1 Lab 1

Date: Jan 30, 2020

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

In the listings which follow, comments are any text extending from a `#` character to end-of-line.

1.1 Aims

The aim of this lab is to introduce you to theuse of `Makefile`'s under Unix for building c++ programs. After completing this lab, you should be familiar with the following topics:

- The basic operation of `make` for building C++ programs.
- Common problems when using `make`.
- The use of `make` *variables*.
- The presence of *implicit commands* in `make`.
- Writing simple C++ programs.
- Some idea of the growth of different functions measuring program performance.

1.2 Background

A typical large program consists of multiple sub-systems and libraries. Each subsystem or library will contain multiple source files. Building the program entails compiling all sub-systems and libraries with the correct options and assembling them together. This can often be quite complex and time consuming. If any source file changes, it should be possible to rebuild the program while redoing as little work as possible. The `make` program allows the automation of such tasks. The operation of `make` is controlled by a file typically named `Makefile` in the directory where `make` is invoked.

Note that `make` is an example of a *build* tool. The `make` used in this lab is typical of that found in Unix systems. Microsoft's `nmake` is a similar program. Build tools like Java's `ant`, Ruby's `rake` and Python's `scons` have similar functionality.

1.2.1 Principles

A `Makefile` basically consists of a set of *rules*. Each rule describes the *prerequisite* files for building a *target* file and the *recipe* which needs to be carried out if any of the prerequisite files are newer than the target file.

```
target:         prequisite ...
                recipe
```

The *recipe* can consist of multiple Unix shell commands (this can include compilation commands), which must be run to make the *target* from the *prerequisite* files.

The *target* for one rule can be a *prequisite* for another rule. Hence the first rule will not be run until the prerequisite is made up-to-date by its rule. The `make` program (at least the GNU version) tracks these dependencies across any number of levels and executes all necessary recipes to bring the targets of all relevant rules up-to-date.

To build a particular target, `make` can be invoked with that `target` as its command-line argument. If invoked with no targets, it will attempt to build the target for the **first rule** in the `Makefile`.

Consider building an executable `hello` from 3 files: a `hello` module consisting of a specification header file `hello.hh` and an implementation file `hello.cc` and a `main.cc` which includes the `hello.hh` header file. This can be achieved using the following `Makefile`:

```
hello:                  main.o hello.o
                       #link main.o and hello.o to executable hello
                        gcc main.o hello.o -o hello

hello.o:                hello.cc hello.hh
                        #compile hello.c to object file hello.o
                        g++ -g -Wall -std=c++17 -c hello.cc

main.o:                 main.c hello.hh
                        #compile main.c to object file main.o
                        gcc -g -Wall -std=c++17 -c main.cv

clean:
                        rm -f *~ *.o hello
```

Note the last target `clean`. It does not have any prerequisites and hence will run its recipe whenever it is invoked (typically invoked explicitly as `make clean`). It's recipe runs the shell `rm` command which will remove all emacs backup files specified by the wildcard pattern `*~`, all object files specified by `*.o` as well as the built `hello` executable. The name `clean` is conventionally used for such targets which clean-up files built by `make` as well as any garbage files.

1.2.2 Variables in make

Note that in the previous example, both the `hello.o` and `main.o` using the compiler options `-g` to turn on debugging and `-Wall` to turn on reasonable warnings. This is a violation of the *DRY principle*, since the same options were specified multiple times. Such violations can be avoided by the use of `make` variables.

A `make` variable is defined on a line which consists of an identifier *VAR* followed by an `=` character which may be preceeded/followed by linear whitespace (i.e. whitespace within the same line) followed by a definition. If the definition is spread across multiple lines, then the last character must be a \ on all except the last line of the definition.

The use of a variable *VAR* within a rule is indicated by `$(`*VAR*`)` and is replaced by its definition. If a `$` is to occur within a rule, then it must be quoted by repeating it.

Additionally, within each rule, the special `make` variable `$@` stands for the target and the special `make` variable `$<` stands for the first prerequisite and `$^` stands for all the prerequisites with spaces between them.

