
csce215 — UNIX/Linux Fundamentals

Spring 2022 — Lecture Notes: Time to make the donuts

This document contains slides from the lecture, formatted to be suitable for printing or individual reading, and with some supplemental explanations added. It is intended as a supplement to, rather than a replacement for, the lectures themselves — you should not expect the notes to be self-contained or complete on their own.

(5.1) *Last time*

Last time we learned about:

- How the shell passes **arguments** to the programs we run.
- How certain **special characters** such as wildcards * ? and quotation marks " affect those arguments.
- How multiple commands can be **sequenced** with && and | |.
- How **command substitution** \$() allows the output of one command to form the arguments of another command.

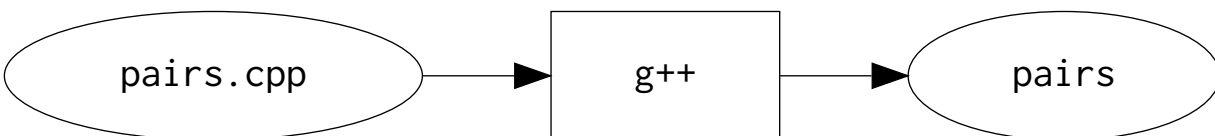
Today, we will learn about an important tool called **make** that automates the process of finding and executing commands that need to be run to keep things up-to-date.

(5.2) *Commands that create files*

A common pattern is for a shell command to process/execute/compile a file to produce another file.

Example (from Assignment 4):

```
g++ pairs.cpp -o pairs
```

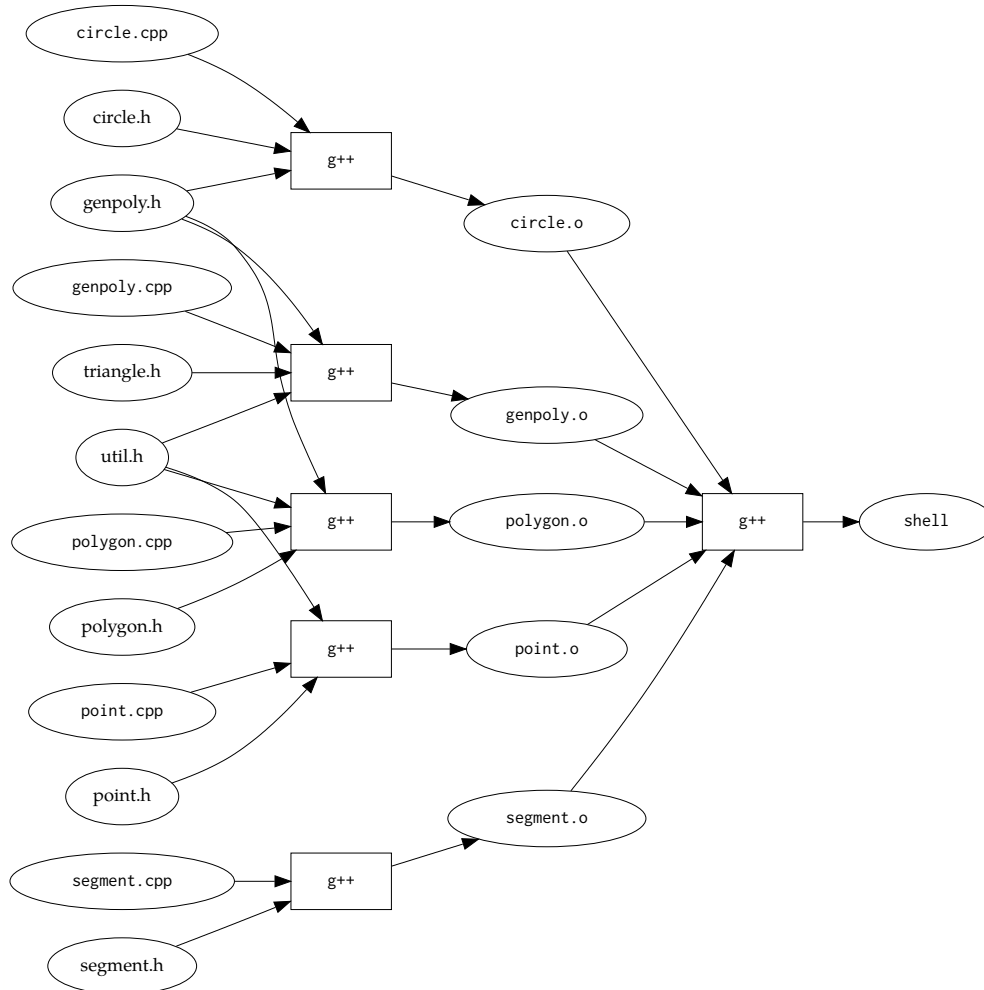


Key idea: One file **depends on** another file.

