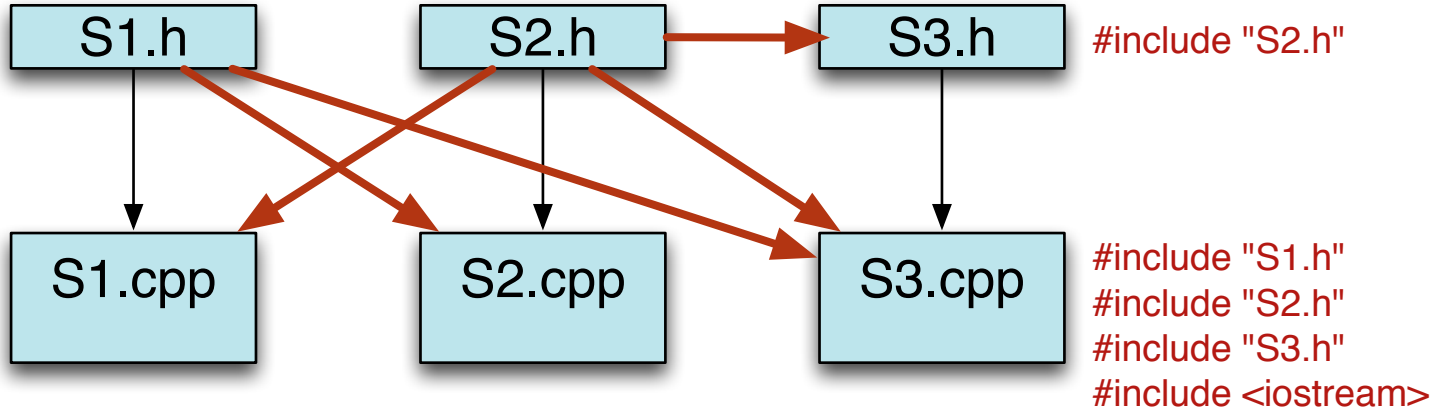
CS 247: Software Engineering Principles

Modules

Readings: Eckel, Vol. 1
Ch. 2 Making and Using Objects: The Process of Language Translation
Ch. 3 The C in C++: Make: Managing separate compilation

Program Decomposition

```
// export local definitions
class B { ... };
int bar ();
extern int k;
```

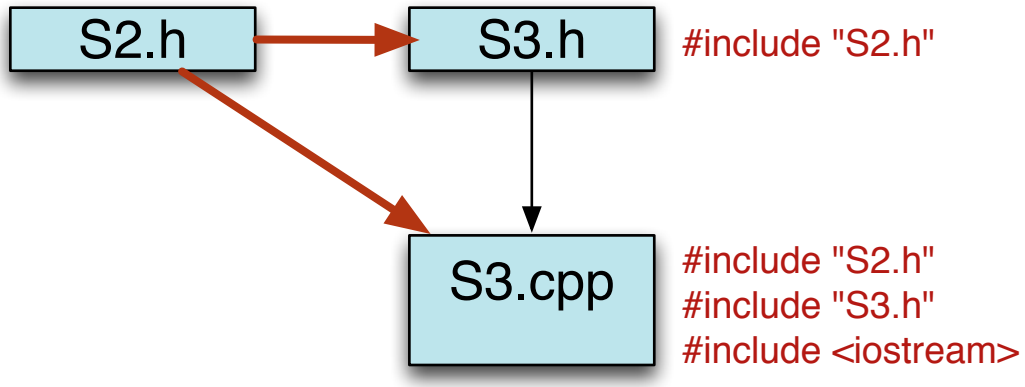


Approach: Put all of the declarations needed by other modules that "use" our module into a Header File

- any module that refers to a nonlocal name from our module must #include our module's header file

- the preprocessor replaces each #include statement with the contents of the specified header

