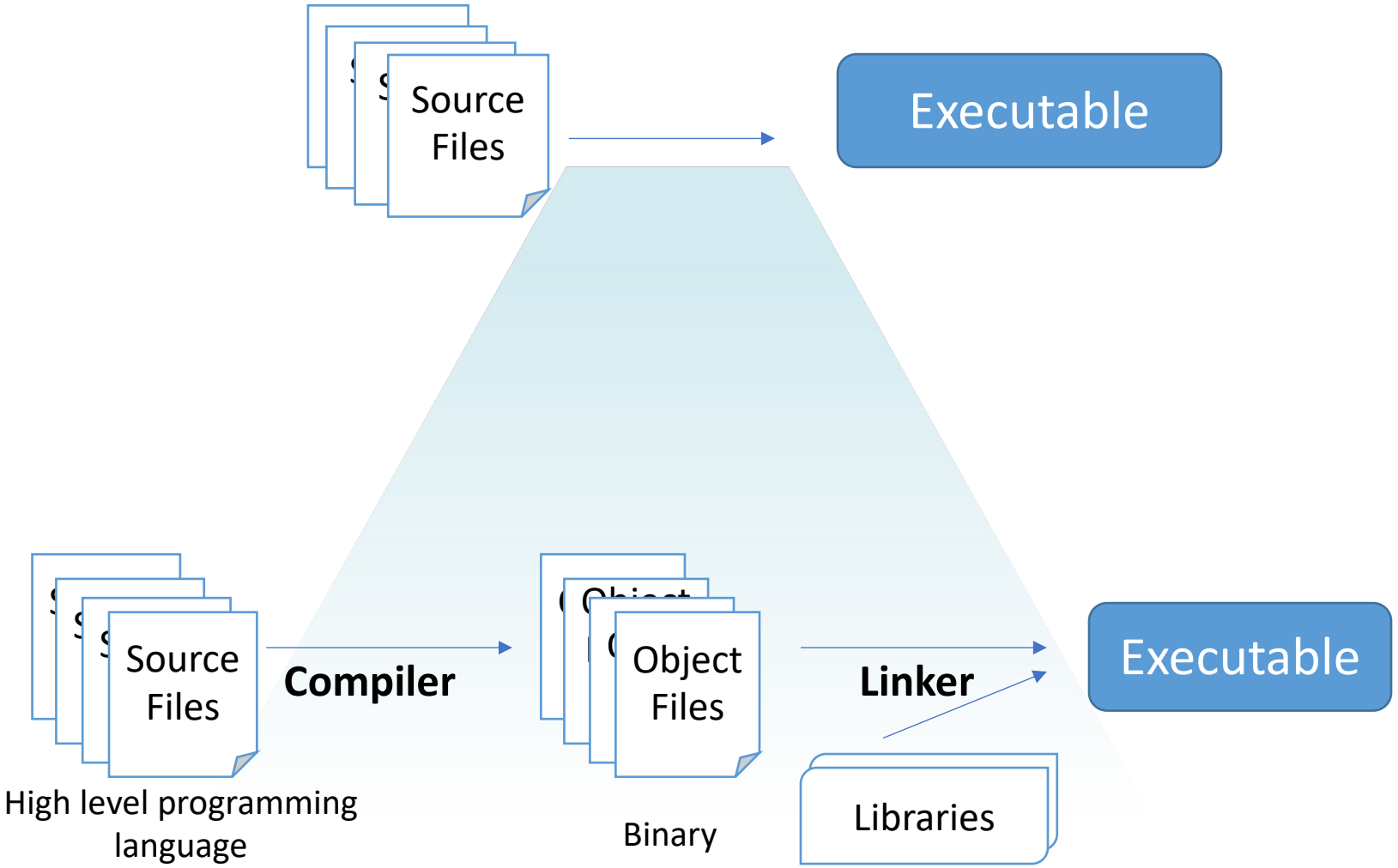
CS 6015: Software Engineering

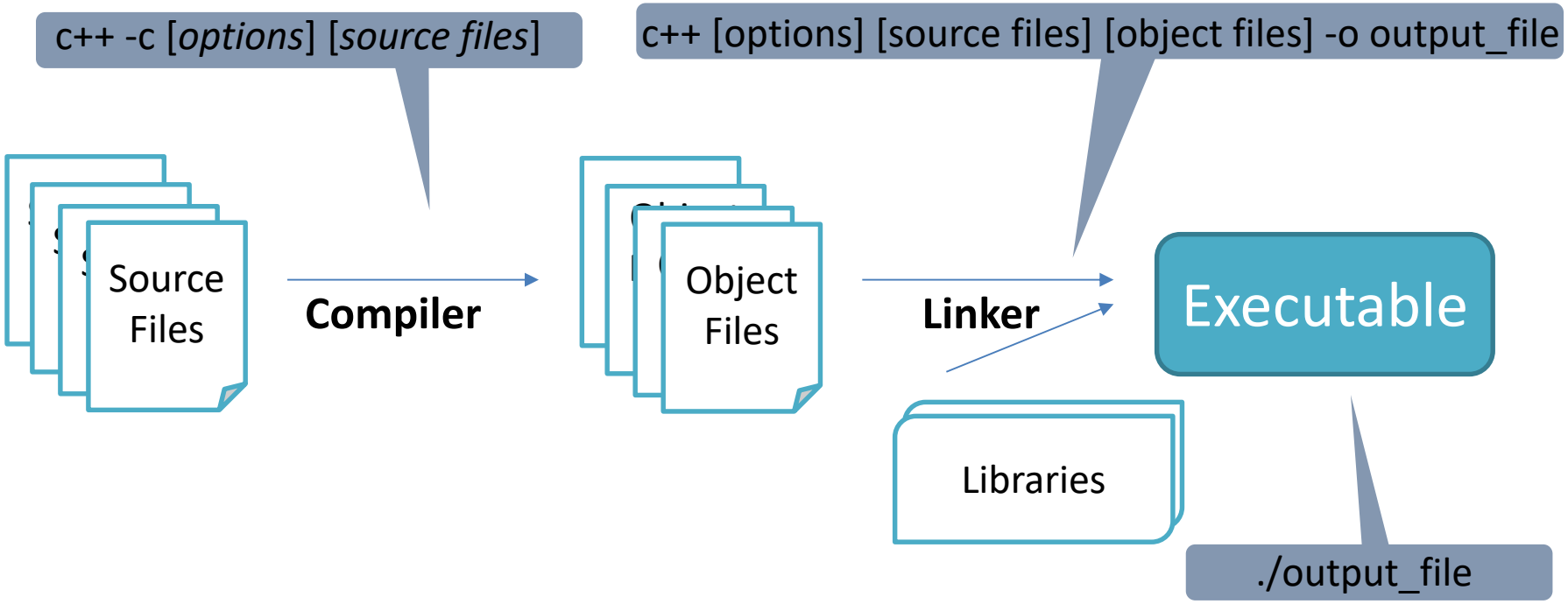
Spring 2024

Lecture 1: Makefiles

Building programs: Recall



Building programs: Recall



Working with large projects

- Large projects have many source files
  - Each needs to be compiled into a .o files
  - All .o files need to be linked into an executable

- What needs to change if I edit one source file?
  - Corresponding .o file needs to be recompiled
  - Executable needs to be relinked
  - What about other .o files?
  - What if I edit a .h file?

Can tools help?

- Programmers don’t want to manually run `c++ -c whatever.cpp` and relink
  - Easy to forget one or both of these steps, get the files wrong, etc
  - How do we avoid doing complex steps manually? Have a program do it for us
- What does a tool need to do?
  - Figure out which inputs changed
  - Rerun required commands when their input files change
- The tool we’ll use: Make
  - Write a “recipe” of the steps to build our program called a Makefile
  - Make will rerun steps when the input files change, will reuse existing .o files if the .cpp files didn’t change

Automatic build

- Software tools for automatic build:
  - ***gmake*** (GNU Make): build system for C/C++ code on Linux, Mac OS
  - Microsoft ***nmake***: build system for C/C++ code on Windows
  - Ant: build system for Java code on Linux, Mac OS, Windows

- Makefiles
  - Files used by ***gmake*** to build programs
  - Include rules for compiling and linking files into executables
  - Include rules for running various commands (cleaning files, generating doc,..)
  - Uses “file modification timestamp” to decide whether to execute the command or not