- When pairs.cpp is changed,
- run the command g++ pairs.cpp -o pairs,
- to update pairs.

(5.3) *It can get complicated!*

Here's a small slice of the dependencies from one of my C++ programs.



(5.4) *make*

make



Execute commands to create or update files, based on the rules in a makefile.

To use make, tell it what you want.

```
$ make pairs
g++ pairs.cpp -o pairs
```

This will print and execute commands to create or update the **goal** we specify.

If the goal is omitted (i.e. make by itself), the first target will be used.

(5.5) *Makefiles*

A **makefile** describes the dependencies between a collection of files. 🤖

It's a collection of **rules** like this:

- When ____ is changed, prerequisites 🤖
- run the command(s) ____, recipe 🤖
- to update _____. target 🤖

The format looks like this:

```
target: prerequisites
    recipe
```

Important (and easy to mess up): The whitespace before the recipe must be a **tab character**, not spaces. If you see an error about a 'missing separator', this is probably the problem.

(5.6) *Making a makefile in vim*

To type a **tab character** in vim in insert mode, you may need to press Ctrl-V, which means 'take the next character literally', before pressing Tab.

To check whether you have inserted Tab characters correctly, use :set hlsearch to enable highlighting of text that matches the most recent search, and then use / to search for a space.

(5.7) *A tiny example*

In the example from before, we would make a file called Makefile (capital M) that looks like this:

```
pairs: pairs.cpp
      g++ pairs.cpp -o pairs
```

Then we can build pairs like this:

```
$ make pairs
g++ pairs.cpp -o pairs
```

If we make again, without changing pairs.cpp:

```
$ make pairs
make: 'pairs' is up to date.
```

(5.8) *Why bother?*

Using a make instead of typing the commands directly has two big advantages.

Repeatability: You don't need to remember or re-type the commands each time. If you include the makefile with your source code, other developers will be able to build your project.

Efficiency: Only execute the steps that are necessary to update the target. This is mostly irrelevant for small projects, but can make a huge difference very quickly.

(Also: Most of you are likely to use make extensively in CSCE 240.)

(5.9) *Modification times*

Linux keeps track of the **modification time** for each file, which we can see using `ls -l`:

```
$ ls -l
total 44
-rw-rw-r-- 1 jokane jokane  42 Oct  1 09:00 Makefile
-rwxrwxr-x 1 jokane jokane 32840 Oct  1 09:05 pairs
-rw-r----- 1 jokane jokane 1010 Oct  1 09:00 pairs.cpp
```

When a file is edited, its modification time is changed to the current time.

(5.10) *Changing modification times*

We can also use `touch` to change the modification time of a file directly.

`touch`



Change a file's modification time to the current time.

This can be useful for testing makefiles, to 'simulate' a change to a prerequisite.

```
$ ls -l
total 44
-rw-rw-r-- 1 jokane jokane    42 Oct  1 09:00 Makefile
-rwxrwxr-x 1 jokane jokane 32840 Oct  1 09:05 pairs
-rw-r----- 1 jokane jokane  1010 Oct  1 09:00 pairs.cpp
$ touch pairs.cpp
$ ls -l
total 44
-rw-rw-r-- 1 jokane jokane    42 Oct  1 09:00 Makefile
-rwxrwxr-x 1 jokane jokane 32840 Oct  1 09:05 pairs
-rw-r----- 1 jokane jokane  1010 Oct  1 09:07 pairs.cpp
```

(5.11) *What make does*

The goal of `make` is to determine whether or not the commands in a recipe need to be executed.

The recipe commands will be executed if either:

- the target file **does not exist**, or
- any of the prerequisites are **newer than the target**.

```
$ make pairs
g++ pairs.cpp -o pairs
$ make pairs
make: 'pairs' is up to date.
$ touch pairs.cpp
$ make pairs
g++ pairs.cpp -o pairs
```

```
$ make pairs
make: 'pairs' is up to date.
$ touch pairs
$ make pairs
make: 'pairs' is up to date.
```

(5.12) *Multiple steps*

The real power of make comes when prerequisites have rules of their own.

```
report.pdf: report.tex piechart.pdf
    pdflatex report | grep Output

piechart.pdf: piechart.py
    python3 piechart.py

clean:
    rm -f -v *.pdf *.aux *.log
```

```
$ make report.pdf
python3 piechart.py
Running piechart.py to create piechart.pdf.
pdflatex report | grep Output
Output written on report.pdf (1 page, 61453 bytes).
```

```
$ touch report.tex
$ make report.pdf
pdflatex report | grep Output
Output written on report.pdf (1 page, 51775 bytes).
```

```
$ touch piechart.py
$ make report.pdf
python3 piechart.py
Running piechart.py to create piechart.pdf.
pdflatex report | grep Output
Output written on report.pdf (1 page, 51776 bytes).
```

(5.13) *Makefile variables*

We can define **variables**  in our makefiles to avoid repeating ourselves.