Duplicate Header Inclusions



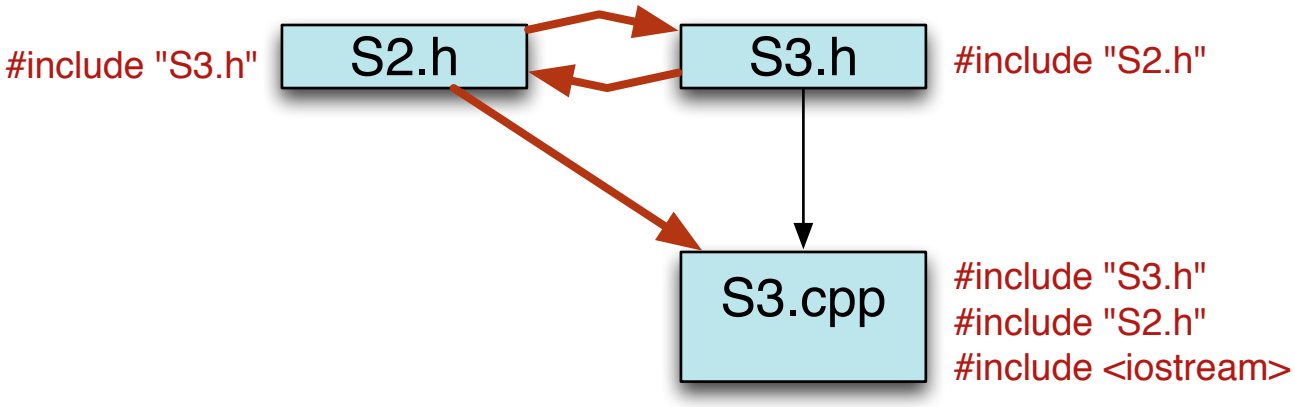
Problem: Compilation of a single source file may include the same header declarations multiple times.

Header Guard

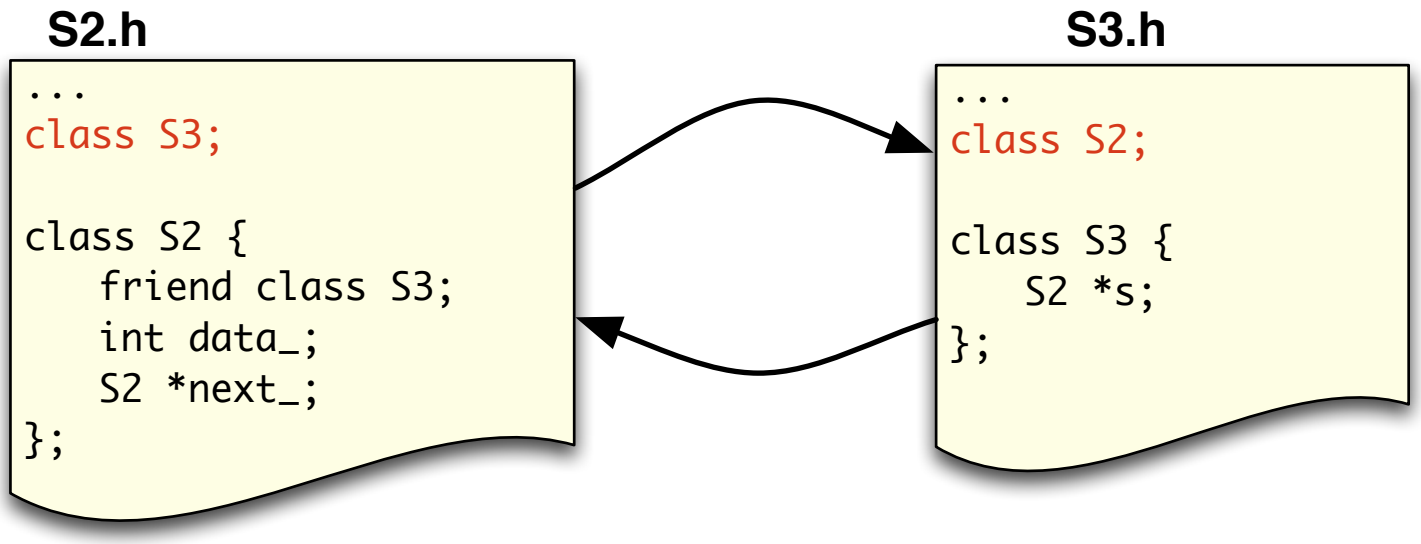
```
#ifndef RATIONAL_H
#define RATIONAL_H

// class Rational declaration and related functions go here
    ...

#endif
```

Circular Dependencies

A circular dependency occurs when two or more header files depend on each other.



