Make utility

- dependencies
- Makefile
- rules
- using gmake
- variables
- built-in rules
- bigger example
- naming the Makefile, and more

code and Makefiles for example in:
`~csci455/code/11-30`

Announcements

- Final exam: Tue, 12/12, 4:30pm – 6:30pm
  THH 101 / 202 (room assignments soon)
  - closed book, closed note, no extra paper
  - bring USC ID card
- Review session info and finals week office hours on week-by-week page
- Extra credit assignment available (due Tue 12/5)
- No labs this week
- Course evals available through 12/5
  - via email from c-evals@usc.edu
  - or via blackboard.usc.edu
- Check that MT2 scores were entered correctly on d2l (and other scores as you receive them)

Class time for course evaluations

Your personalized link to online course evaluations:

- via email you received from c-evals@usc.edu
- or log into blackboard.usc.edu

Review: Fraction example

- assume we want to use separate compilation to compile the pieces of the following program.

`Fraction.h`
header file for a Fraction class

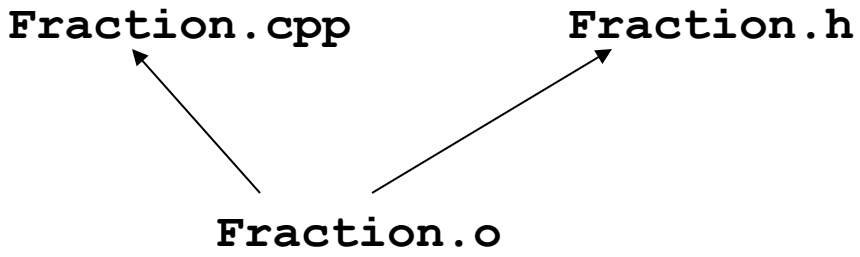
`Fraction.cpp`
implementation file for a Fraction class
has **`#include "Fraction.h"`** line

`testFract.cpp`
a program that uses Fractions
has **`#include "Fraction.h"`** line

What is make?

- make (and gmake)
  - remembers long compile commands
  - keeps track of dependencies between source files
  - automatically recreates files that are out of date
- For example:
  - if we compile our **`testFract`** program
  - later, change **`testFract.cpp`**
  - make figures out that we need to recreate **`testFract.o`** and **`testFract`** but not recreate **`Fraction.o`**
- necessitates writing a makefile to specify the commands and dependencies between files.

