CS45, Lecture 11
Build Systems & DevOps

Spring 2023

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Learning Goals

- Understand how build systems like make and cmake work
- Understand what CI and CD systems are and how they're used
  - See what it's like to trigger a CI and CD pipeline
- **Have concrete experience building a Makefile for a simple C project**

Compiling & Building

- When building software, there are often a bunch of commands we have to use to **build** or **compile** that software– but that's a boring+rote task!
  - Incremental changes and iteration are **key** to software development, (so you want builds to be as seamless as possible)
  - If you're re-typing commands often it's possible to make mistakes, too
- We use **build systems** to streamline this process.
  - At its most basic, they just re-run build commands for you (**GNU Make**)
  - More complicated build systems can automatically download, install, and link *libraries* into a project (**Gradle, Maven, go, sbt, ...**) and/or integrate with IDEs to give information about your project (**cmake**)

Intro to make

If you've taken CS107 or CS111 before, you've probably seen a **Makefile** before.

GNU Make (a.k.a. **make**) is a simple build system that follows the UNIX philosophy of "do one small thing well."

It reads a **Makefile**, which specifies a set of **rules**, each of which direct **make** on how to build a certain file (or set of files).

make will then record which files have been changed, and only rebuild as needed.

Makefiles

The only thing you need in order to use make is a Makefile– so let's learn how to create one!

[demo time]

Makefile rules

Makefile **rules** define:

- the **output** (or “target”) of a rule,
- the **input** (or “prerequisites”) of a rule, and
- the **commands that generate the output from the input**

For example:

```
sample.o: sample.c
    cc -c sample.c
```

Makefile rules

Makefile **rules** define:

- the **output** of a rule,
- the **input** of a rule, and
- the **commands that generate the output from the input**

For example:

```
sample.o: sample.c
    cc -c sample.c
```

Makefile rules

Makefile **rules** define:

- the **output** of a rule,
- the **input** of a rule, and
- the **commands that generate the output from the input**

For example:

```
sample.o: sample.c
    cc -c sample.c
```

Makefile rules

Makefile **rules** define:

- the **output** of a rule,
- the **input** of a rule, and
- the **commands that generate the output from the input**

For example:

```
sample.o: sample.c
    cc -c sample.c
```

Makefile rules

Makefile **rules** define:

- the **output** of a rule,
- the **input** of a rule, and
- the **commands that generate the output from the input**

For example:

```
sample.o: sample.c
    cc -c sample.c
```

“Create **sample.o** from **sample.c** using the command **cc -c sample.c**”

Makefile rules

Makefile rules define:

- the output of a rule,
- the input of a rule, and
- the commands that generate the output from the input

Make can also deduce very simple compilations. The following rule is exactly identical to the previous one.

```
sample.o:
```

“Create **sample.o** from **sample.c** using the command **cc -c sample.c**”

Makefile rules

Makefile rules define:

- the output of a rule,
- the input of a rule, and
- the commands that generate the output from the input

We can also make the rule depend on multiple files:

```
sample.o: sample.c sample-util.h
    cc -c sample.c
```

“Create **sample.o** from **sample.c and sample-util.h** using the command **cc -c sample.c**”

Makefile rules

Makefile rules define:

- the output of a rule,
- the input of a rule, and
- the commands that generate the output from the input

And we can similarly shorten them:

```
sample.o: sample-util.h
```

“Create **sample.o** from **sample.c and sample-util.h** using the command **cc -c sample.c**”

Makefile rules

Makefile rules define:

- the output of a rule,
- the input of a rule, and
- the commands that generate the output from the input

We can use this to build simple rules for many outputs:

```
sample.o sample2.o sample3.o: sample-util.h
```

"Create **sample.o, sample2.o, and sample3.o** from **sample.c, sample2.c, and sample3.c (respectively) and sample-util.h** using the commands **cc -c sample.c, cc -c sample2.c, cc -c sample3.c (respectively)**"

Makefile rules

Makefile rules define:

- the output of a rule,
- the input of a rule, and
- the commands that generate the output from the input

Finally, we can chain rules together.

```
sample: sample.o
    cc -o sample sample.o
sample.o: sample.c
    cc -c sample.c
```

Create **sample.o** from **sample.c** using the command **cc -c sample.c**, then create the **sample** binary from **sample.o** by running **cc -o sample sample.o**.

Makefile rules

Makefile rules define:

- the output of a rule,
- the input of a rule, and
- the commands that generate the output from the input

Make will automatically figure out the order to run the rules.

```
sample: sample.o
    cc -o sample sample.o
sample.o: sample.c
    cc -c sample.c
```

Create **sample.o** from **sample.c** using the command **cc -c sample.c**, then create the **sample** binary from **sample.o** by running **cc -o sample sample.o**. Since the **sample** rule depends on **sample.o**, we'll run the **sample.o** rule first.

