Separate compilation and **`make`**

- why
- how to use separately linked code
- what goes in header files
- Example: separately compiled **`Fraction`** class
- how to compile and link
- Using **`#ifndef`**
- make
  - dependencies
  - Makefile
  - rules
  - using make
  - variables

code for example available on Vocareum under **`04-27`**

Announcements

- Review session [also posted on week-by-week schedule]:
  Mon. 5/10 2 – 4pm PDT (also recorded)
- Final exam:
  Tue. 5/11 from 8 – 10am PDT (alternate time TBA)
- Sample final exams available now
- Soon: Finals' week office hours on piazza (updated office hours post)
- Course evals:
  - available now, last day is Tue 5/4
  - look for email from c-evals@usc.edu or log into blackboard.usc.edu
  - will provide class time on Thur.
- Lab 15 this week on C strings (Thurs. lecture)

Separate compilation

- What it is?
  - put different parts of the program in different files
  - compile each file separately into object files
  - *link* all the pieces together into one executable

Why separate compilation

- separating program files increases modularity
  - good for team projects
- separate compilation saves time:
  only recompile parts that changed since the last compile
  - on large projects, one compile can be slow
- also, can save space: some systems have *dynamically linked libraries*: only one copy of the library in main memory shared by multiple processes.

Using a built-in library

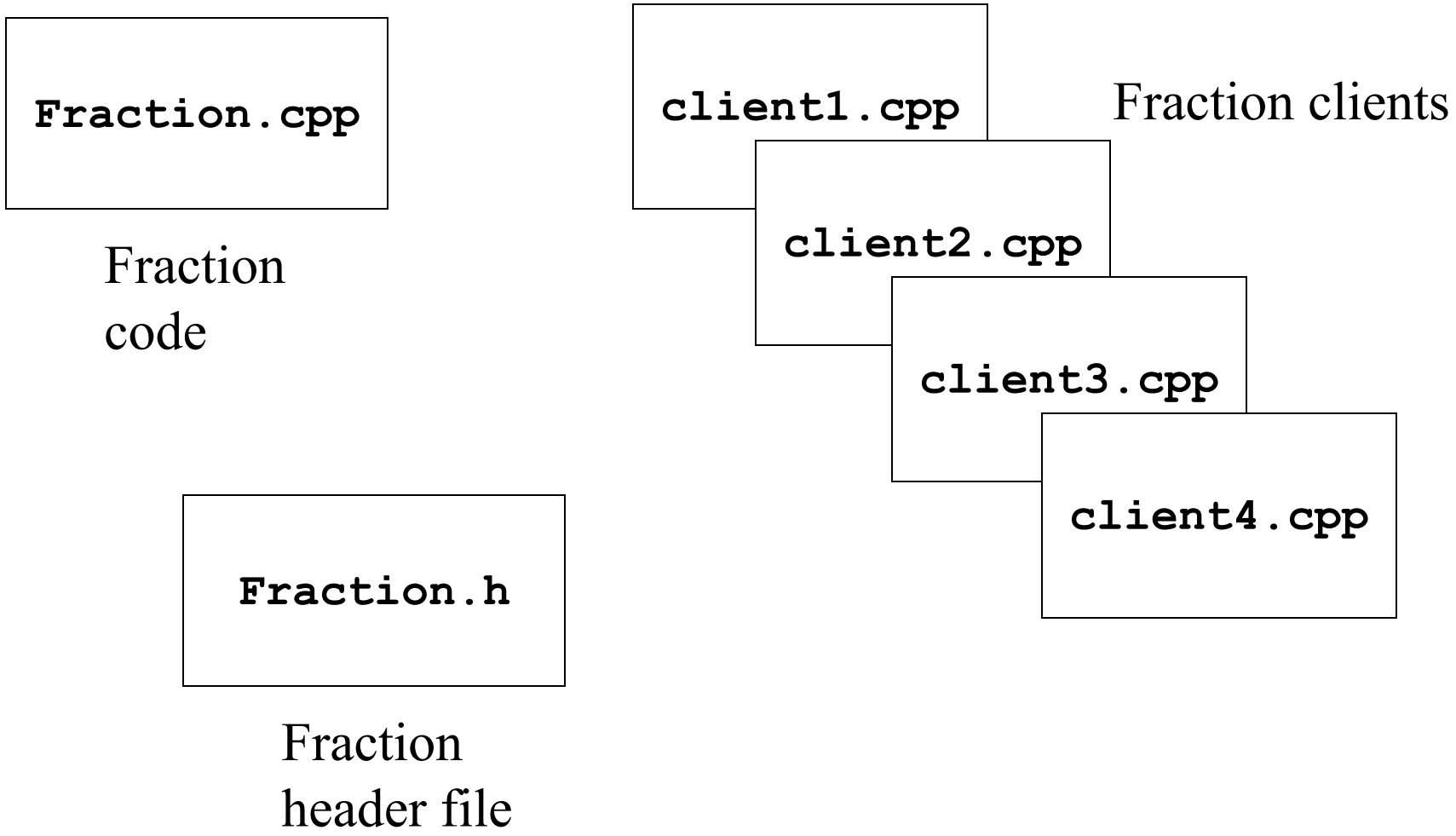
- When using C or C++ standard library, code is already compiled; gets linked with our code automatically.
- We just have to include the necessary header file(s)
- E.g.: **`myprog.cpp`** uses the C math library:
  - **`myprog.cpp`** needs prototype declarations from header file:
    `#include <cmath>`
  - Math library code will automatically get linked into executable (no special compile option necessary)
    `g++ -Wall -ggdb myprog.cpp`

