Homework 6
Build Systems

Assigned: Friday, February 12, 11:00AM

Due: Friday, February 19, 11:00AM (Hard Deadline)

Submission Instructions

Submit this assignment on Gradescope. You must submit every page of this PDF. We recommend using the free online tool PDFescape to edit and fill out this PDF. You may also print, handwrite, and scan this assignment.

Even if there are not questions to answer on the first page, please still submit every page of this PDF.

There may multiple answers for each question. If you are unsure, state your assumptions and your reasoning for why you think your answer makes sense.

For this assignment, we will experiment in the EECS 280 W15 repository you created for Homework 4.

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?

All of the test targets depend on `p2.cpp`, so when it changed they rebuilt. None of them list `p2.h` as a dependency in the Makefile, so when it changed make did not know it needed to do anything.

This can be a particularly frustrating issue if a header file `#define`'s a constant that is used by code files, but the code does not rebuild correctly to use the new constant.

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

```
simple_test: simple_test.cpp p2.cpp Recursive_list.cpp Binary_tree.cpp \
          recursive.cpp test_helpers.cpp \
          p2.h Binary_tree.h Recursive_list.h recursive.h test_helpers.h
```

Notice that you need more than just `p2.h`, you also need the header files that `p2.h` includes, `recursive.h`, and then the header files that `recursive.h` includes...

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

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?

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

`make -__ clean`

`-B, --always-make` — Unconditionally make all targets.

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

Make understands that some targets, such as "clean", are not actually files that it needs to build, you just have to tell make that. These are called "phony targets". To fix the Makefile, add:

```
.PHONY: clean
```

[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.

There are two changes you should make to the original Makefile.

1. Change test so that it depends on the all target:

```
-test:  filter_test simple_test tree_insert_test p2-tests
+test:  all
```

2. Mark the all target as phony (you aren't actually making a thing called "all" anywhere):

```
+.PHONY: all
```

A different (but equally correct) approach would be to define a variable that holds all of the targets, and use that variable for both rules:

```
TARGETS = filter_test simple_test tree_insert_test p2-tests

all: $(TARGETS)

test: $(TARGETS)
      ...
```

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.

Note that this question asks about the experiments you ran, some good tests:

Rename the "all" rule to something else, what runs?

Move the "all" rule to a different location, what runs?

As a bonus, check out what happens if you rename the rule to ".all". What runs by default? Can you still "make all"?

5 Pulling Some Pieces Together

Option 3 from Question 3 on Homework 3 (during week 3!) noted:

> First cool thing that happened: Yes, you can run make without any Makefile anywhere. We'll cover how that happened during build-system week.

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

```
> mkdir /tmp/wk6 && cd $_
> echo -e '#include <stdio.h>\n\nint main() {\n\tprintf("Howdy\\n");\n\treturn 0;\n}\n' > hello.c
> make
> make hello
> ./hello
```

Does `make clean` work? Why not?

Now try

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

What does the `-r` flag do?

Next try

```
> make CFLAGS=-O3 hello
```

What changed when hello was built this time?

Finally run

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

Find the built-in rule that runs to create "hello" from "hello.c" and copy it here:

```
%: %.c
# commands to execute (built-in):
	$(LINK.c) $^ $(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`).

> Using some more of what we learned from the previous question:
>
> ```
> LINK.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
> LINK.cc = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
> ```
>
> Notice that `CFLAGS` appears only for building C files and `CXXFLAGS` appears only for building C++ files (however, `CPPFILES` appears for both, which can also be useful).
>
> ```
> make CFLAGS=-Os CXXFLAGS=-O3 hello wazzup
> ```