Make Dependencies



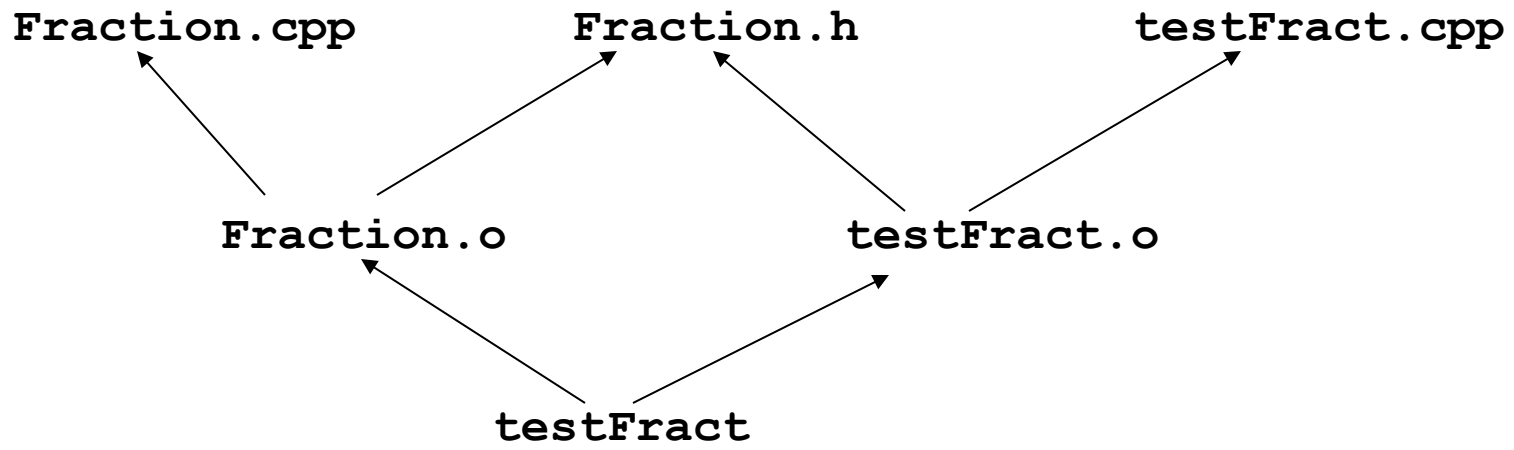
example of dependencies between files

- The makefile encodes the dependencies between source files, and object and executable files
- It also has the "action" to (re)create the dependent file
- General form of a make rule:

```
targetfile:   file(s) it depends on
<tab>action(s) to recreate targetfile
      from files it depends on
```

- Warning: put exactly one "tab" char before each action.

Makefile: version I



- Here's a makefile for the dependencies given above:

```
testFract: Fraction.o testFract.o                                    (1)
    g++ -Wall -ggdb Fraction.o testFract.o –o testFract

Fraction.o: Fraction.cpp Fraction.h                                  (2)
    g++ -Wall -ggdb -c Fraction.cpp

testFract.o: testFract.cpp Fraction.h                                (3)
    g++ -Wall -ggdb -c testFract.cpp
```

Using make with the example makefile

- Example calls to gmake:

```
gmake testFract
gmake Fraction.o
gmake
```

- What happens the first time we do **`gmake testFract`**

- Suppose, we then change **`Fraction.cpp`** only

- Suppose we then change **`Fraction.h`** only

- Suppose we remove **`testFract.o`**

More about what make knows about

- Not smart:
  - *does* know about last change date of files and whether they exist or not
  - *does not* look in the file to see what is #include'd to figure out dependencies
    - you have to tell it this in your Makefile
    - other make-like utilities do figure this out for you (e.g., in IDE's such as VC++)
  - however, there are some built-in rules that make uses to do common things (e.g., <*name*>.*cpp* ⟶ <*name*>.o) (more about that soon)

More about make dependency graphs

- a single makefile can have more than one "graph"
- the order of rules is not significant, unless you call gmake with no args (then it uses the first rule found).
- E.g., we'll use makefiles that can be used to create one or more executables and to submit our program:

```
# Makefile for an assgt ("#" lines are comments)

getfiles:
. . .
grades: . . .
. . .
concord: . . .
. . .
submit:
    submit -user csci455 -tag pa5 Table.cpp . . .
```

make variables

- Variables:

```
VAR = value          define a var
$(VAR)               use a var
```

- We use upper case for variable names by convention
- Some examples of variable definitions:

```
CXX = g++
CXXFLAGS = -Wall -ggdb
OBJS = Fraction.o testFract.o
```

- An example of using variables:

```
testFract: $(OBJS)
        $(CXX) $(CXXFLAGS) $(OBJS) -o $@
```

- The $@ is a special variable that means the name of the current target (e.g., **`testFract`**).