- Define with =.
- Use with \$.

Example:

```
CC = g++
CFLAGS = -I/usr/include/qt -c -Wall -g -finline-functions -O
LFLAGS = -lqt -o a.out
TARGET = lma

$(TARGET): filters.o image.o lma.o template.o vertex.o
    $(CC) filters.o image.o lma.o template.o vertex.o $(LFLAGS) -o $(TARGET)
filters.o: filters.cpp image.h filters.h vertex.h image.h image.h vertex.h
    $(CC) $(CFLAGS) -c filters.cpp
image.o: image.cpp image.h
    $(CC) $(CFLAGS) -c image.cpp
lma.o: lma.cpp lma.h image.h image.h filters.h vertex.h
    $(CC) $(CFLAGS) -c lma.cpp
template.o: template.cpp
    $(CC) $(CFLAGS) -c template.cpp
vertex.o: vertex.cpp vertex.h image.h filters.h
    $(CC) $(CFLAGS) -c vertex.cpp
```

(5.14) *Automatic variables*

Make recognizes a few **automatic variables** that get values automatically within each recipe. 

- Name of the target: \$@

-
- All prerequisites: $\*
 - All prerequisites newer than the target: $\$?$
 - The first prerequisite: $\$<$

```
vertex.o: vertex.cpp vertex.h image.h filters.h
$(CC) $(CFLAGS) -c $< -o $@
```

(5.15) *Fake targets*

Sometimes it helps to create 'fake' targets that are not really files. 📖

```
clean:
    rm -f -v *.pdf *.log *.aux
```

```
$ make clean
rm -f -v *.pdf *.aux *.log
removed 'piechart.pdf'
removed 'report.pdf'
removed 'report.aux'
removed 'report.log'
```

(5.16) Sample final exam questions

1. A makefile describes the _____ between a collection of files.
- A. goals
 - B. order
 - C. dependencies
 - D. rules
2. When running make, the command line arguments should be _____.
- A. Files that have changed since the last time make was run.
 - B. A list of commands that make should execute.
 - C. Files that should be updated (i.e. what you want).
 - D. Actually, make does not accept command line arguments.
3. The purpose of the make command is to _____.
- A. search for and acts on files in or below a directory
 - B. execute commands to create or update files, based on the rules in a makefile
 - C. create a new file in the current directory
 - D. find lines that match a pattern
4. Recall this explanation for the meaning of a makefile rule:
- When _____ is changed, run the command _____ to update _____.*
- The *first* blank in this explanation should be:
- A. prerequisite
 - B. recipe
 - C. target
 - D. None of the above
5. Here is an example makefile rule.
- ```
aaa.txt: bbb.txt
ccc bbb.txt > aaa.txt
```
- In this example, the filename `aaa.txt` on the first line is the rule's \_\_\_\_\_.
- A. recipe
  - B. prerequisite
  - C. target
  - D. None of the above
6. Recall this explanation for the meaning of a makefile rule:
- When \_\_\_\_\_ is changed, run the command \_\_\_\_\_ to update \_\_\_\_\_.*
- The *second* blank in this explanation should be:
- A. target
  - B. prerequisite
  - C. recipe
  - D. None of the above

---

7. Recall this explanation for the meaning of a makefile rule:

*When \_\_\_\_\_ is changed, run the command \_\_\_\_\_ to update \_\_\_\_\_.*

The *third* blank in this explanation should be:

- A. recipe
- B. prerequisite
- C. target
- D. None of the above

8. Here is an example makefile rule.

```
aaa.txt: bbb.txt
ccc bbb.txt > aaa.txt
```

In this example, when does make execute the command on the second line?

- A. when bbb.txt is older than aaa.txt
- B. when bbb.txt is newer than aaa.txt
- C. when ccc cannot be found
- D. when bbb.txt does not exist

9. The purpose of the touch command is to \_\_\_\_\_.

- A. Change a file's owner to be the current user.
- B. Show a file's access history.
- C. Change a file's modification time to the current time.
- D. Show a file's permissions.

10. Here is an example makefile rule.

```
aaa.txt: bbb.txt
ccc bbb.txt > aaa.txt
```

In this example, the command ccc bbb.txt > aaa.txt is the rule's \_\_\_\_\_.

- A. recipe
- B. target
- C. prerequisite
- D. None of the above

11. Here is an example makefile rule.

```
aaa.txt: bbb.txt
ccc bbb.txt > aaa.txt
```

In this example, the filename bbb.txt on the first line is the rule's \_\_\_\_\_.

- A. target
- B. prerequisite
- C. recipe
- D. None of the above