Select the right page in Gradescope or we will not grade the question!

Homework 4
Build Systems

Due: Saturday, October 8, 10:00PM (Hard Deadline)

Submission Instructions

Submit this assignment on Gradescope. You may find the free online tool PDFescape helpful to edit and fill out this PDF. You may also print, handwrite, and scan this assignment.

For this assignment, we will experiment in the EECS 280 W15 repository you created for Homework 2.

1 Expressing Dependencies

1. Run `make` to build everything.
2. Run `make` again (nothing happens).
3. Edit `p2.cpp` and make a change (add a comment or something).
4. Run `make` again.
5. Edit `p2.h` and make a change (add a comment or something).
6. Run `make` again.

Why did make rebuild things after step 4 but not after step 6? Why is this a problem?

Rebuild made things after step 4 because p2.cpp was included as one of the dependencies for Make all. P2.h, however, was not one of the dependencies included, so when a change was made in the header file, it wouldn't rebuild things. This is a problem because we might've changed something in p2.h that would break the build and eventually cause problems/side effects, so we want to make sure we're able to rebuild!

Rewrite the rule for `simple_test` so that make rebuilds correctly for any changes you make:[1]

There are a few ways we could do this. The easiest way is to just include all the potential header and source files inside of simple_test. Additionally, we could make simple_test PHONY, so that it would automatically rebuild every single time (however, this would rebuild it even if changes weren't made)

Think this is a pain? Check out the advanced homework for a BetterWay^{TM}.

2 From Build Engine to Rules Engine

Makefiles are often asked to do more than simply build your software. A common example is a rule named `clean` that deletes everything built by the Makefile.

1. Run `make` to build everything.
2. Run `make clean` to delete everything that was built.
3. Run `make` to build everything.
4. Run `touch clean`
5. Run `ls` (do you understand what `touch` does?)
6. Run `make clean`

Why did make run the `clean` rule after step 2 but not after step 5?

The makefile looks for something called clean inside the directory to determine if it will run or not. Because we made a file called "clean," the makefile believes that it doesn't need to run the command anymore.

What flag can you pass to make so that it unconditionally "builds" the `clean` target?

`make -__ clean` We can pass in the flag -B

Describe how to fix the Makefile so that fake targets like `clean` work correctly.

As mentioned in the answer earlier, we can make it a PHONY target. Here's what we would change.
Example:
.PHONY: clean
clean:
    rm -vf test

[1] You do not need to worry about system header files

3 Removing Duplicated Effort

Notice that currently the `all` target and the `test` target have the same list of dependencies.

List all the changes you have to make to the Makefile so that the `test` target correctly depends on the `all` target in all cases.

If I'm understanding the problem correctly, we want to make sure that all will always run before test does. If that's the case, we can remove all the dependencies from the test target, and change its dependency to "all." This way, we will guarantee that all has to run before test does!

4 Anything Special about All?

Currently, if you just type `make`, make will run `make all`. One might wonder why make chooses the `all` goal by default. While you could look this up, we are computer *scientists*. Make changes to the Makefile until you are confident that you understand how make chooses the default goal.

Describe the experiments you ran in order to determine what target make builds by default.

1. I created a new file called eric.cpp
2. I edited the make file to make a new target called eric. Inside of this target, I ran g++ eric.cpp -o eric.
3. I put this target at the top of the Makefile.
4. I ran make. Make decided to run Make eric, so I"m assuming that make builds the first target it sees inside the Makefile!

5 Manipulating **make**'s environment

One neat feature of `make` is that it ships with a large number of *implicit rules*. `make` understands that `foo.c`→`foo.o`→`foo` without you writing any rules. In fact, you can actually run `make` without a `Makefile`! Let's play with this a little.

First let's get a simple environment set up and try some things out:

```
> mkdir /tmp/wk4 && cd $_
> echo -e '#include <stdio.h>\n\nint main() {\n\tprintf("Howdy\\n");\n\treturn 0;\n}\n' > hello.c
> cat hello.c   # just so you can see what that did
> make
> make hello
> ./hello
```

Does `make clean` work? Why not? make clean did not work. I'm assuming this is because there is no implicit rule to clean!

Now try

```
> rm hello
> make -r hello
```

What does the `-r` flag do? I couldn't figure this one out without googling. It eliminates the use of implicit rules!

Next try

```
> rm hello
> make CFLAGS=-03 hello
```

What changed when hello was built this time? When we built this time, the make command compiled with the -O3 flag as well.

Finally run

```
> make hello -p | less
```

Make an educated guess at which built-in rule is used to create "hello" from "hello.c" and copy it here. What makes you think this rule is responsible?

When we run "make hello," we see cc hello.c -o hello, so I'm guessing that the cc rule is used to create it.

```
%: %.cc
#  commands to execute (built-in):
    $(LINK.cc) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

Now let's add an additional file to the mix, only a C++ file this time:
(This example uses a special shell syntax for easily writing multiple lines to a shell command)

```
> cat << MARKER > wazzup.cpp
#include <iostream>

int main() {
  std::cout << "Wazzup?" << std::endl;
  return 0;
}
MARKER
> cat wazzup.cpp
```

Using what you have learned, write a single make command (i.e. only call make once) that, without a Makefile, will build both "hello" and "wazzup", but builds hello optimized for speed (`-O3`) and wazzup optimized for size (`-Os`). ***Hint: One is a `C` program and one is a `C++` program...***

The command that I ran was this:

"make wazzup hello CFLAGS=-O3 CXXFLAGS=-Os"

I thought it would be CPPFLAGS=-Os, but unfortunately, this optimized space for the c file as well, so I googled around to find out how to do it just for C++ files.

Roughly how long did you spend on this assignment? 2 hours