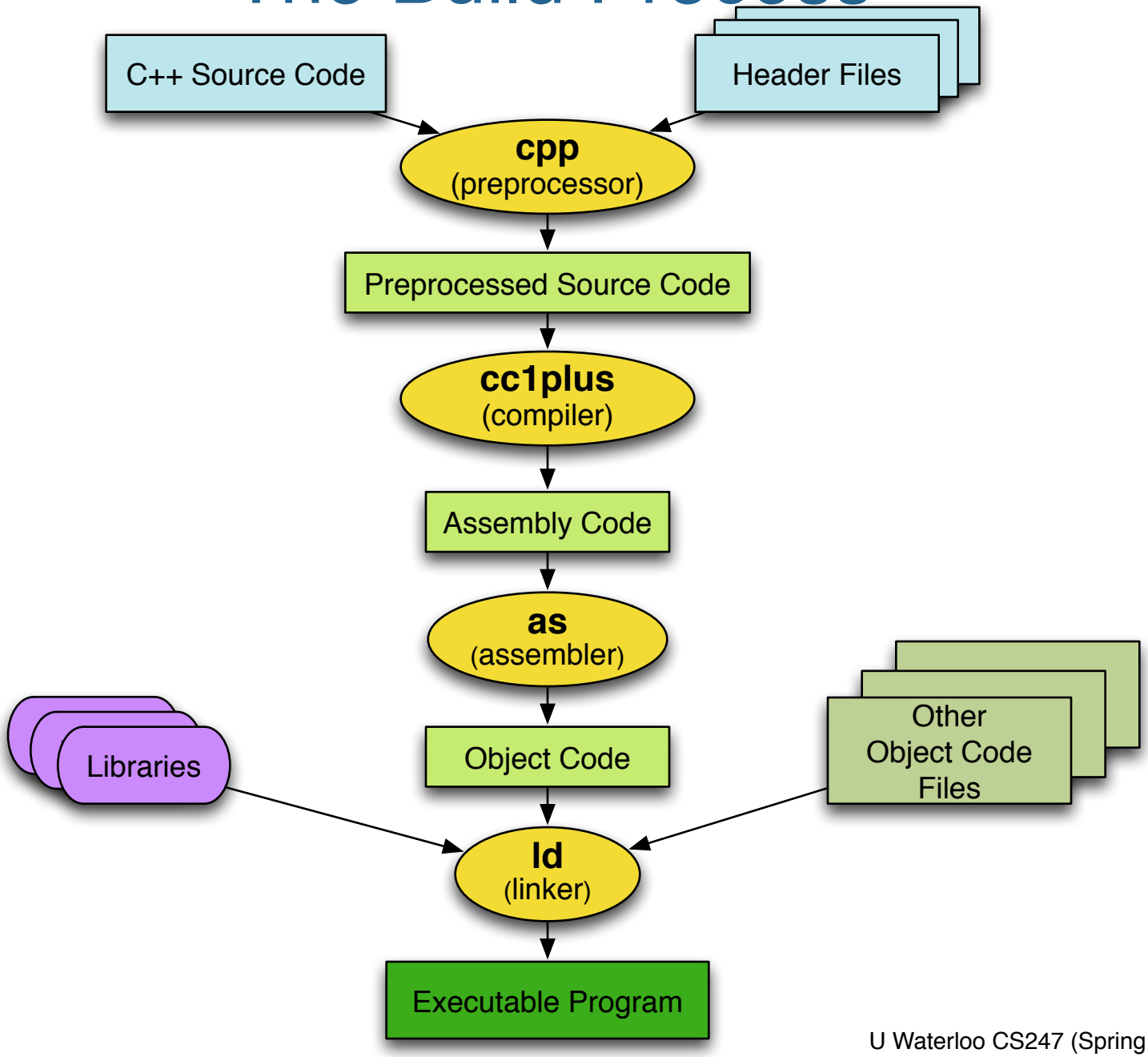
Forward Declarations



A forward declaration notifies compiler that data or function will be declared/defined in the future.