Using a third-party library

- For other libraries we need a special compile command to get the library code linked in to executable.
- E.g.: using the X11 library (Unix GUI programs) :
  - Need prototype declarations from header file. For example: **`#include <X11/Xlib.h>`**
  - Compile command says where library is (part in bold):

```
g++ -Wall -ggdb myGUIProg.cpp -L /usr/X11R6/lib -lX11
```

Ex: compiling Fraction class separately



What goes in header files

- any shared information between modules
- Usually that means declarations, but not function definitions.
- Some things that go in there:
  - class definitions
  - Non-method function prototypes
  - definitions of global constants
  - #define (for C – don't use much in C++)
  - #include for declarations from other modules needed by header file itself
  - typedefs
  - global variable declarations (extern)

    [globals are bad programming practice]

Fraction example

- Fraction with one example client: **`testFract.cpp`**
- Here are the files and what they will contain:
  - **`Fraction.h`**
    has Fraction class definition and associated function prototypes
    (e.g., Fraction arithmetic ops are non-methods)
  - **`Fraction.cpp`**
    has **`#include "Fraction.h"`**
    has Fraction method definitions and associated function definitions
  - **`testFract.cpp`**
    has #include "Fraction.h"
    has main program that uses Fractions

Compiling the Fraction program

1. To create an object file (piece of compiled program) use **`-c`** option

(1) **`g++ -Wall -ggdb -c Fraction.cpp`**

creates ***`Fraction.o`***

(2) **`g++ -Wall -ggdb -c testFract.cpp`**

creates ***`testFract.o`***

2. To create the executable (complete program) from the pieces of compiled code (i.e., to *link* the program)

(3) **`g++ -Wall -ggdb testFract.o Fraction.o`**

creates ***`a.out`***

What compiler needs to compile a piece of a program

- Want to create an object file for **`foo.cpp`**
- **`foo.cpp`** contents:

```
int main() {
   func1(3, 7);
   func2(12);
   Bar b;
   b.method();
   return 0;
}
```

need to #include header file(s) with these things:

need function prototypes

need class definition

- Do not need, *definitions* for called (member) functions to compile **`foo.cpp`** module

Linking a program

- The link step (create a complete executable)
  - looks for **`main`**
  - looks for function definitions for functions we call
- If any of the functions are not found we get link errors. E.g., if we try to compile just one program file into the executable (note no **`-c`** below):

```
g++ -Wall -ggdb testFract.cpp
```

- Or if we do

```
g++ -Wall -ggdb Fraction.cpp
```

Suppose we change the program

- Only have to recompile parts that changed.
- Suppose we change only **`Fraction.cpp`**?

