A Simple Makefile Tutorial

Modified by Dr. Donald House for use at Clemson in CPSC 1070 from a tutorial developed at Colby College by Dr. Bruce Maxwell used by permission

Makefiles are a simple way to organize code compilation. This tutorial does not even scratch the surface of what is possible using *make*, but is intended as a starter's guide so that you can quickly and easily create your own makefiles for small to medium-sized projects.

A Simple Example

Let's start off with the following three files, `hellomake.c`, `hellofunc.c`, and `hellomake.h`, which would represent a typical main program, some functional code in a separate file, and an include file, respectively.

hellomake.c

```
#include "hellomake.h"

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.c

```
#include <stdio.h>
#include "hellomake.h"

void myPrintHelloMake(void) {

  printf("Hello makefiles!\n");

  return;
}
```

hellomake.h

```
/*
  example include file
*/

void myPrintHelloMake(void);
```

Normally, you would compile this collection of code by executing the following command:

```
gcc -g -o hellomake hellomake.c hellofunc.c
```

This compiles the two .c files and names the executable hellomake. The -g flag tells the compiler to produce information needed by the debugger (we will look at using the debugger during the next lab). Without a makefile, the typical approach to the test/modify/debug cycle is to use the up arrow in a terminal to go back to your last compile command so you don't have to type it each time, especially once you've added a few more .c files to the mix.

Unfortunately, this approach to compilation has two downfalls. First, if you lose the compile command or switch computers you have to retype it from scratch, which is inefficient at best. Second, if you are only making changes to one .c file, recompiling all of them every time is also time-consuming and inefficient. So, it's time to see what we can do with a makefile.

The simplest makefile you could create would look something like:

Makefile 1

```
hellomake: hellomake.c hellofunc.c
    gcc -g -o hellomake hellomake.c hellofunc.c
```

If you put this rule into a file called `Makefile` or `makefile` and then type `make` on the command line it will execute the compile command as you have written it in the makefile. Note that `make` with no arguments executes the first rule in the file. Furthermore, by putting the list of files on which the command depends on the first line after the `:`, make knows that the rule `hellomake` needs to be executed if any of those files change. Immediately, you have solved problem #1 and can avoid using the up arrow repeatedly, looking for your last compile command. However, the system is still not being efficient in terms of compiling only the latest changes.

One very important thing to note is that there is a tab before the `gcc` command in the makefile. There must be a tab at the beginning of any command, and `make` will not be happy if it's not there.

In order to be a bit more efficient, let's try the following:

Makefile 2

```
CC = gcc
CFLAGS = -g

hellomake: hellomake.o hellofunc.o
    $(CC) -o hellomake hellomake.o hellofunc.o
```

So now we've defined some constants `CC` and `CFLAGS`. It turns out these are special constants that communicate to `make` how we want to compile the files hellomake.c and hellofunc.c. In particular, the macro `CC` tells `make` what C compiler to use, and `CFLAGS` is the list of flags to pass to the compilation command. By putting the object files--`hellomake.o` and `hellofunc.o`--in the dependency list and in the rule, `make` knows it must first compile the .c versions individually, and then build the executable hellomake.

Using this form of makefile is sufficient for most small scale projects. However, there is one thing missing: dependency on the include files. If you were to make a change to hellomake.h, for example, `make` would not recompile the .c files, even though they needed to be recompiled. In order to fix this, we need to tell `make` that all .c files depend on certain .h files. We can do this by writing a simple rule and adding it to the makefile.

Makefile 3

```
CC = gcc
CFLAGS = -g
DEPS = hellomake.h

%.o: %.c $(DEPS)
    $(CC) $< $(CFLAGS) -c -o $@

hellomake: hellomake.o hellofunc.o
    $(CC) $(CFLAGS) -o hellomake hellomake.o hellofunc.o
```

This addition first creates the macro DEPS, which is the set of .h files on which the .c files depend. Then we define a rule that applies to all files ending in the .o suffix. The rule says that the .o file depends upon the .c version of the file and the .h files included in the DEPS macro. The rule then says that to generate the .o file, `make` needs to compile the .c file using the compiler defined in the CC macro. The -c flag says to generate the object file, the `-o $@` says to put the output of the compilation in the file named on the left side of the `:`, the `$<` is the first item in the dependencies list, and the CFLAGS macro is defined as above.

As a final simplification, let's use the special macros `$@` and `$^`, which are the left and right sides of the `:`, respectively, to make the overall compilation rule more general. In the example below, all of the include files should be listed as part of the macro DEPS, and all of the object files should be listed as part of the macro OBJ.

Makefile 4

```
CC = gcc
CFLAGS = -g
DEPS = hellomake.h
OBJ = hellomake.o hellofunc.o

%.o: %.c $(DEPS)
    $(CC) $(CFLAGS) -c -o $@ $<

hellomake: $(OBJ)
    $(CC) $(CFLAGS) -o $@ $^
```

So, now let us look at a makefile that you could use for your EZ Draw projects. Start by creating a directory named `ezdraw` at the top of your directory structure, and place the `ezdraw.h` and `libezdraw.a` files in it. The NAME macro should be set to the name of your program. The IDIR and LDIR macros define paths to this new directory used for the EZ Draw include and lib directories. The LIBS macro includes the names of any libraries that your

program needs during linking, such as the math library `-lm`. The complicated looking CFLAGS and LDFLAGS macros define the compiler and loader flags that should be used. The `sdl2-config` entries call a SDL2 program that inserts text into the macro. These provide special compiler flags needed when compiling a program dependent on SDL2, where to find the include files for SDL2, and where to find the SDL2 library. Note that this `makefile` also includes a rule for cleaning up your source and object directories if you type `make clean`.

Makefile 5

```
CC = gcc
NAME = tiger

# Assumes you have a directory ezdraw at the top of your directory structure
# that contains the ezdraw.h and libezdraw.a files
IDIR = ~/ezdraw/
LDIR = ~/ezdraw/

# Libraries to load
LIBS = -lezdraw -lm

# Warnings frequently signal any errors
# Choose compiler flags for SDL2, and tell where to find the SDL2 include files
CFLAGS = `sdl2-config --cflags` -g -W -Wall -Wextra -pedantic -O0 -I `sdl2-config --prefix`/include/

# Tell where the SDL2 libraries are and request them to be loaded
LDFLAGS = `sdl2-config --libs`

OBJS = $(NAME).o

%.o: %.c $(IDIR)ezdraw.h
    $(CC) $(CFLAGS) -I $(IDIR) -c $< -o $@

$(NAME): $(OBJS) $(LDIR)libezdraw.a
    $(CC) $(CFLAGS) -o $@ $(OBJS) -L $(LDIR) $(LIBS) $(LDFLAGS)

clean:
    rm -f *.o
    rm -f *~
    rm -f $(NAME)
```

So now you have a perfectly good makefile that you can modify to manage small and medium-sized EZ Draw software projects. You can add multiple rules to a makefile; you can even create rules that call other rules. For more information on makefiles and the `make` function, check out the GNU Make Manual, which will tell you more than you ever wanted to know (really).