MAKEFILES

INTRODUCTION TO MAKEFILES:

A makefile is a tool to help you organize your program to allow for easier rebuilds, which is especially useful when you have multiple files. A makefile is simply a text file that is placed in the current working directory (where the source code needing to be compiled is), that should be called "*Makefile*" or "*makefile*". The reason for this naming convention is that, when using the '*make*' program from the command line, it will look for a file named Makefile (or with lower case 'm') to parse for what it needs - discussed below.

When making one or more executables for a program, typically (unless it is a very basic program), you will likely be using multiple files, which will require multiple compile commands and intermediate files. A makefile is used to simplify the commands you use to recompile, and also to reduce the amount of times that you recompile.

The construction of a makefile must follow a strict format:

```
target: dependencies
 [TAB]  command
```

* *It is important to note the 'tab' before the command. This is necessary in order to differentiate dependency lines from command lines. Also, you may have more than one command per target, and each command line must start with a 'tab'.*

The **target** is the file to be created by the **command** provided after the tab space. Therefore, the target can be your executable, or it can be an object code file. The **dependencies** are the files that the target file depends on when executing the corresponding command.
For example, consider a simple program which displays the following output:

```
Absolute Value of -9 is: 9
             9 cubed is: 729
         9 factorial is: 362880
           9 squared is: 81
```

For simplicity, this program has 4 different '.c' files with one function each, to calculate each result (there is also a corresponding '.h' file). Listing the contents of this directory looks like this:

```
~/ClassNotes/Makefiles 7 $ l
absoluteValue.c  cubed.c  factorial.c  main.c    squared.c
absoluteValue.h  cubed.h  factorial.h  Makefile  squared.h
```

To create the executable *main*, object code for each of the 4 '.c' files will need to be linked together with the 'main.c' file. So, '*main*' will be the target file that we want created, and creating that target is dependent on 4 object code files. We also need the appropriate command to actually link these together. The makefile for this will look like the following:

```
main: main.c absoluteValue.o cubed.o factorial.o squared.o
        gcc -std=c99 main.c absoluteValue.o cubed.o factorial.o
squared.o -o main
```

* *Note how the command line wraps around. The only way you can 'tab' over the second line of the command line (to make it easier to read) is to use the backslash character, so that it is still considered the same command, as follows:*

```
main: main.c absoluteValue.o cubed.o factorial.o squared.o
        gcc -std=c99 main.c absoluteValue.o cubed.o \
        factorial.o squared.o -o main
```

* *Above, target is blue, dependency is green, and command (or rule) is red.*

Of course, we now need to create our object code files. If we want to create our object code for our '*factorial.c*' file, we can do the following:

```
factorial.o: factorial.c factorial.h
        gcc -std=c99 -c factorial.c
```

The way that we use our Makefile is to type '*make*' in the command line, followed by the name of the target that we want to create. So, to create '*factorial.o*', we simply type '*make factorial.o*' in the command line, and the make program will execute '*gcc -std=c99 -c factorial.c*'.

If we've made all of the necessary target-dependency-commands for each of the '.o' files that we need, we can use the '*make*' program to make our lives much easier. Since the depencies for making our '*main*' executable include each of the '.o' files, and each of those '.o' files are their own targets (which have commands to compile), we can simply type '*make main*' in the command line, and this will perform the following:

```
~/ClassNotes/Makefiles 75 $ make main
gcc -std=c99 -c absoluteValue.c
gcc -std=c99 -c cubed.c
gcc -std=c99 -c factorial.c
gcc -std=c99 -c squared.c
gcc -std=c99 main.c absoluteValue.o cubed.o factorial.o
squared.o  -o main
```

The make program does this by examining each of the dependencies, and if any of those files are targets themselves, it will execute their command(s) only if the file has been modified since its most recent compliation. If any dependencies were last updated before the target was last updated (by executing its command), it will not compile anything. If only a couple of the dependencies were updated more recently than the target, then only those dependencies will get recompiled, and only if necessary, all of the dependencies will be updated (as above). Especially as programs get very large, and remembering which files you've been working on since last compiling gets more difficult, this feature of the make program can greatly reduce the amount of files that get recompiled. For example, if after the previous code example, we were to update just our '*cubed.c*' file, typing '*make main*' again would look like this:

```
~/ClassNotes/Makefiles 5 $ make main
gcc -std=c99 -c cubed.c
gcc -std=c99 main.c absoluteValue.o cubed.o factorial.o
squared.o -o main
```

* *Only cubed.c needed to be recompiled to updated object code, and because of that, the main executable required that the object code files be linked again. However, nothing else gets recompiled that doesn't need to be, making things easier for us, and making less work for the compiler.*

