The GNU `make` utility and Makefiles

The 'make' method serves as a recipe for your code. It makes compilation easier and cleaner. Here, we'll see how to use 'make' through Makefiles.

We'll cover the following

- Introducing more generalization to the Makefile

There is a UNIX tool called `make` that is commonly used to compile C programs that are made up of several files, and (sometimes) involve several compilation steps. There is a lot of power in the `make` tool, but what I want to introduce here is a simple use of it, which lets you avoid having to remember a long, complicated compile command (e.g. in line 1 of the output from the prime number program above).

The `make` utility uses a special plain-text file that you write that has to reside in the same directory as your program, and has to be called `Makefile`. You can think of a Makefile as a **recipe** for making your program (i.e. linking and compiling).

A simple Makefile for our prime number program above might look like this:

```
go: go.c primes.c
	gcc -o go go.c primes.c
```

Makefile1.txt

The first word and colon `go:` on line 1 represents the **name** of a recipe called **go**. The list of files after the colon (`go.c primes.c`) represent all of the things that go **depends upon**. On the next line, there is a `TAB` (not spaces) followed by a compile command. This represents the steps (there could be more lines for more steps if there were any) that are required to “make” the “go” recipe. Here we simply have put our compile command.

Now all we have to do from the command-line is type `make`, and make will “make” the recipe for “go”. (Running `make` with no arguments executes the first rule (recipe) in the Makefile). The `make` program knows that the “go” rule needs to be

executed if any of the files that it depends upon, (because they follow the colon on line 1) changes.

Here is what it looks like when we run `make` using `Makefile1.txt`:

```
plg@wildebeest:~/Desktop/CBootCamp/code/primes$ cp Makefile1.txt Makefile
plg@wildebeest:~/Desktop/CBootCamp/code/primes$ make
gcc -o go go.c primes.c
```

You can see the command (line 3) that ends up being executed by `make`.

Introducing more generalization to the Makefile

There are a number of features of a Makefile we can utilize to make the whole idea more useful. We can introduce “macros” (like a variable) to generalize the name of the C compiler to use, the flags to pass the compiler, the location of any library files, etc etc. Here is what a more generalized Makefile might look like for our primes example from above:

```
CC = gcc
CFLAGS = -Wall
DEPS = primes.h
OBJ = go.o primes.o

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

go: $(OBJ)
	gcc $(CFLAGS) -o $@ $^
```

Makefile2.txt

You can see we have moved all the specific details (filenames, compiler flags, etc) into the macros on the top, and what remains below in the rules themselves, is expressed only in terms of those macros. There’s nothing wrong with using a Makefile that is simple (as in `Makefile1.txt`, it is a choice for you about how fancy to get. The one limitation of `Makefile1.txt` is that the header file `primes.h` doesn’t appear anywhere … this means if that file changes, then `make` will not think it has to recompile anything (because nothing in the rule “go” depends on `primes.h`). In `Makefile2.txt`, we introduce a dependency of `.o` files (on line 6) on `$(DEPS)`, which is defined above on line 3, and includes the header file `primes.h`.

Note that we have term in the `CFLAGS` macro that looks like this: `-Wall`. This is a flag to the compiler to turn on all Warnings. There are many warnings that the compiler will tell you about, like variables that are never used, uninitialized

variables, etc. Consult the documentation for full details.

Here is what it looks like when we run `make` using `Makefile2.txt`:

```
plg@wildebeest:~/Desktop/CBootCamp/code/primes$ cp Makefile2.txt Makefile
plg@wildebeest:~/Desktop/CBootCamp/code/primes$ make
gcc -c -o go.o go.c
gcc -c -o primes.o primes.c
gcc -o go go.o primes.o
```

You can see that in this case, three commands end up being run (lines 3-5).

Now the neat thing is, if we type `make` again (without changing anything) we get this:

```
plg@wildebeest:~/Desktop/CBootCamp/code/primes$ make
make: `go' is up to date.
```

We are told that the “go” rule is “up to date”. The `make` program checks to see which files have changed since the last make, and only executes the step in the rule(s) (if any) that need to be done (it figures this out based on the dependencies that you set up in the rules).

In the long run, using Makefiles is a good idea, because:

- it’s faster to recompile things (less typing, and it only recompiles based on what’s changed and leaves the rest)
- it organizes all the “steps” in a (potentially complex) compilation into one place (the Makefile), which makes it easier for other people to compile your code

See the links in the next lesson for more details and examples of how GNU make can be used to your advantage.

The next chapter in this course deals with the GNU Project Debugger. It is a very useful tool used for detecting and handling errors in our code.