Use forward declarations to break circular header dependencies

The Build Process



Separate Compilation

Because linking is a separate process, can rebuild the program by

- recompiling only the files that have changed
- linking together the object files

```
g++ -c main.cpp
g++ -c ADT2.cpp
g++ -c ADT1.cpp

g++ ADT1.o ADT2.o main.o -o exec       # linking
```

Compilation Dependencies

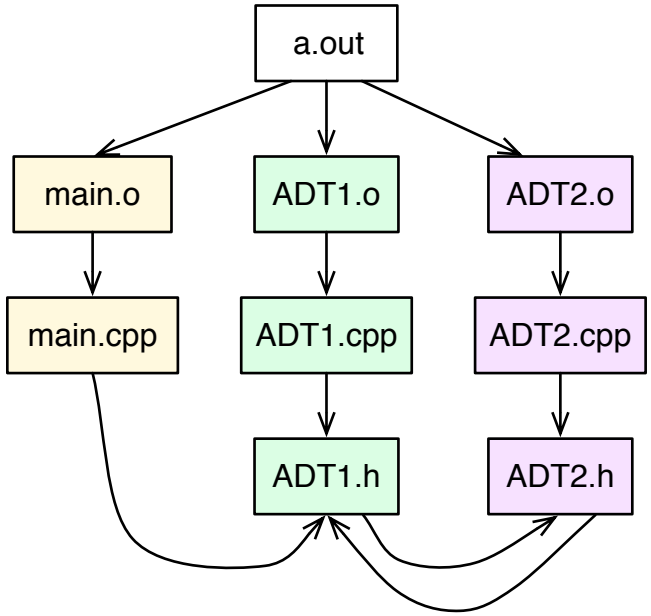
If a file F changes, need to recompile F *and all files that depend on F*

Example:

main.cpp: **#include** "ADT1.h"

ADT1.h: **#include** "ADT2.h"
ADT1.cpp: **#include** "ADT1.h"

ADT2.h: **class** ADT1
ADT2.cpp: **#include** "ADT2.h"



- the executable depends on `.o` files
- the `.o` files depend on `.cpp` files
- the `.cpp` files depend on `.h` files

Automated Builds

Goal: Want a fully automated build system that

- is sure to incorporate all updated source files into executable
- is incremental and rebuilds only what has changed
- automatically derives the dependency relationships among files

Make

make is a UNIX command that uses (1) build instructions and file dependencies provided in a Makefile, and (2) file timestamps to decide which files to recompile/rebuild

Example Makefile:

```
# A comment
program.exe: main.o ADT1.o ADT2.o        # dependency graph
    g++ main.o ADT1.o ADT2.o -o program.exe  # build rule
```

tab

Makefile with Macros

Makefile more reusable if rules can be defined in terms of macros that are set at the top of the file.

```
CXX = g++                       # = specifies a new macro
CXXFLAGS = -g -Wall
OBJECTS = main.o ADT1.o ADT2.o
EXEC = program.exe

${EXEC} : ${OBJECTS}            # ${ } expand the macro
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

main.o : main.cpp stack.h
    ${CXX} ${CXXFLAGS} -c main.cpp
```

Implicit Rules

```
CXX = g++                       # variables and initialization
CXXFLAGS = -g -Wall
OBJECTS = main.o stack.o node.o
EXEC = program

${EXEC} : ${OBJECTS}            # default target
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

# gmake knows how to build .o files; just list dependencies
main.o : main.cpp stack.h
stack.o : stack.cpp stack.h node.h
node.o : node.cpp node.h stack.h

clean :                         # second target
    rm -rf ${OBJECTS} ${EXEC}
```

gmake has implicit rules for processing files with specific suffixes and when special variable names are used.

Target `clean` removes files that can be rebuilt (to save space)

Automatically Derive Dependencies

```
CXX = g++                       # variables and initialization
CXXFLAGS = -g -Wall -MMD # builds dependency lists in .d files
OBJECTS = main.o stack.o node.o
DEPENDS = ${OBJECTS:.o=.d} # substitute ".o" with ".d"
EXEC = program

${EXEC} : ${OBJECTS}
    ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

clean :
    rm -rf ${DEPENDS} ${OBJECTS} ${EXEC}

-include ${DEPENDS}         # reads the .d files and reruns
                            # dependencies
```

- g++ flag -MMD generates a dependency graph for user source files.