
Getting Started: Remotely log in to the Linux network, create a new `lab4` directory in your `cmsc240` directory, and then recursively copy all the files and directories from the appropriate directory in my home directory: `cp -r /home/dszajda/outbox/cs240/lab4/* .`

Answers to all underlined questions must go in the `cmsc240_lab4_NETID.txt` (which you must rename!). And as in the previous lab, this "answer" file also has parts of questions.

The Makefile: The *make* utility reads a special file known as a makefile. The makefile (or Makefile) describes the files involved in the project and the dependencies among them. Each group of lines in a makefile has the following form.

```
targets: dependencies
<Tab> commands
```

In this context,

- `targets` is a list of file(s) to be created,
- `dependencies` is a list of files on which the target(s) depend, and
- `commands` is a list of commands used to (re)create the targets.

If any of the files in a dependency list is modified, `make` will recompile and/or relink the target.

Important: a `<Tab>` character must precede the `commands` — not spaces.

Task 1: Presented below are the contents of the very simple makefile named `Makefile` provided in the starter code for this lab.

```
hello: hello.cpp factorial.cpp main.cpp functions.h
    g++ -g -Wall -std=c++17 hello.cpp factorial.cpp main.cpp -o hello
```

In this context:

- `hello` is the target, i.e., the file to be created.
- The four files following the colon are the dependencies. If *any* of these files change, `make` will know to recreate the target `hello`.
- The second line is the command — a typical compile-and-link command for a C++ program, in this case with debugging information included, all warnings enabled, and conforming to the C++17 standard.

Execute the following commands in order, and answer the associated questions in your answer file provided with the starter code.

Q1. `make`

Q2. `make`

Q3, Q4. `touch functions.h; make`

The problem with this approach is that if any file in the dependency list changes, *all* of the source files must be completely recompiled. For example, you may have changed only `hello.cpp`, but because of the compile command used in the makefile, each of `hello.cpp`, `factorial.cpp`, and `main.cpp` must be recompiled (and then linked) — clearly not ideal.

Task 2: Start by removing the existing executable `hello` and then create a new (initially empty) makefile named `Makefile2` by executing the following:

```
rm hello; touch Makefile2
```

One way to avoid having all files be recompiled each time is to make use of object files. To create an object file for the `hello.cpp` source file, you could issue the command `g++ -g -std=c++17 -c hello.cpp` (but do not do so here). The result would be a file `hello.o`. This file is not executable because no linking has taken place, only compilation — the result of using the `-c` flag rather than the `-o` flag.

Create a rule in your `Makefile2` for creating `hello.o` similar to the following (**remember tab, not spaces!**):

```
hello.o: hello.cpp functions.h
    g++ -g -Wall -std=c++17 -c hello.cpp
```

Notice that the dependencies for `hello.o` are `hello.cpp` and `functions.h`. The `hello.cpp` source file as a dependency should be obvious. The `function.h` is also a dependency because it is included in the file `hello.cpp`.

Now create similar rules for creating `main.o` and `factorial.o`. (What dependencies do these two need? Make sure to modify the corresponding `commands` in the current `Makefile2` as well.)

By default, `make` will look for a makefile named `makefile` and `Makefile` in that order. To override and use a differently-named makefile, use the `-f` flag. In the current context, we want to use `Makefile2` as our makefile, so execute the following and answer the corresponding question in the answer file.

Q5, Q6. `make -f Makefile2`

Unless told otherwise, `make` will only make the first target encountered in the makefile. You can alternatively specify a target to make.

Q7. `make -f Makefile2 main.o`

Q8. `make -f Makefile2 factorial.o; ls -al`

We have only created object files — the source has been compiled, but not linked. To link the object files together and create an executable, you could issue the command `g++ -g -std=c++17 hello.o main.o factorial.o -o hello` (but do not do so).

Create another rule as the last rule in `Makefile2` that will create the target `hello` based on the dependencies `hello.o`, `main.o`, and `factorial.o`. (After issuing the first command below, you should have a working executable. If not, your target to create `hello` is not correct/complete.)

Q9. `make -f Makefile2 hello; ./hello`

Q10,Q11. `rm *.o hello; make -f Makefile2; ./hello`

Make the necessary modification to have `hello` be the default target created by `make`.