Now, after simply typing '*make main*' in the command line, our directory looks like the following:

```
~/ClassNotes/Makefiles 9 $ l
absoluteValue.c  cubed.c  factorial.c  main*     squared.c
absoluteValue.h  cubed.h  factorial.h  main.c    squared.h
absoluteValue.o  cubed.o  factorial.o  Makefile  squared.o
```

* *We now have our executable main*, and we have all of the necessary object code that needed to be linked together.*

Macros:

Similar to any C program, you can use macros when you have things that appear multiple times in your makefile, so that they can be switched in just one line of code, rather than every line that they appear in. Also, with meaningful macro names, it can make code easier to read. The following example demonstrates a few different macros, and how they would be applied to our example:

```
#  macro definition - compiler flags
CFLAGS = -std=c99 -c

#  macro definition - object files
OBJ = absoluteValue.o cubed.o factorial.o squared.o

#  macro definition - executable
EXEC = main

#  compiling & linking
```

```
${EXEC}: main.c ${OBJ}
	gcc -std=c99 main.c ${OBJ} -o ${EXEC}
absoluteValue.o: absoluteValue.c absoluteValue.h
	gcc ${CFLAGS} absoluteValue.c
cubed.o: cubed.c cubed.h
	gcc ${CFLAGS} cubed.c
factorial.o: factorial.c factorial.h
	gcc ${CFLAGS} factorial.c
squared.o: squared.c squared.h
	gcc ${CFLAGS} squared.c
```

* *Macros are highlighted in red. Changing any of the macro definitions will change each instance of that macro everywhere else that it appears. Note: the backslash character to used to indicate a new line in makefiles, so the '#' symbol is used to add comments (highlighted in blue).*

Extras:

One important addition to makefiles, is providing a command to safely remove your executable(s), and any object files, so that you can clean things before compiling. In this example, '*clean*' is considered a *phony target*, and does not have any dependencies - it simply executes the command when called.

```
clean:
	rm -f ${OBJ} ${EXEC}
```

* *The file 'clean' does not exist (or get created) in this instance. If for some reason it ever did (in the same directory), executing 'make clean' would not execute it's command. Since it has no dependencies, it would be considered up to date. To avoid this, you can use a special built-in target name, '.PHONY'.*

```
.PHONY: clean
clean:
	rm -f ${OBJ} ${EXEC}
```

* *Now, 'make clean' will run its commands regardless of whether or not there is another file named 'clean'.*

Makefiles need not just be used for compiling. Suppose you are going back and forth between two or more files a lot, and you find it annoying to write '*vim absoluteValue.c*' & '*vim absoluteValue.h*' back and forth. You could add this code to your makefile:

```
eac: absoluteValue.c
     vim absoluteValue.c
eah: absoluteValue.h
     vim absoluteValue.h
```

** In this example, 'e' represents that you're opening up something to edit, 'a' represents that it's one of your absoluteValue files, and 'c' and 'h' represent that it's either the '.c' file or the '.h' file. Given more complicated names in bigger programs, this could be very useful and efficient.*

Now, you could type 'make eac' and your vim editor will open up to edit your specified file - in this instance, '*absoluteValue.c*'.

The complete makefile, as examined in this tutorial:

```
#  macro definition - compiler flags
CFLAGS = -std=c99 -c

#  macro definition - object files
OBJ = absoluteValue.o cubed.o factorial.o squared.o

#  macro definition - executable
EXEC = main

#  editing source code
eac: absoluteValue.c
        vim absoluteValue.c
eah: absoluteValue.h
        vim absoluteValue.h

#  compiling & linking
${EXEC}: main.c ${OBJ}
        gcc -std=c99 main.c ${OBJ} -o ${EXEC}
absoluteValue.o: absoluteValue.c absoluteValue.h
        gcc ${CFLAGS} absoluteValue.c
cubed.o: cubed.c cubed.h
        gcc ${CFLAGS} cubed.c
factorial.o: factorial.c factorial.h
        gcc ${CFLAGS} factorial.c
squared.o: squared.c squared.h
        gcc ${CFLAGS} squared.c


#  clean/remove files
.PHONY: clean
clean:
        rm -f ${OBJ} ${EXEC}
```

References:

1) Alex Allain. *Makefiles*. URL: http://www.cprogramming.com/tutorial/makefiles.html
2) Various Authors. *Guide: Makefiles.* URL: http://www.delorie.com/djgpp/doc/ug/larger/makefiles.html
3) Richard M. Stallman and Roland McGrath. *An Introduction to Makefiles*. 1995. http://www.chemie.fu-berlin.de/chemnet/use/info/make/make_2.html
4) Richard M. Stallman and Roland McGrath. *Writing Rules*. 1995. URL: http://www.chemie.fu-berlin.de/chemnet/use/info/make/make_4.html
5) M. Jason Hinek. *Notes And Code*. 2013. URL: http://people.scs.carleton.ca/~mjhinek/W13/COMP2401/notes_and_code/