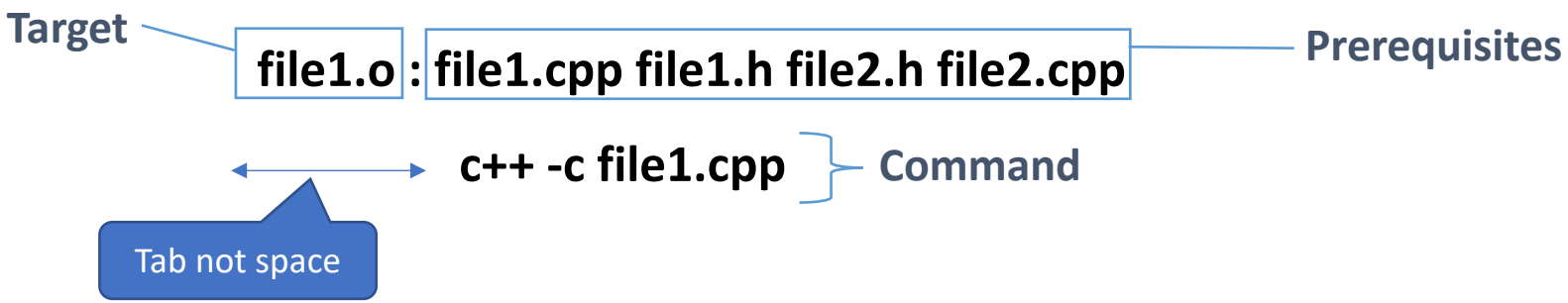
Makefiles

- General structure:
    1. Header:
        - Comments that describe the makefile
        - What the makefile builds
        - Comments start with a # character

```
# Makefile - builds CS 6015 Project
#
# To build a target, enter:
# make <target>
#
# Targets:
# all -buildseverything
# clean - deletes all .o, .a, binaries, etc.
```

Makefiles

- General structure:
  2. **Rules**: describe dependencies and has command to execute.



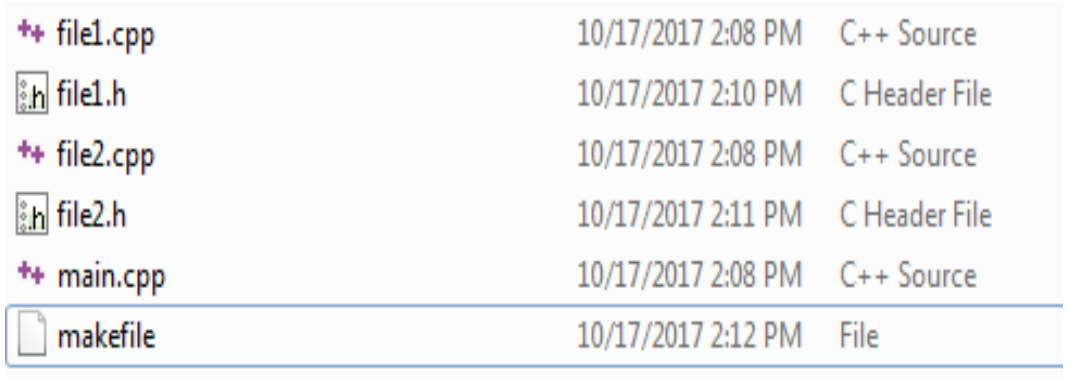
  - If the target or any of the prerequisites has changed, then the rule commands are executed again
    - Based on the timestamp of the files and the last timestamp at which the rule was called

Makefiles: Target

- Preferable common target names:
  - all: to build everything
  - doc: to build documentation
  - clean: to delete derived files such as .o files

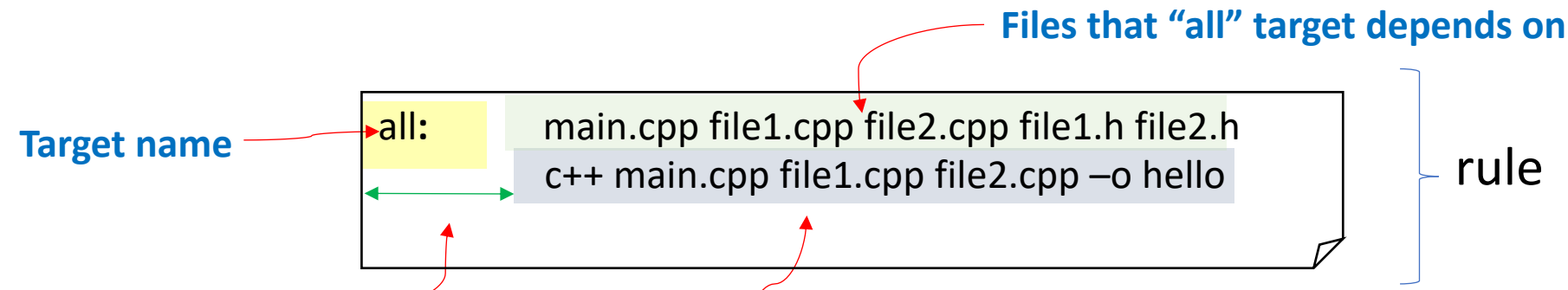
- As in classes, methods and variables, choose a sensible name

Makefiles: Example



makefile

Text file with no extension that includes rules



Required tab

Allows detecting that next is a command

Command to execute

Only runs if any of the dependencies have changes since last time you ran make

Makefiles: Example

```
#Targets:
#clean deletes program, and all .o and .out files
all: program //Optional – just as a default rule name
program: file1.o file2.o
        c++ file1.o file2.o -o program
file1.o: file1.cpp file1.h
        c++ -c file1.cpp
file2.o: file2.cpp file2.h
        c++ -c file2.cpp
clean:
        rm -f *.o *.out program
```