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

- What if a new program **`foo.cpp`** uses Fractions, what do we need to do to compile it?

Not separate compilation

- The following is *not* a separately compiled program...
- **`testFract.cpp`** contains…

```
#include "Fraction.h"
#include "Fraction.cpp"
int main() {
   Fraction a;
   . . .
}
```

- can make executable with

```
g++ -ggdb -Wall testFract.cpp
```

- any time Fraction.cpp changes, we need to recompile testFract code

What's `#ifndef`?

- C++ compilers don't like it if a class is defined more than once.
- For other things that go in header files it's fine to have multiple instances. e.g., function prototypes
- With #include, can get multiple copies of a class definition. To make sure the C++ compiler only sees one:

```
#ifndef FRACTION_H
#define FRACTION_H
class Fraction {
  . . .
};
#endif
```

POLL: separate compilation

- Refer to **`studentProg.cpp`** linked on week-by-week schedule on 4/13 lecture

---

Asynchronous participation: Link to C++ separate compilation poll

Make utility

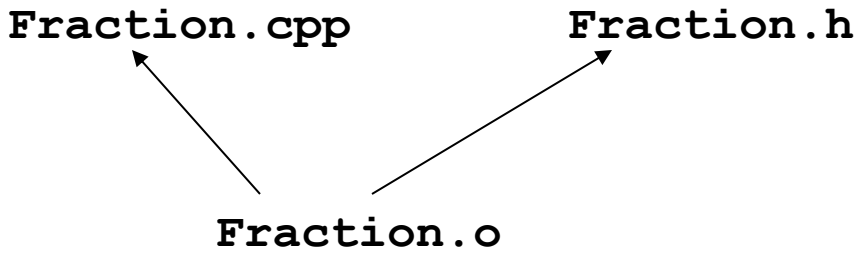
- What files changed since I last compiled?
- What are the exact compile commands to use?

  → tedious to remember and type
- . . . if you have a program composed of 20 or more source files it becomes impossible.

What is make?

