Separate compilation

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

code for example in:
`~csci455/code/04-20`

Announcements

- Review session:
  Tue. 5/9 from 9 – 11am in OHE 132
- Final exam:
  Wed. 5/10 from 8am – 10am in SGM 124
- Sample final exams available now
- Soon: Week-by-week schedule will have final exam week schedule (office hours, review session).
- Course evals:
  - available now, last day is Tue 5/2
  - look for email from c-evals@usc.edu
  - will provide class time next week.

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 users.

Using a built-in library

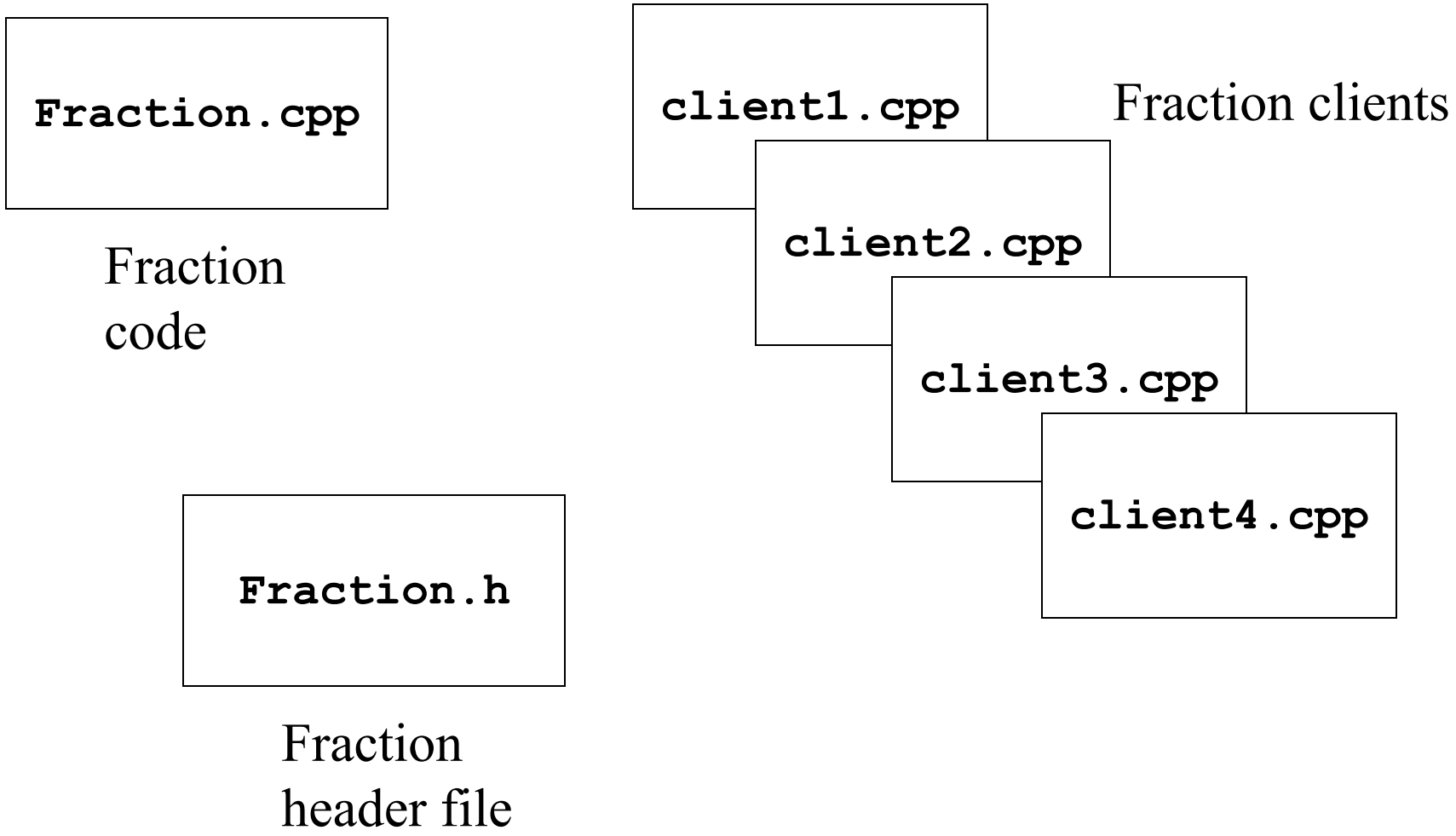
- When using C or C++ standard library, code is already compiled; gets linked with our code automatically.
- We just have to include the necessary header file(s)
- E.g.: **`myprog.cpp`** uses the math library:
  - **`myprog.cpp`** needs prototype declarations from header file:
    `#include <cmath>`
  - 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-member function prototypes
  - definitions of global constants
  - #define (for C -- don't usually use in C++)
  - 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-member functions)
  - **`Fraction.cpp`**
    has **`#include "Fraction.h"`**
    has Fraction member function 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
```

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.
- That's why we have tools like **`make`** (locally, **`gmake`**)
- make manages the compilation process for us.
- but requires writing a correct **`Makefile`**