CS256—Advanced Programming  
Lab 4—6, Feb.  

Jim Rogers  

jrogers@cs.earlham.edu  

Spring’04

Objectives

1. To gain familiarity with the unix make facility.

Background

This week’s lab introduces the make facility. This facility allows you to easily maintain and keep your files up to date when you are working on a project that is split across several cpp files (and their associated header files). For each of your future assignments, you should keep all of the files for that assignment in one directory, and create a make file to manage the files in that directory.

Procedure

1. Make a project directory If you don’t already have a separate directory for the class and for Assignment 1 you should make them:

   bash-2.03$ mkdir cs256  
bash-2.03$ mkdir cs256/asgn1

2. Switch to your directory for Assignment 1:

   bash-2.03$ cd cs256/asgn1

3. Copy an example Makefile Copy a file named Makefile from the class directory:

   bash-2.03$ cp ~jrogers/cs256/lab4/Makefile .

Don’t forget the period at the end, meaning "copy to the current directory". If you don’t already have the three source files tank.h, tank.cpp and testtank.cpp from completing Assignment 1, then you should also copy them from the class directory to your directory as well:

   bash-2.03$ cp ~jrogers/cs256/lab4/{tank.h,tank.cpp,testtank.cpp} .

(The \{\ldots\} fills in each of the three filenames in turn.)

4. **What's in a Makefile?** The file `Makefile` is the usual input file for a tool called `make`. The purpose of `make` is to help you maintain and update a collection of related program files. The collection usually has header files, `cpp` files, and compiled files—object files (.o) and executables—all of which depend on each other. For example, consider the `testtank` program from the last assignment. The final product is an executable file named `testtank`, which is created by linking together two other compiled files: `testtank.o` and `tank.o`. The two compiled files were created by compiling `testtank.cpp` and `tank.cpp` (both of which used the header file `tank.h`). The complete dependencies among the files can be drawn like this:

```
      testtank
    Executable file
      |
  testtank.o  tank.o
  Object code  Object code
      |
  testtank.cpp  tank.h  tank.cpp
  Source code  Header file  Source code
```

The upward arrows in the file express how each file is created. For example, the file `testtank.o` is created by compiling `testtank.cpp` and also includes `tank.h`. To state the matter simply:

If `testtank.cpp` or `tank.h` changes, then `testtank.o` must be regenerated by giving the compiler command: `g++ -Wall -c -gstabs+ testtank.cpp`.

This requirement to sometimes regenerate `testtank.o` is one of the dependencies that the example `makefile` expresses. To see this dependency, use `emacs` to open up the file named `Makefile`.

Near the bottom of the file you’ll find these two lines:

```
testtank.o: testtank.cpp tank.h
g++ -Wall -c -gstabs+ testtank.cpp
```

The first line is called a target line, which begins with a file name and a colon. After the colon is a list of more file names. Here’s how to interpret the line: The file before the colon (called the target file) depends on the other files (after the colon). Whenever one of the files after the colon changes, the `make` tool knows that the target file needs
to be regenerated. After the target line, there is a series of commands that tell exactly how to regenerate the target file. For the case of testtank.o, we only need the one g++ command to regenerate the file. (Notice that we included the -c flag to indicate that we should only compile and not create an executable file yet. We also included the -gstabs+ flag in case we want to use the debugger.)

There is one other peculiar requirement: The command lines (such as the g++ command) must each begin with a tab (\texttt{(Ctrl)-I}, not with 8 spaces). You cannot enter a tab into an emacs buffer using just the \texttt{(Tab)} or \texttt{(Ctrl)-I} keys (why?), rather you have to “quote” it using the sequence \texttt{(Ctrl)-Q \texttt{(Tab)} or \texttt{(Ctrl)-Q \texttt{(Ctrl)-I}}. (The \texttt{(Ctrl)-Q} will always insert the next thing you type verbatim.)