- make
  - 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



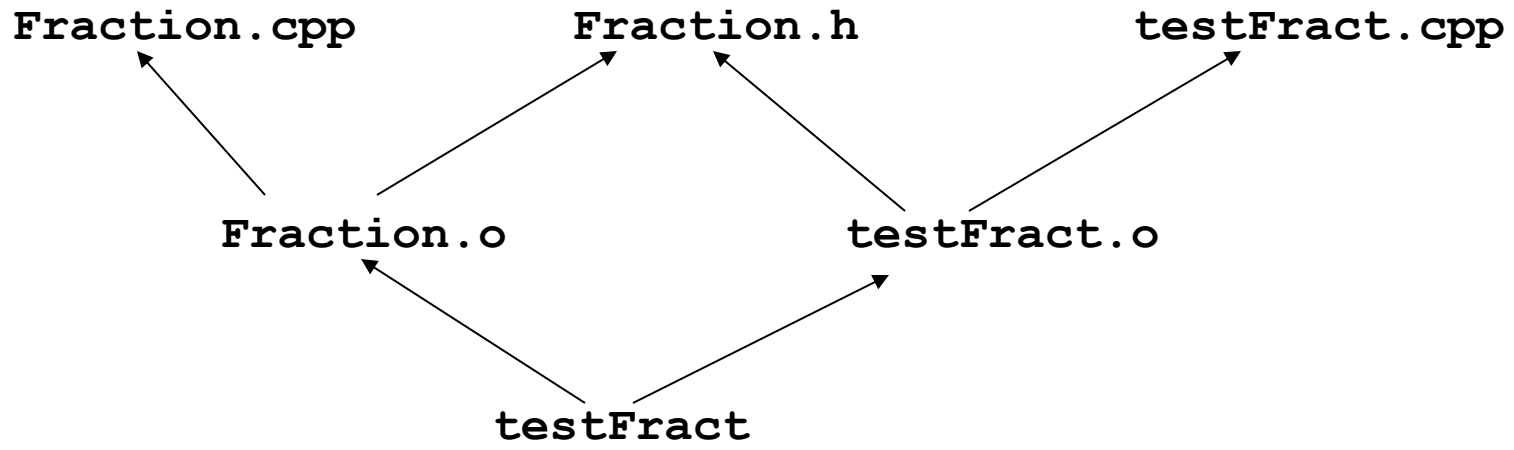
example of dependencies between files

- 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 make:

```
make testFract
make Fraction.o
make
```

- What happens the first time we do **`make 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 build tools 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 make with no args (then it uses the first rule found).
- E.g., A makefile that can be used to create one or more executables and to clean the directory:

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

concord: . . .
. . .
grades: . . .
. . .
pa5list: . . .
. . .
clean:
    rm grades concord grades.o Table.o . . .
```

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`**).

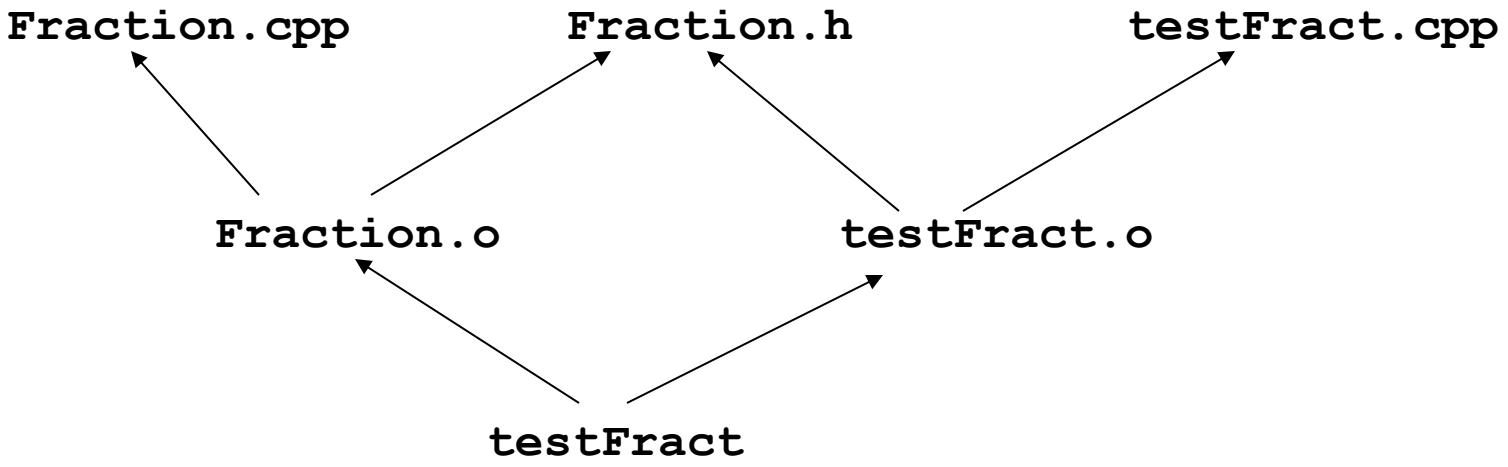
Built-in rules

- for example, make knows about
  - <*name*>.*cpp* ← <*name*>.o dependency
  - and associated compile command to use
  - its compile command uses variables **CXX** and **CXXFLAGS** variables (for make -- 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 make 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.

```
make -f makecalc calc
make -f maketestFract testFract
```

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

```
% make 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:

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