CS270 Recitation 2
“Makefile Build Exercise”

Goals

- To understand how to use a Makefile to build C source code into a program.
- To learn how to modify the Makefile for new source files and targets.

The Assignment

Make a subdirectory called R2 for the recitation; all files should reside in this subdirectory. Copy the following files to your R2 subdirectory (note: these files are part of programming assignment 1):

```
http://www.cs.colostate.edu/~cs270/Assignments/PA1/main.c
http://www.cs.colostate.edu/~cs270/Assignments/PA1/myfunctions.h
http://www.cs.colostate.edu/~cs270/Assignments/PA1/myfunctions.c
http://www.cs.colostate.edu/~cs270/Assignments/PA1/Makefile
```

1) Build the source code into an executable by typing the command below, and note which commands are executed when the program is built. Identify the executable that gets built, and run it. Does it produce correct results?

```
%> make
```

2) Next, open Makefile with a text editor and note that variables are declared at the top of the file. Now look at the rest of Makefile and observe how these variables are referenced—in particular, notice that they are referenced within enclosing “$(“ and “)” symbols. Also note the string value assigned to each variable. What do you think each variable is used for? Edit each variable name to make it more understandable to you. Leave the variables referenced within “$(“ and “)” symbols alone for now.

3) After the variable declarations are a list of *rules*, all of which have the following form:

```
target:   dependencies . . .
          shell_command
          . . .
```

***note – each shell_command must be indented once with the tab key**
By default, the top-most rule is executed first. What does the top-most rule in your Makefile depend upon? What does *that* rule depend upon? What shell commands do each of these rules execute? Could you draw a dependency graph to determine the order in which these shell commands are executed? Now update the variables referenced within “$(“ and “)” symbols to the variable names you have chosen for them, and run make again—were you careful not to break your Makefile?

4) Rules can be run individually, and rules need not have dependencies. In fact, you could write a rule that exists solely to execute a shell command. Take the “clean” rule, for instance—it deletes all object files, as well as our executable program. The clean rule is executed with the following command:

```
%> make clean
```

Now invent another rule, add it to your Makefile, and execute it using make.

5) Using everything you've learned (plus online documentation) download the following object file and edit your Makefile to use this file instead of myfunctions.o to create a new executable image named “pa1_reference”. Build this executable using Makefile and note which files are recompiled and linked.

```
http://www.cs.colostate.edu/~cs270/Recitations/R2/myfunctions_reference.o
```

6) Finally, use the “pack” rule to create a tar.gz package of all files. You may need to edit this rule slightly if it fails (why might it fail?).

```
%> make pack
```

7) Show your working reference program, “pa1_reference”, your tar file, and your new Makefile rule to the TA.