Q12, Q13. `touch functions.h; make -f Makefile2`

Q14, Q15. `touch hello.cpp; make -f Makefile2`

Task 3: Start by removing `hello` and all `.o` files using the following command.

```
rm *.o hello
```

Could we get `make` to handle cleanup work like this for us? Yes — the target does not *have* to be a file-to-be-created. In the examples we have seen thus far, the command has been a compile command which naturally creates a file. We can issue other command-prompt commands that do not result in a created file.

The typical way to have `make` handle the cleanup is to create a new rule at the end of your makefile. The rule should have a target called `clean` (no dependencies) with the command `/bin/rm -f *.o hello` (compare this to the first command you executed in Task 3). Add this rule to the end `Makefile2`.

Q16, Q17. `make -f Makefile2; make -f Makefile2 clean`

Task 4: Start by copying `Makefile2` to `Makefile3`.

Our makefile solution is still not ideal. Although it now no longer recompiles every source file every time, consider the steps necessary to make changes to the makefile. For example, to change the g++ compile command to some other compiler or to add an additional compile flag, we would have to make the changes for *every single target*.

Alternatively, we can use macros (sometimes referred to as variables) within the makefile. Following are some typical macro definitions (i.e., typical macro names) used in makefiles.

```
CC = g++        # the compiler to use
LD = $(CC)      # command used to link objects (usually same as compiler)

INCDIR = -I../  # additional directories to look for include files

CCFLAGS = -Wall -std=c++17 $(INCDIR) -g -c   # compiler options
LDFLAGS = -Wall -std=c++17 -g                # linker options
```

From the above example, you should notice:

- Macro names are typically in all caps.
- Previously defined macro values can be obtained by surrounding the macro name with parentheses and preceding with a dollar sign (like `CC` and `INCDIR` above).
- Any text that begins with `#` is considered a comment — don't be afraid to use comments in a makefile.

Add the above macro definitions to the beginning of `Makefile3` (don't copy-and-paste from the PDF!). Then change each of the target rules to use the macros. For example, the target to create `main.o` should now look like:

```
main.o: main.cpp functions.h
    $(CC) $(CCFLAGS) main.cpp
```

Make sure to use the LD and LDFLAGS, rather than CC and CCFLAGS, in the command that creates the final executable (i.e., hello) — after all, you are simply linking object files in this step rather than compiling.

Q18. `make -f Makefile3 clean; make -f Makefile3`

Further, you can create macros to represent lists of files. For example, you can create a macro to represent all the object files as follows.

```
OBJS = hello.o main.o factorial.o
```

Add this macro to the beginning of Makefile3 and then use the macro in *all* appropriate places (there are two places in the hello target). Also update your clean rule to use the OBJS macro rather than *.o.

Q19. `grep OBJS Makefile3; make -f Makefile3 clean; make -f Makefile3`

As another example of using a macro, modify the CC macro to use gcc instead of g++.

Q20, Q21. `make -f Makefile3 clean; make -f Makefile3`

Now change the CC macro back to use g++.

In more advanced cases, your makefile is likely to have more than one executable target. As you know, only the first target encountered will be made. How can you get all executable targets to be made by default? The typical solution is to include a rule with target all as the first rule in the file. The rule should list all executables as dependencies but contain no command, similar to the following example:

```
all: executable1 executable2 executable3 executable4
```

Issuing a make command would cause all four executables to be created.

In Makefile3, copy the rule that creates hello, making two new rules. Name the target in the first copy ciao and the target in the second copy hola, and modify the -o arguments accordingly — these copies will create the executables ciao and hola in the same way that the hello executable is created.

Also create a new macro called EXECS with the three executable names hello, ciao, and hola. Then create a new rule with target all that will cause all three executables to be created by default. Your rule should use the EXECS macro. Also update your clean rule to use the EXECS macro rather than explicitly listing the executable(s).

Q22. `grep EXECS Makefile3; make -f Makefile3 clean; make -f Makefile3; ./ciao`

More information about the make utility is available via its man pages. In particular, you should become familiar with the "automatic variables" available in make, described here:

`https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html`

Naming:

Same as usual. The name of your submission file for this lab **MUST** be cmsc240_lab4_netID.txt, where the two netIDentries are the netIDs of the group members. Note this is a .txt file not a .tar file.

Submission:

The high level picture is that to submit any labs/project in this course, you send an email to a special email address with your **single** submission file attached. This has the effect of placing your submission in the appropriate Box folder. If your submission requires more than one file, you should `tar` or `zip` your files together to create your single submission file.

The email address for this lab is

`lab4.wy4b88m5yy58023h@u.box.com`.