Makefiles – A making of a file, by Eugene Kolodenker

With samples borrowed from Professor Ari Trachtenberg and Opussoftware
Version 1.0 4/5/2012 - eugenek[at]bu.edu

Introduction:

Compiling source code can be a long and tedious task. It might not be a problem when you're dealing with 1 or 3 files, but when projects become more than what any one person can handle efficiently it's time for a makefile. They also can add ease to small projects. Additionally you don't want to have to recompile everything in a project for every small change, sometimes compilation can take on the order of hours or even days. Instead, you can check the timestamp of different files and see if your small edit actually has to recompile every file, or just a specific file.

Makefiles are specific to Linux. When working with Windows, the make process will be handled for you through an IDE such as Visual Studio. When you press that "make & debug" button or hotkey, it does its magic. However, this isn't the case for Linux (when is it ever easy?).

Tutorial:

1. Please look at the appendix for the files necessary. Create each of those files by copying them into their respective **bolded** file. The makefile **<u>must</u>** be named 'makefile', no extension.

2. As you learned in class, you can compile all these files into one final executable with the command: *g++ fib.cpp factorial.cpp main.cpp* generalized into g++ file1.cpp file2.cpp file3.cpp ...
However, as you learned in the beginning of the semester, there is more to the compilation stage. This easy command is actually a set of commands:

```
g++ -c fib.cpp
g++ -c factorial.cpp
g++ -c main.cpp
g++ fib.o main.o factorial.o -o hello
```

The -c argument tells the compiler to stop the linker. The three main steps in this compilation process is, compile (process for syntax errors), assembling (assembly language output), and linking (creating the final executable). The -c argument, stops that last step from happening, and instead you are left with 3 **object** files in assembly or machine code. The final line of the expanded code links these 3 object files and creates and executable called 'hello'.

3. As you are probably thinking, this is cumbersome and lame. Makefiles to the rescue! They take something boring and lame and automate it for you, so you can be excited during your entire programming experience.

First, begin by removing all the object files (use rm). Next, type "*make hello*" (you must have the makefile created). Observe what happens. The compiler does the same steps you did earlier, with an additional flag -Wall, which enables all warnings.

4. Now, try making a change to *fib.cpp,* and then try '*make hello'* again. You'll notice that the only instructions completed were the recreation of the fib.o object file and the final linking command. This is because; the only file that actually changed in your project was *fib.cpp.*

5. Next, try the command "*make clean*". You'll notice a different instruction was completed, "rm -rf *.o". This command recursively and forcefully removes all files that end with .o. In our case, all the .o files. **Be careful with such commands!**

How it all comes together:

1. Open up the makefile in your favourite text editor (*nano makefile*). You'll notice that the first two lines are:

```
CC=g++
CFLAGS=-c –Wall
```

Makefiles allow you to create variables known as macro definitions. In a makefile, expressions of the form $(name) are replaced with the right hand value after the equal sign of name, as shown in the next two lines:

```
main.o: main.cpp
   $(CC) $(CFLAGS) main.cpp
```

The first line of this part, is called a dependency line. They determine when the target is to be rebuilt – if any changes happen to main.cpp, remake main.o, using the below command: *$(CC) $(CFLAGS) main.cpp* which is actually *g++ -c –Wall main.cpp*. **Note: It is important to tab after a dependency rule line. It will not work otherwise.**

2. The factorial file and fib file both work under the same idea. The next unique line is:

```
all: hello
```

This means that the output, "all", is dependent on hello. If you were to type "make all", it would then see if hello has changed, where hello is:

```
hello: main.o factorial.o fib.o
```

hello, is dependent on main.o, factorial.o and fib.o.

The take away points from this are that, you can specifically "make" outputs, commands such as "make fib.o" and "make all", are valid. It will follow the rule for that name.

3. Finally, the last command is the clean. This means that when you type *make clean*, it will run the rule for "clean", which is "rm –rf *o".

4. An important note about makefiles also is that, the first dependency is the one used when you just type "make", in our case this would be the "all". This is why the "all" is usually listed first.

Additional Information/Resources:

http://www.opussoftware.com/tutorial/TutMakefile.htm
http://www.gnu.org/software/make/manual/make.html
http://mrbook.org/tutorials/make/
http://en.wikipedia.org/wiki/Make_(software)
http://google.com

Files necessary for this tutorial:

main.cpp

```
#include <iostream.h>
#include "functions.h"

int main(){
    cout << "The fifth Fibonacci number is " << fib(5);
    cout << endl;
    cout << "The factorial of 5 is " << factorial(5) << endl;
    return 0;
}
```

functions.h

```
int fib(int n);
int factorial(int n);
```

fib.cpp

```
#include "functions.h"

int fib(int n){
    if(n==0 || n==1)
    return 1;
    else
    return fib(n-1)+fib(n-2);
}
```

factorial.cpp

```
#include "functions.h"

int factorial(int n){
    if(n!=1){
    return(n * factorial(n-1));
    }
    else return 1;
}
```

makefile

```
CC=g++
CFLAGS=-c -Wall

all: hello

main.o: main.cpp
    $(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
    $(CC) $(CFLAGS) factorial.cpp

fib.o: fib.cpp
```

```
    $(CC) $(CFLAGS) fib.cpp

hello: main.o factorial.o fib.o
    $(CC) main.o factorial.o fib.o -o hello

clean:
    rm -rf *.o
```