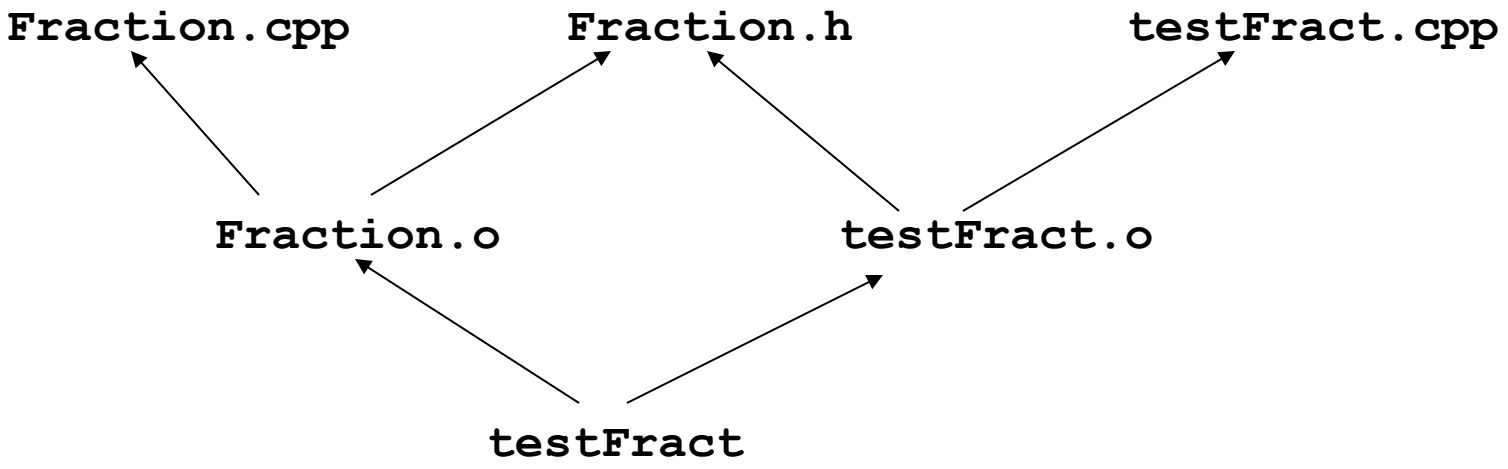
Built-in rules

- for example, gmake knows about
  - <*name*>.*cpp* ← <*name*>.o dependency
  - and associated compile command to use
  - its compile command uses variables **CXX** and **CXXFLAGS** variables (for gmake -- names and default values may be different for make)
  - if we don't define these, it may use a different compiler, and no flags
  - this is not the syntax for the default rules, but this gives you an idea what the rule is:

```
anyfile.o: anyfile.cpp
        $(CXX) $(CXXFLAGS) -c anyfile.cpp
```

- this means we can leave those rules out of our makefile
- but, remember that make doesn't know about #include, so we still need to give those dependencies

A fancier makefile



- Let's write a new makefile for the dependencies given above that uses variables and built-in rules

Bigger make example

- **`calc.cpp`** A calculator program that uses a stack of fractions.
- **`Stack.h .cpp`** – class to store a stack of Fractions
- **`Fraction.h .cpp`** – Fraction class
- Files:

```
// calc.cpp
#incl. "Fr.h"
#incl. "St.h"
int main() {
  . . .
};
```

```
// Stack.h
#incl. "Fr.h"
class Stack
{
  . . .
};
```

```
// Stack.cpp
#incl. "St.h"
// Stack
// mbr funcs
 . . .
```

```
// Fraction.h
class Fraction
{
  . . .
};
```

```
// Fraction.cpp
#incl. "Fr.h"
// Fraction
// mbr funcs
 . . .
```

Bigger make example

- A calculator program that uses a stack of fractions:

```
CXX = g++
CXXFLAGS = -ggdb -Wall
OBJS = calc.o Stack.o Fraction.o

calc: $(OBJS)
        $(CXX) $(CXXFLAGS) $(OBJS) -o $@
calc.o: Stack.h Fraction.h
Stack.o: Stack.h Fraction.h
Fraction.o: Fraction.h
```

Does it have to be called Makefile?

- by default gmake looks for a file called **`Makefile`** or **`makefile`** (the latter takes priority).
- if you want multiple makefiles in the same directory, you can use different names and the **`-f`** option.

```
gmake -f makecalc calc
gmake -f maketestFract testFract
```

- if there is *no* makefile, gmake will just try to use its built-in rules:

```
% gmake calc
g++ calc.cpp -o calc
```

- if there is a makefile, but *no* rule with the name you gave it also uses built-in rules:

```
% gmake testio
g++ -ggdb -Wall testio.cpp -o testio
```