As a second example of a dependency, the executable file testtank is created by compiling together the object files testtank.o and tank.o. If either of these two object files should change, then testtank also needs to be recreated. Here is the appropriate target line and command from our makefile:

```
testtank: testtank.o tank.o
g++ -Wall -gstabs+ testtank.o tank.o -o testtank
```

This target line says that if testtank.o or tank.o should happen to change, then the testtank must be regenerated with the g++ command that is shown.

5. Using Make to Regenerate a Specified Target File In order to illustrate how the make facility works, start by getting rid of all the object files and executable files. You can do this with the remove command:

```
bash-2.03$ rm tank.o testtank.o testtank
```

There are two simple ways to use the make facility to automatically regenerate your files. The first approach regenerates a specific file. For example, suppose you want to regenerate tank.o. Then you can use the make command, as shown here:

```
bash-2.03$ make -k tank.o
```

The make command will find the dependency information in the makefile file. It sees that tank.o depends on other files, so it will first ensure that those files are present (and regenerate them if necessary). In this example, the two files tank.h and tank.cpp are necessary for generating tank.o. These two files are present, so the make command proceeds to generate tank.o, using the g++ command that is specified in the makefile. When the g++ command is executed, it is displayed on the screen, so you will see this appear on the screen:

```
g++ -Wall -c -gstabs+ tank.cpp
```

The -k option tells make to keep trying to make whatever it can even if the commands for some files fail. This is handy most of the time but on big projects the errors can
scroll off the screen before the make completes. The default (without the \texttt{-k}) is to stop on the first error.

When you compile under emacs, the Compile command checks to see if there is a file named \texttt{Makefile} in your current directory. If there is, the default compile command will be \texttt{make -k} (you have to add the name of the target you want to make) instead of the normal \texttt{g++} command.

After this command finishes, you should list the files in your directory, where you will find the object file \texttt{tank.o} is once again present.

6. Using Make without Specifying a Target File You may also use the make command without specifying a file, like this:

\begin{verbatim}
bash-2.03$ make -k
\end{verbatim}

Without a specified file, the make command will regenerate the first target that it finds in \texttt{makefile}. Try this now, and you will see that the executable file \texttt{testtank} is regenerated, since \texttt{testtank} is the first target file in \texttt{makefile}. During the process of regenerating the \texttt{testtank} file, the make command had to carry out several steps. In the first step, the make command realizes that \texttt{testtank} depends on \texttt{tank.o} and also on \texttt{testtank.o}. But the file \texttt{testtank.o} is not present. So, the make command first regenerates \texttt{testtank.o}, and then it can proceed to regenerate the executable file \texttt{testtank}. On the screen you’ll see the two steps displayed:

\begin{verbatim}
g++ -Wall -c -gstabs+ testtank.cpp
g++ -Wall -gstabs+ testtank.o tank.o -o testtank
\end{verbatim}

7. Using Make to Automatically Propagate Changes

The best feature of the make command is how it automatically keeps track of exactly which object files and executable files need to be recompiled. As an example, you should now change one of your source files. I suggest that you use emacs to make a small change to the \texttt{testtank.cpp} program, perhaps just modifying a comment. Then save the new \texttt{testtank.cpp} and then issue the make command:

\begin{verbatim}
bash-2.03$ make -k
\end{verbatim}

The make facility realizes that \texttt{testtank.cpp} has changed, and therefore the object file \texttt{testtank.o} is regenerated with the command:

\begin{verbatim}
g++ -Wall -c -gstabs+ testtank.cpp
\end{verbatim}

Next, the make facility realizes that \texttt{testtank.o} has just changed, and therefore the executable file \texttt{testtank} is regenerated with:

\begin{verbatim}
g++ -Wall -gstabs+ testtank.o tank.o -o testtank
\end{verbatim}
But, notice that the object file tank.o was not recompiled. The dependencies in the
makefile were sufficient to show that tank.o did not need recreation.