Makefiles: run make

- Use the following commands
  make <target_name>

```
$ make
$ make clean
```

- *Makefile:* default file name

- Can use different file names, but include the flag -f:
  make –f <make_file_name> <target_name>

Makefiles: Macros

- Macros: variables used in makefile
  - Make it easier when projects get larger
  - Can be a value for file, path, or flags

```
CXX=c++
CFLAGS= -std=c++11
CCSOURCE= file1.cpp file2.cpp

program: $(CCSOURCE)
        $(CXX) $(CFLAGS) -o program $(CCSOURCE)
```

- Macros with wildcard:
  - CCSOURCE = ${wildcard *.cpp} → gives all .cpp files

Makefiles: automatic variables

- Macros: short-hand for names of targets and prerequisites
  - $@ Target filename of the current rule
  - $^ All the filenames of all the prerequisites, separated by spaces
  - $< First prerequisite filename in the list
  - $? All out of date prerequisites

```
file.o:    file1.cpp file2.cpp
           c++ $^ –o $@
```

```
file.o:    file1.cpp file2.cpp
           c++ file1.cpp file2.cpp –o file.o
```

Makefiles: implicit rules

- Implicit rules: built-in rules in make
  - Used when there is no rule at all or if there is a rule but with no command for .o file

```
file:  file1.o file2.o
        c++ file1.o file2.o –o file
```

NO rules for file1.o and file.o
Implicit rule is used.

```
INCS = sum.h
OBJS = main.o sum.o
CXXFLAGS = --std=c++14 -O2
sum: $(OBJS)
        $(CXX) $(CXXFLAGS) -o sum $(OBJS)

main.o: main.cpp $(INCS)

sum.o: sum.cpp $(INCS)
```

`$(CXX) $(CXXFLAGS) -c $<` is the default command for getting from `.cpp` to `.o` **(implicit rule)**

Makefiles: implicit rules

Or simply

```
INCS = sum.h
OBJS = main.o sum.o
CXXFLAGS = --std=c++14 -O2
sum: $(OBJS)
	$(CXX) $(CXXFLAGS) -o sum $(OBJS)

main.o: $(INCS)

sum.o:  $(INCS)
```

`$(CXX) $(CXXFLAGS) -c $<` is the default command for getting from `.cpp` to `.o` **(implicit rule)**

Makefiles: Example

```
OBJS = main.o sum.o

CXXFLAGS = --std=c++14 -O2

INCS = sum.h

sum: $(OBJS)
	$(CXX) $(CXXFLAGS) -o sum $(OBJS)

main.o: main.cpp $(INCS)
	$(CXX) $(CXXFLAGS) -c main.cpp

sum.o: expr.cpp $(INCS)
	$(CXX) $(CXXFLAGS) -c sum.cpp
```

`CXX` is predefned

Makefiles: Example

```
OBJS = main.o sum.o

CXXFLAGS = --std=c++14 -O2

INCS = sum.h

sum: $(OBJS)
       cd some_dir; echo "Dir";\
        $(CXX) $(CXXFLAGS) -o sum $(OBJS)

main.o: main.cpp $(INCS)
       $(CXX) $(CXXFLAGS) -c main.cpp

sum.o: expr.cpp $(INCS)
       $(CXX) $(CXXFLAGS) -c sum.cpp
```

Multiple commands and changing directories

\ to avoid interpreting the new line as new command

Makefiles: Example

```
OBJS = main.o sum.o

CXXFLAGS = --std=c++14 -O2

Makefile

INCS = sum.h

sum: $(OBJS)
       cd some_dir&& echo "Dir"&&\
        $(CXX) $(CXXFLAGS) -o sum $(OBJS)

main.o: main.cpp $(INCS)
       $(CXX) $(CXXFLAGS) -c main.cpp

sum.o: expr.cpp $(INCS)
       $(CXX) $(CXXFLAGS) -c sum.cpp
```

&& implies that next command only gets executed if the preceding command was successful

Makefiles: Example

```
OBJS = main.o sum.o
CXXFLAGS = --std=c++14 -O2
Makefile
INCS = sum.h
sum: $(OBJS)
        $(CXX) $(CXXFLAGS) -o sum $(OBJS)
main.o: main.cpp $(INCS)
       $(CXX) $(CXXFLAGS) -c main.cpp
sum.o: expr.cpp $(INCS)
       $(CXX) $(CXXFLAGS) -c sum.cpp

.PHONY: clean
clean:
       rm -rf *.o
```

PHONY explicitly tells *make* that these targets are not associated with files

Like a target that is always-out-of-date