With the use of variables, the previous `Makefile` can become:

```
TARGET = hello
OBJS = \
  main.o \
  hello.o

CXX = gcc
CXXFLAGS = -g -Wall -std=c++17
LDFLAGS =

$(TARGET):              $(OBJS)
                        #link $(OBJS) to executable hello
                        $(CXX) $(OBJS) $(LDFLAGS) -o $@

hello.o:                hello.cc hello.hh
                        #compile hello.c to object file hello.o
                        $(CXX) $(CXXFLAGS) -c $<

main.o:                 main.cc hello.hh
                        #compile main.c to object file main.o
                        $(CXX) $(CXXFLAGS) -c $<

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

1.2.3 Implicit Rules

Note that in the previous example, the recipes for building both `hello.o` and `main.o` are absolutely identical. In fact, a little thought will reveal that this recipe can always be used for building a `.o` file from a `.c` file. So `make` contains a set of implicit rule similar to this. If there is no recipe given for building a prerequisite file, then `make` uses its implicit rules.

With the use of implicit rules, the `Makefile` can be simplified to:

```
TARGET = hello
OBJS = \
  main.o \
  hello.o

CXX = gcc
CXXFLAGS = -g -Wall -std=c++17
LDFLAGS =

$(TARGET):              $(OBJS)
                        #link $(OBJS) to executable hello
                        $(CC) $(OBJS) $(LDFLAGS) -o $@

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

hello.o:                hello.c hello.h
main.o:                 main.c hello.h
```

Note that the rules specifying the dependencies for the `.o` file have been moved to the bottom of the `Makefile` as they are purely declarative (using the implicit rules). If it was not necessary to record the fact that both `hello.o` and `main.o` also depended on `hello.h`, then the last two lines too could be removed as `make` is capable of concluding that `hello.o` depends on `hello.c` and `main.o` depends on `main.c`.

1.2.4 Gotcha's

The `make` program evolved in the 1970's when many programming languages were line-oriented. Hence it has a line-oriented syntax with some very peculiar syntax rules which can result in extremely painful gotcha's for the unwary.

- The lines containing recipes **MUST BEGIN WITH A TAB CHARACTER**. Since most text editors do not distinguish between the display of tab and space characters, this is a very common problem (the `emacs` editor will warn you about *suspicious lines*).

- When `make` variable definitions or recipe commands extend over multiple lines, all but the last line must terminate with a \ character. There **CANNOT BE ANY SPACES** after the \ character.
- Each command in a recipe is run in a separate shell. Hence a command cannot affect the state of the shell for a subsequent command.

  For example, the following rule attempts to delete all `.o` files in directory `dir`:

  ```
  clean-dir:
                  cd dir
                  rm -f *.o
  ```

  This will not work. The first command runs in a separate shell and changes its current directory to `dir`, but then that shell terminates. The second command runs in a new shell and will delete all `*.o` in the current directory, not the `dir` directory.

  The fix for this is to run both commands within a single shell as follows:

  ```
  clean-dir:
                  cd dir; \
                  rm -f *.o
  ```

  By using the trailing \ after the first command, only a single shell is used to run the sequential shell command `cd dir; rm -f *.o` which has the desired effect.

1.3 Exercises

Follow the *provided directions* for starting up this lab in a new git `lab1` branch and a new `submit/lab1` directory. Copy all the lab1 exercises into your `submit/lab1` directory by copying the contents of the `~/cs240/labs/lab1/exercises`:

```
$ cd ~/i240?/submit/lab1
$ cp -r ~/cs240/labs/lab1/exercises/* .
$ cp -r ~/cs240/labs/lab1/.gitignore/* .
```

When the exercises mention a new Unix command you are unfamiliar with, it is a good idea to do a `man` or google lookup on that command to get an idea of its capabilities.

1.3.1 Exercise 0: Hello World

Change over to the `0-hello` directory.

```
$ cd ~/i240?/submit/lab1/0-hello
$ ls -l
```

You should see that the directory contains a single `hello.cc` file.

Simply type `make` in that directory. You should get an error message. However, now try `make hello`. You should see that `make` automatically builds a `hello` executable. Type `ls -l` to see the created file, use the command `file hello` to confirm that it is an executable, and type `./hello` to execute it. You should see the usual `hello world` message.

How did `make` know how to build `hello` even though there is no `Makefile` in the directory? The answer is by using implicit rules.

To see the list of `make`'s builtin implicit rules, type `make -p | less` The `less` command allows you to page back and forth through the output using the spacebar and the `b` key respectively). You will see that the set of rules is quite extensive. To see lines which are relevant to c++ programs, type `make -p | egrep 'cc|CXX'` (the `grep` program filters out lines which do not match the pattern given by its argument). You will see lines relevant to compiling C++ programs (but you will also see lines related to `YACC` which is a parser-generator program).

1.3.2 Exercise 1: Measuring Growth of Functions

In this course, we will be analyzing algorithms for their time and space complexity. This analysis will result in formulas in terms of n, where n is some measure of the size of the problem. The exercises in the rest of this lab will compare how the results of different complexity functions $f(n)$.

Change over to the `1-monolithic` directory. It contains a single file fns-compare.cc which is set up to print out n and the corresponding value of the function $\text{linear}(n) \equiv 100000 \times n$.

Build the program by typing `make fns-compare`. The program should build correctly using an implicit make rule. You can run it by typing `./fns-compare` and you should get n and $\text{linear}(n)$ printed out for $n \in \{1, 10, 100, 1000, 10000\}$.

If you look at the code in fns-compare.cc, you will see that there is a function `quadratic()` which is unused. The compiler can warn you about unused function, but not with the options used by the implicit make rule. You can build it by invoking the compiler directly:

```
$ g++ -g -Wall -std=c++17 fns-compare.cc -o fns-compare
```

The options used above have the following effect:

`-g` Include information necessary for debugging in the generated executable.

`-Wall` Output reasonable warnings during compilation.

`-std=c++17` Specify that the program uses C++-17.

`-o fns-compare` Specify the file to hold the executable output. If not specified, the executable will be output to `a.out`.

Because of the `-Wall` option, you should receive a warning that the `quadratic¬()` function is not being used. Modify the file to print out `quadratic(n)`; i.e. each output line should contain `n`, `linear(n)` and `quadratic(n)`. Recompile until you get a clean compile (no errors and warnings). Run the program and verify that the output has 3 "columns" containing the values of `n`, `linear(n)` and `quadratic(n)`.

A problem with our approach is that all our code lives within a single file. Though that is fine for this toy example, such an approach will not scale as the size of our program increases. The next exercise looks at partitioning a program source code among multiple source files and compiling each source file separately.

1.3.3 Exercise 2: Separate Compilation

Change over to the `2-separate-compilation` directory and take a look at the files there. This directory contains a main `fns-compare` module having specification file `fns-compare.hh` and implementation file `fns-compare.cc`. This module is responsible for iterating through specified values of `n` and calling all defined complexity functions.

However, this `fns-compare` module does not have any direct knowledge of any complexity function. Instead, each defined complexity function registers itself with the `fns-compare` module. This registration is supported by the interface given in `fns-compare.hh` which defines the following:

`FN` The initial typedef defines a `FN` to be a function type which takes a single `double` argument and returns a `double`.

`FnInfo` This is a structure containing 2 fields:

> `descr` A `char *` `NUL`-terminated C string giving a description for the complexity function.
>
> `fn` A pointer to a function implementing the complexity function.

The structure is initialized using the `FnInfo()` constructor. It uses C++ syntax to initialize the two fields directly from the arguments using the initializers after the `:` and before the empty constructor body `{ }`.

`register(FnInfo fnInfo)` This function can be called to register a `fnInfo`.

The implementation file `fns-compare.cc` uses a C++ STL `vector` to hold all the registered `FnInfo`'s with the implementation of `register()` merely adding the incoming `FnInfo` to the vector using `vector`'s `push_back()` method.

The `main()` function contained in `fns-compare.cc` prints out a header line containing n and the names of all the registered complexity functions. It then loops through values for `n`, printing out the results of each registered complexity function for that value of `n`. To ensure that columns line up, it uses a `WIDTH` constant to specify the width for each column.

Finally, `linear.cc` implements the same `linear()` function as in the previous exercise. It uses the initialization of a `static` variable to register itself with the `fns-check` module.

We will compile the files separately:

```
#compile source code *.cc into binary object file *.o
$ g++ -g -Wall -std=c++17 -c fns-compare.cc
$ g++ -g -Wall -std=c++17 -c linear.cc

