CO222: Programming Methodology
Lab: 09

Deadline: Mar 25th 2016 @ 11.55PM

Learning to write a Makefile is important and it will be a skill you'll appreciate when starting to take on more complex software projects.

The Unix utility program ***make*** is used by programmers to automate the process of recompiling their code. This can be especially useful when a program contains many files. As well as determining how to compile and link the component files and libraries, make can determine which files need to be re-compiled if any of the files in the program are updated or modified. To use make, you need to write a file called Makefile that specifies what the target program is and how it is to be generated. This is done by specifying rules describing which files are required to generate the target and the actions that need to be performed. When make is run, it automatically determines which files need to be compiled to generate the target program and then issues the necessary commands to do so. This lab will take you through the process of writing a Makefile and running make.

The Vigenere Cipher

The program for which you will be writing the Makefile implements a coding system called the "Vigenere cipher". More information on the Vigenere cipher can be found on this Wikipedia page. https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher

Instructions

- To get started, create a new directory download and extract the lab09.zip file. Notice that the file vigenere.h is #include'd in both cipher.c and vigenere.c because it contains declarations and function prototypes that are needed by both .c files. The global variable char key[] has an extern declaration because it is used by both files.

- Create a file called Makefile and specify your first rule. We'll develop our Makefile using a "bottom-up" approach, i.e., we specify the rule for making the target executable first and then work on defining its dependencies, which in turn also needs to be defined further and so on until we reach the "top", the C source files.

- Give our target executable, the name ‘cipher’. Then enter into our Makefile the name of our executable target followed by a colon (':') and a list of the files (object files in this case) on which this target depends.

- You now need to specify how your target is to be compiled. Add the command for compiling the object files in to a single binary using gcc. Note that the second line of the Makefile (i.e., the one beginning gcc) MUST be indented by one TAB character otherwise make will not work correctly.
  To re-cap, on one line of the Makefile you need to specify a target and the files on which it depends. On the subsequent lines you need to list the actions that are required in order to

generate the target. These lines must be indented by one tab character. In this case, there is only one action to perform which involves running gcc

- When make comes to generate the target it first checks whether anything needs to be done to generate any of the files on which it depends. Once these have been made, it performs the action required to generate the target.

- Now put rules that tells how to create the .o files. This Makefile says that creation of the program cipher depends on having files cipher.o and vigenere.o Furthermore, to create the file cipher.o, the files cipher.c and vigenere.h are required, and the command gcc -Wall -c cipher.c needs to be performed. Similar rules are specified for vigenere.o. Once cipher.o and vigenere.o have been generated, the program cipher is generated by performing gcc -o cipher cipher.o vigenere.o.

- It is now time to test your Makefile. You can do so without having make actually perform the required commands by using the "-n" option, i.e., *make –n.* This will tell you which commands make will invoke without actually performing them.

- To run make and perform the commands required, simply type make by itself. Try this now. If you run the make command and it says that there is nothing to make, then simply "touch" (`see man touch`) one of the source files and that will force make to rebuild that file, i.e.,:

    touch cipher.c
    make

The touch command simply changes the date on the file you specified to the time the touch command was run. make uses the file modification times to determine which files it needs to re-compile. If the relocatable object file corresponding to a C source code file is up to date, make will not bother re-compiling it.

- Run the program by typing ./cipher

- Just as with C source code files, you can add comments to your Makefile. This is a good idea. Comments in Makefiles start with the '#' character:

    # Makefile

    # This is our target

Any characters after the '#' and up to the newline character at the end of the line are ignored.

- You will notice that there is a fair amount of repetition in the current Makefile. For example, the command gcc occurs a number of times. It would be better to define a value in one place and then use this definition throughout. Variables can be introduced by expressions of the following type

    <variable name> = <value>

Once introduced, variables can be referred to in the following way

    $(<variable name>)

Applying this idea to our Makefile we can introduce variables that refer to the C compiler we will use and for any flags or options that will be used by the compiler. Use the variable names CC and CFLAGS to specify the compiler name and compiler flags such as –Wall.

If you decide to change compiler or change the options that are to be used with the compiler, there is only one place where these changes need to be made instead of the many in the previous version. Apply these changes to your Makefile and test that they work.

- make has a number of other ways of making things easier using variables. Consider the following three lines:

```
CSRC = cipher.c vigenere.c

HSRC = vigenere.h

OBJ = $(CSRC:.c=.o)
```

The first two expressions assign the variables CSRC and HSRC while the last line assigns to the variable OBJ the value cipher.o vigenere.o. The right-hand side of this expression says that the value of OBJ is obtained from the value of CSRC by replacing the .c suffix with the .o suffix. This is a useful way of specifying dependencies. Integrate these variables to your makefile.

- Another very useful rule is the following:

```
%o:%c

$(CC) $(CFLAGS) -c $<
```

The first line says that any file with a .o suffix is dependent on a file with the same name but having a .c suffix. For instance, cipher.o depends on cipher.c (i.e. the file cipher.c is required in order to generate cipher.o). The second line gives the command necessary for generating any .o file. The $< is replaced by the relevant .c file (in the case of cipher.o it would be replaced by cipher.c). Modify the Makefile in the appropriate way to use this technique.

Notice how the features mentioned above have allowed us to simplify the Makefile considerably. In particular, we no longer need individual rules to generate the .o files as this is specified by a general rule.

- Test the Makefile and make sure it works. make can also be run by specifying the target on the command line. For example,

```
make cipher
```

explicitly asks make to generate the target cipher. It does so by looking for a rule in the Makefile in which cipher appears to the left of a ':'. When you have multiple targets, you need to be explicit about which you are referring to when make is run without arguments. By default, the first target in the Makefile is the default target.

- It can also be useful to specify rules that do not generate files but perform some useful action. For example:

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

When make is run by specifying clean as the target

make clean

the action performed is to remove all the OBJ files (i.e., cipher.o and vigenere.o).

In these cases it is best to add a rule for these additional targets

.PHONY: clean

The .PHONY tells make that clean is not a file to be generated as a target but that the rule has another effect. Once you have made these modifications, try running make again.

- Once you're satisfied that your Makefile is doing what it's supposed to do, you should add one new target to your Makefile: "clobber". The role of "clean" is to remove all the temporary files that were generated during the compilation process, e.g., vigenere.o. "clobber" does all that "clean" does, but it also removes any executables generated, e.g., cipher. You invoke the "clobber" target by typing:

  make clobber

- Typing in lots of test data every time you test your program gets a little tiresome when you are trying to troubleshoot your program. As a result you tend not to test it as well or as rigorously once you reach that point. With cipher, because you are reading your input from stdin (standard input), it is possible to redirect stdin to come from a text file.

Firstly, create a new file (say test.dat) and type in some input that you want to use as test data. Here is a simple example:

co222
This is lab 9.

Once you've done that, you can feed it into your program cipher, by typing the command:

./cipher < test.dat

If all goes well then you should see the data in test.dat converted accordingly and printed out on the screen. Being able to test your code using a consistent test file means you can test your programs in a more methodical way.

Now modify your Makefile and add to it a new "phony" target called "test", which behaves like this:

make test
./cipher < test.dat
Enter key: Enter text:
vvT^TuzLM

make will automatically recompile your code if necessary and then execute the program with the test data.

When you have finished writing the Makefile submit a single zip file named E13XXX_lab9.zip where XXX is your e-number.