Makefile rules

Makefile rules define:

- the output of a rule,
- the input of a rule, and
- the commands that generate the output from the input

If no rule is manually specified, Make always runs the first rule it finds.

```
sample: sample.o
    cc -o sample sample.o
sample.o: sample.c
    cc -c sample.c
```

(Make will automatically also run any prerequisites, if necessary, in order to run the first rule it finds successfully)

Wildcards & substitutions

- In the previous examples, we **manually specified the files we were building,** e.g. **sample.o**.
- What if we wanted to write a **general rule?**

Wildcards & substitutions

- In the previous examples, we **manually specified the files we were building,** e.g. **sample.o**.
- What if we wanted to write a **general rule?**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- In the previous examples, we **manually specified the files we were building,** e.g. **sample.o**.
- What if we wanted to write a **general rule?**
- **This is complicated. Let's break it down.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- The **objects** variable contains a list of all the objects we want to compile. We use it so we can easily change it later without changing all our rules.

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- Our first rule is for compiling our **executable** (binary). That's our output program. **It depends on all the objects.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- The syntax **`$(variable_name)`** is an **interpolation**. We're "using" the variable here.

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- The syntax **`$(variable_name)`** is an **interpolation**. We're "using" the variable here.

```
objects = sample.o sample2.o sample3.o

sample: sample.o sample2.o sample3.o
    cc -o sample sample.o sample2.o sample3.o

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- **Our last rule is more complicated.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- Our last rule is more complicated.
- It is a **“static pattern rule.”** It’s like a rule that makes more rules for us.

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- Let's start with the **target**. It's our list of objects. (a.k.a. our input).

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- Let's start with the **target**. It's our list of objects.
- This static pattern rule will try to make some rules for each of our objects.

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- This part to the right is our rule pattern.

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- This part to the right is our rule pattern. This part says to find all the targets that end in **.o** (which will be all of them)

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- This part to the right is our rule pattern. This part says to **create a rule matching the corresponding .c file for each of the matched .o targets.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- **Let's expand.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- **Let's expand.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

sample.o: sample.c
    cc -c $< -o $@
sample2.o: sample2.c
    cc -c $< -o $@
sample3.o: sample3.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- **Notice that we got one rule per target...**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

sample.o: sample.c
    cc -c $< -o $@
sample2.o: sample2.c
    cc -c $< -o $@
sample3.o: sample3.c
    cc -c $< -o $@
```

Wildcards & substitutions

- This Makefile contains **two rules** and **one variable.**
- **...corresponding to the pattern `%.o: %.c`**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

sample.o: sample.c
    cc -c $< -o $@
sample2.o: sample2.c
    cc -c $< -o $@
sample3.o: sample3.c
    cc -c $< -o $@
```

Wildcards & substitutions

- **Now let's break down the rules themselves.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

sample.o: sample.c
    cc -c $< -o $@
sample2.o: sample2.c
    cc -c $< -o $@
sample3.o: sample3.c
    cc -c $< -o $@
```

Wildcards & substitutions

- Now let's break down the rules themselves.
- **They're almost the same as before, except for these weird $< and $@.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

sample.o: sample.c
    cc -c $< -o $@
sample2.o: sample2.c
    cc -c $< -o $@
sample3.o: sample3.c
    cc -c $< -o $@
```

Wildcards & substitutions

- **$< and $@ are automatic variables.** They correspond to something in the rule header/definition (the targets and prerequisites).

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

sample.o: sample.c
    cc -c $< -o $@
sample2.o: sample2.c
    cc -c $< -o $@
sample3.o: sample3.c
    cc -c $< -o $@
```

Wildcards & substitutions

- **$<** refers to the **first prerequisite** (input).
- **$@** refers to the **target** (output).

