file. Create a directory 'Csources'. Copy helloworld.cc, frog.cc, main.cc and frog.h into Csources. Move the makefile from the last section into Csources. Modify it to look like this file (changes are in green):

```
    #makefile for compiling the frog and helloworld
projects

    #you can compile one at a time or use 'make all' to
compile both

    #Author:  Adam Anthony



    CC=g++

    CFLAGS=-g -Wall

    RM=/bin/rm -f

    #for Windows, use RM=del



    #calling 'make all' will ensure that every feature of
the program is compiled
```

```
all: main helloworld

#create the object file for frog.cc
frog.o: frog.h frog.cc
	${CC} ${CFLAGS} -c frog.cc

#create the object file for the main file
main.o: frog.h main.cc
	${CC} ${CFLAGS} -c main.cc

#create the executable
main: main.o frog.o
	${CC} ${CFLAGS} -o main main.o frog.o

#create helloworld
helloworld: helloworld.cc
	${CC} ${CFLAGS} -o helloworld helloworld.cc

	# rule for cleaning files generated during
compilations.  Call 'make clean' to
	#use it
```

```
clean:

     ${RM} *.o main
```

Try any combination of the following calls, to see how make behaves. To be sure you are seeing the proper behavior, run 'make clean' and then try a new combination.

```
> make all



> make main



> make helloworld
```

What you will find is that make all will compile everything, and the other two calls will only build the referred executable. If, however, one executable is already built, make all will only build any unbuilt targets. To test yourself, see if you can re-organize this final makefile so that calling 'make' will compile just frog.cc and main.cc, and calling 'make all' will still compile both projects.

Some Things to Think About

Now that you have a good idea about the inner workings of make, see if you can find the answer to the following questions on your own:

- Must I keep all of my files in the same directory for make to be able to find them?

The answer is no. You may want to experiment with keeping different aspects of your project-GUI, Math Libraries, Main files, etc - in different directories. Perhaps answers to the following questions will help you learn how to handle this new setup.

- What kinds of commands can I use in a makefile to satisfy dependencies?

  Any command you can type on the command line(cd, ls, grep, tar, javac), you can use for the tabbed-in portion of a dependency.

- Can make call itself?
  From the answer above, the answer here is Yes.

- Can make only be used with gcc/g++?
  Again from the answer to the second question, no. Make is not limited to gcc/g++. One advantage to using the CC and CFLAGS variables is that you can quickly change what compiler you are using and what flags to include, without editing the entire file. However, LaTeX users can use makefiles, as well as java programmers. Make can also be used to keep archives up-to-date without having to re-compress every single file in the archive. Since all make does is look at file ages and compare with its own records, it can be used for a multitude of applications.

- I have a project with hundreds of dependencies. Must I create a makefile by hand?
  No. Dependency generators are outside the scope of this tutorial. A good place to start looking is here.

Once you understand the answers to the above questions, you should feel comfortable working with and creating your own custom makefiles.