#link object files into executable
$ g++ fns-compare.o linear.o -o fns-compare
```

Now you should be able to run the `fns-compare` executable.

Add a `quadratic` function to the program computing `1000 * n * n`. You should be able to do so by cutting and pasting code from the previous exercise and code from `linear.cc` in the current exercise. Build your program using `g++`. Specifically, you will need to compile your new `quadratic.cc` into a `quadratic.o` and then link `fns-compare.o`, `linear.o` and `quadratic.o` into a `fns-compare` executable. Test your program, iterating the previous steps until you are sure it is working.

At this point, you should realize that typing separate commands for compilation of each source file as well as linking all the files manually is very tedious and error prone. The next exercise will use a `Makefile` to automate the process.

1.3.4 Exercise 3: Using a Makefile

Change over to the `3-makefile` directory. You will see a slightly modified `fns-compare` module along with files defining the following complexity functions:

$$
\begin{array}{lcl}
\mathrm{lg}(n) & \equiv & 1{,}000{,}000 \times \mathrm{lg}(n) \\
\mathrm{linear}(n) & \equiv & 100{,}000 \times n \\
\mathrm{nlg}(n) & \equiv & 10{,}000 \times n \times \mathrm{lg}(n) \\
\mathrm{quadratic}(n) & \equiv & 1{,}000 \times n^2 \\
\mathrm{cubic}(n) & \equiv & 100 \times n^3 \\
\mathrm{exponential}(n) & \equiv & 10 \times 2^n \\
\mathrm{factorial}(n) & \equiv & 1 \times n!
\end{array}
$$

Besides the additions of these additional complexity functions, the changes from the previous exercise involve performing computations using `long double`'s to minimize the occurrence of overflow.

Looks at the provided `Makefile`. You should be able to understand it based on the discussion given at the start of this document. Note that all the compilation steps are done using make's implicit rules.

Compile the program simply using the command `make`. It should compile and link the program. Run it; it should print out the values of all the complexity functions at the different values of n. Note that even though the slower growth functions like `lg()` and `linear()` have very large multipliers, their output is rapidly overtaken by the higher growth functions like `exponential()` and `factorial()` even though those functions have much smaller multipliers.

Without deleting any of your `*.o` files or the `fns-compare` executable, go into `fns-compare.hh` and change the definition of `Float` from `long double` to `float`. If you now try to rebuild using `make`, nothing will happen. This is wrong!! We have made a drastic change to the interface for all the complexity functions and they should all be recompiled.

The problem is that the provided `Makefile` simply has implicit depends of the `*.o` files on the `*.cc` files. All the files should also depend on `fns-compare.hh` but `make` does not know that.

1.3.5 Exercise 4: Dependencies

Change over to the `4-dependencies` directory. The files provided are identical to those from the previous exercise except for the `Makefile`. If you look at the end of the `Makefile`, you will notice that it has the dependencies for each object file explicitly listed.

Build the executable. Unfortunately, the provided Makefile contains an error. Identify the error, it is one of the gotchas listed earlier. Once you fix the error, you should be able to build and run the executable as in the previous exercise.

However, if you now change the definition of `Float` from `long double` to `float` in `fns-compare.hh` and attempt to rebuild, all the files will be recompiled. You will get an error because the `printf()` format specifier does not match the new definition of `Float`. Revert the change to the definition of `Float` and you should be back in business.

1.3.6 Exercise 5: Auto-Dependencies

Change over to the `5-auto-dependencies` directory. The files provided are identical to those from the previous exercise except for the `Makefile`. Instead of explicitly listing the dependencies, the `Makefile` is set up to automatically generate them with help from the compiler.

Compile and run. Everything should work as before. Notice the creation of a `.deps` directory which contains dependency files for each `.cc` file.

1.3.7 Exercise 6: Produce a tar Distribution

Stay in the `5-auto-dependencies` directory. Add a target `dist` to the `Makefile` such that running `make dist` produces a `fns-compare.tar` archive which contains all the source files necessary to build the `fns-compare` executable.

The following tips will be useful:

- Look at the `tar` man page. The command which you will need will be

  ```
  tar -cf fns-compare.tar SRC_FILE...
  ```

  where `SRC_FILE...` are all the necessary source files.
- The source files necessary can be divided into:
  - The `Makefile`.
  - All the `*.cc` files.
  - All the `*.hh` files.
- At the start of the `Makefile` all the `*.cc` files have been pulled into a `make` variable `CXX_FILES`. It should be possible to pull all the `*.hh` files into another make variable in a similar manner. Then it should be possible to define a `SRC_FILES` `make` variable containing all the source files. This `make` variable can be provided as an input to the `tar` command.

Once you build your `fns-compare.tar` distribution using your modified `Makefile`, test your distribution by unpacking it into an empty directory.

```
$ mkdir -p ~/tmp/fns-compare
$ cd ~/tmp/fns-compare
$ tar -xvf ~/i240?/submit/lab1/5*/fns-compare.tar
$ make
```

This should build the `fns-compare` program using the distributed source files.

1.4 Winding Up

Wind up your lab by using the *provided directions* to terminate your log in a `lab1.LOG` file and merging your `lab1` branch into the `master` branch. Once you have the lab on your `master` branch, commit and push your changes to github. Be sure to include your `lab1.LOG` file as well as all the exercise directories.

1.5 References

GNU Make Manual.

Advanced Auto-Dependency Generation.

Robert Mecklenburg, *Managing Projects with GNU Make*, O'Reilly, 2004.