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

sample.o: sample.c
    cc -c $< -o $@
sample2.o: sample2.c
    cc -c $< -o $@
sample3.o: sample3.c
    cc -c $< -o $@
```

Wildcards & substitutions

- **$<** refers to the **first prerequisite** (input).
- **$@** refers to the **target** (output).

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

sample.o: sample.c
    cc -c sample.c -o sample.o
sample2.o: sample2.c
    cc -c sample2.c -o sample2.o
sample3.o: sample3.c
    cc -c sample3.c -o sample3.o
```

Wildcards & substitutions

- **You might sometimes also see $^**

(The following Makefile is also valid)

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $^

sample.o: sample.c
    cc -c sample.c -o sample.o
sample2.o: sample2.c
    cc -c sample2.c -o sample2.o
sample3.o: sample3.c
    cc -c sample3.c -o sample3.o
```

Wildcards & substitutions

- **$^** refers to **all the prerequisites (inputs)**, space-separated

(The following Makefile is also valid)

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $^

sample.o: sample.c
    cc -c sample.c -o sample.o
sample2.o: sample2.c
    cc -c sample2.c -o sample2.o
sample3.o: sample3.c
    cc -c sample3.c -o sample3.o
```

Wildcards & substitutions

- **$^** refers to **all the prerequisites (inputs)**, space-separated

(The following Makefile is also valid)

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

sample.o: sample.c
    cc -c sample.c -o sample.o
sample2.o: sample2.c
    cc -c sample2.c -o sample2.o
sample3.o: sample3.c
    cc -c sample3.c -o sample3.o
```

Wildcards & substitutions

- **$^** refers to **all the prerequisites (inputs)**, space-separated

(The following Makefile is also valid)

```
objects = sample.o sample2.o sample3.o

sample: sample.o sample2.o sample3.o
    cc -o sample sample.o sample2.o sample3.o

sample.o: sample.c
    cc -c sample.c -o sample.o
sample2.o: sample2.c
    cc -c sample2.c -o sample2.o
sample3.o: sample3.c
    cc -c sample3.c -o sample3.o
```

Wildcards & substitutions

- What do you think?

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Wildcards & substitutions

- What do you think?

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $^

$(objects): %.o:%.c
    cc -c $^ -o $@
```

Wildcards & substitutions

- If we didn't want to specify the objects and simply make every `.c` file referenced in the Makefile into a `.o` file, we could do the following:

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $^

%.o: %.c
    cc -c $^ -o $@
```

Wildcards & substitutions

- Or we could even just write this! **Make** knows how to do basic transformations from **.c** to **.o** files.

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $^
```

Phony rules

- What if you want to create a Makefile rule **that doesn’t correspond to a real output file?**
- Enter: Phony rules

Phony rules

- What if you want to create a Makefile rule **that doesn’t correspond to a real output file?**
- Enter: Phony rules
- Phony rules lets you tell Make that it should always run a rule even if a newer file exists with the same name.

Phony rules

- Let's say we want to add a rule that **cleans up our build.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@
```

Phony rules

- Let's say we want to add a rule that **cleans up our build.**

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@

clean:
    rm -f sample $(objects)
```

Phony rules

- However, if we accidentally have a file called **clean** we'll have issues.

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@

clean:
    rm -f sample $(objects)
```

Phony rules

- So we mark it as phony to tell Make that it should always run if we ask it.

```
objects = sample.o sample2.o sample3.o

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@

clean:
    rm -f sample $(objects)

.PHONY: clean
```

“all” rule

- Often, you’ll see a rule called **all** written like this:

```
objects = sample.o sample2.o sample3.o

all: sample

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@


.PHONY: all
```

“all” rule

- This is just a convention. **By default, make runs the first defined rule.**

```
objects = sample.o sample2.o sample3.o

all: sample

sample: $(objects)
    cc -o sample $(objects)

$(objects): %.o:%.c
    cc -c $< -o $@


.PHONY: all
```

A brief interlude: Gradle

- **Gradle** is a build system that's a little "smarter" than Make
  - At the cost of complexity
- Supports Java out of the box, but other languages too.
- **In particular, it has the capability of automatically downloading and linking in *additional libraries and dependencies.***

[gradle demo]

CI & CD

- Now that we've learned how to use a build system, let's talk about **programs that run build systems and tests for us.**
- CI: Continuous Integration
- CD: Continuous Deployment
- "Continuous" means "as you develop"
  - Most of these systems will run on each git commit
- **CI** runs **builds and tests**.
- **CD** runs **deployments**.

Continuous Integration

- Travis CI, Jenkins CI, Buildkite, GitHub Actions
- Runs on each commit
- Rationale: **create consistent build outputs** (probably executables) and **be able to tell when, where, and who when code breaks.**
- Focus: **TESTING!!**
  - Especially **unit tests**.

Continuous Deployment

- Harness, Vercel, Heroku, GitHub Pages/Actions ...
- Runs on each commit
- Lets you deploy versions of code automatically as you work on them
  - Usually deploys a **release** git branch to production, and other git branches to a **development deployment**, so that changes can be tested manually and experimented with before actually going out to the world.

Exploring CI

[github actions demo]

Exploring CD

[github pages demo]