8. Using Make with Special Targets
A makefile can also have special target lines that carry out special actions rather than
regenerate files. There are two such target lines at the bottom of the example makefile:

```
clean:
   rm -f $(EXPENDABLES)

all:
   rm -f $(EXPENDABLES)
   @make $(EXPENDABLES)
```

The special target clean simply removes all of the files that can be generated. (These
are called the "expendable" files because they are easy to replace by regenerating
them.) It does the removal with the usual Unix remove command (rm -f, where the
-f means that it won’t complain if the files were already previously removed). The
term $(EXPENDABLES) is a macro expansion—the make command will look elsewhere
in the file for the meaning of the word EXPENDABLES. It will find this at the top of
the file, where EXPENDABLES is defined to be the sequence of all files that we can
regenerate.

The special target "all" also removes all of the files that can be regenerated, and then
it goes ahead and regenerates them by executing the make command itself. The @
symbol in front of make command suppresses printing of the command, so that "make,
.." will not be printed on the screen. The two special targets can be activated just
like any other target. For example:

```
bash-2.03$ make -k all
```

The make all command is useful when you want to make certain that all files are up
to date, not trusting make’s mechanism of determining when a file was last altered.

9. Using other Macros The macro facility is handy for specifying fragments of commands that may occur frequently in the commands of the makefile and which are likely to always be the same. An example is the options for the compiler: -Wall -gstabs*. Edit the Makefile and define a macro CPPFLAGS at the top, just as EXPENDABLES
is defined. This will represent these options everywhere they occur in the file:

```
CPPFLAGS = -Wall -gstabs*
```

Then modify each of the g++ command lines to use $(CPPFLAGS) rather than the
options themselves:

```
g++ $(CPPFLAGS) tank.o testtank.o -o testtank
```
The reasons for doing this are twofold: when the time comes to change the options, if you upgrade to a debugger that uses a different format for the debugging information than stabs+, for instance, you only have to make the change in one place. The second reason is that make is capable of mostly figuring out on its own what command is appropriate to make many files. The way it knows what compiler options you prefer is by using this (CPPFLAGS) macro.

rm one of your object files and do the make -k again. You should see the compile proceed with the same -Wall -gstabs+ options.

10. **Using Implicit Commands** If you don’t specify the command to use (that is, just specify the target and its dependencies without giving the command line) then make will attempt to apply built in rules to figure out what commands to use. It can’t always do this and it doesn’t always get it right. But if you let it know what compiler flags you like by defining the CPPFLAGS macro, it will usually be able to make your object files without you specifying how to do it.

In the Makefile, remove the command lines for the tank.o and testtank.o targets. Remove only the command lines don’t remove the first line that has the target and dependencies on it. (Remember to save the file.)

In the ssh window give the command:

```
bash-2.03$ touch tank.h
```

This has the effect of making it look like tank.h was modified without actually having to modify it.

Then issue the make -k command again. You should see each of your object files get regenerated—using options you specified in CPPFLAGS—along with the executable.

```
g++ -Wall -gstabs+ -c testtank.cpp
g++ -Wall -gstabs+ -c tank.cpp
g++ -Wall -gstabs+ tank.o testtank.o -o testtank
```

In practice, there is little real need for implicit rules. Usually one creates an initial Makefile for one project and then just copies it with modifications for every subsequent project. Even complicated command lines, then, can just be cut-and-pasted over and over. There is, however, a good reason to use the CPPFLAGS variable. That way you can change the compile options globally for every compile command with a single modification of the Makefile.

**Assignment**

Please submit two things:

1. A transcript of your ssh session while you do the following:
   (a) touch tank.cpp
   (b) make -k
(c) Edit the Makefile to insert ‘#’ before the command for the testtank target (that is, before the second line for testtank, the one that starts with tab and g++.... This will mark it as a comment, effectively removing it as well.

(d) make clean

(e) make -k
    (What happened? Was your make successful?)

(f) ls

(g) Uncomment the command for the testtank target. (Remove the ‘#’. Be careful to not disturb the tab that follows it.)

(h) make -k

Of course your transcript will not include what you did in emacs. It will only have the shell commands and their results.

2. A listing of your final Makefile.

This document is a somewhat modified version of Michael Main and Walter Savitch’s example Data Structures—Lab Exercise 3.