



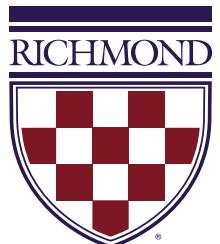
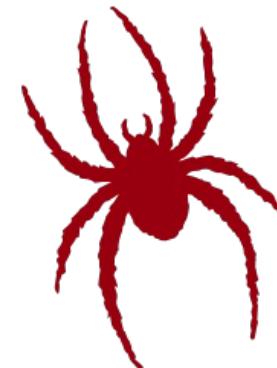
UNIVERSITY OF
RICHMOND

CMSC 240 Lecture 15

CMSC 240 Software Systems Development
Fall 2023

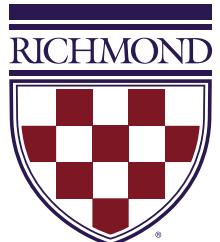
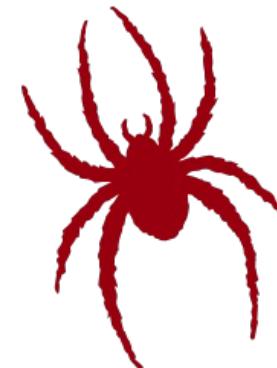
Today – Build Automation

- Compilation pipeline
- Build automation with `make`



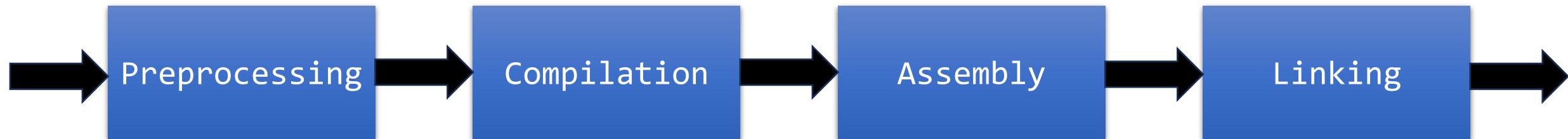
Today – Build Automation

- Compilation pipeline
- Build automation with `make`

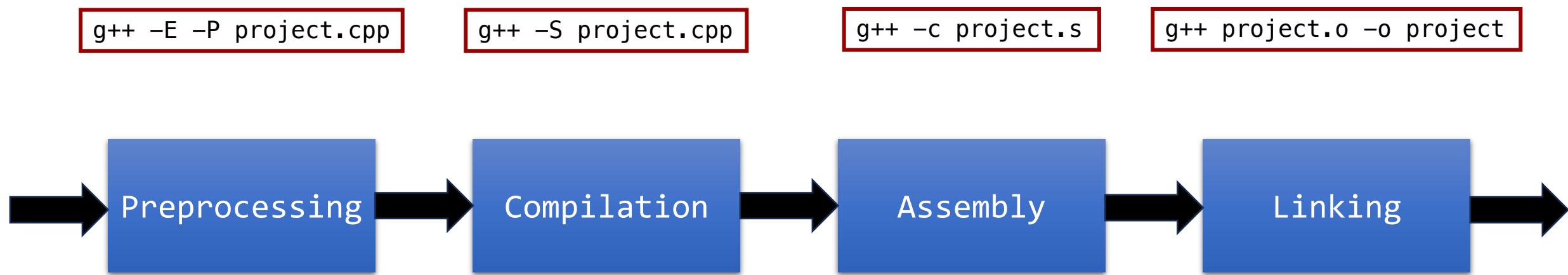


C++ Compilation Pipeline

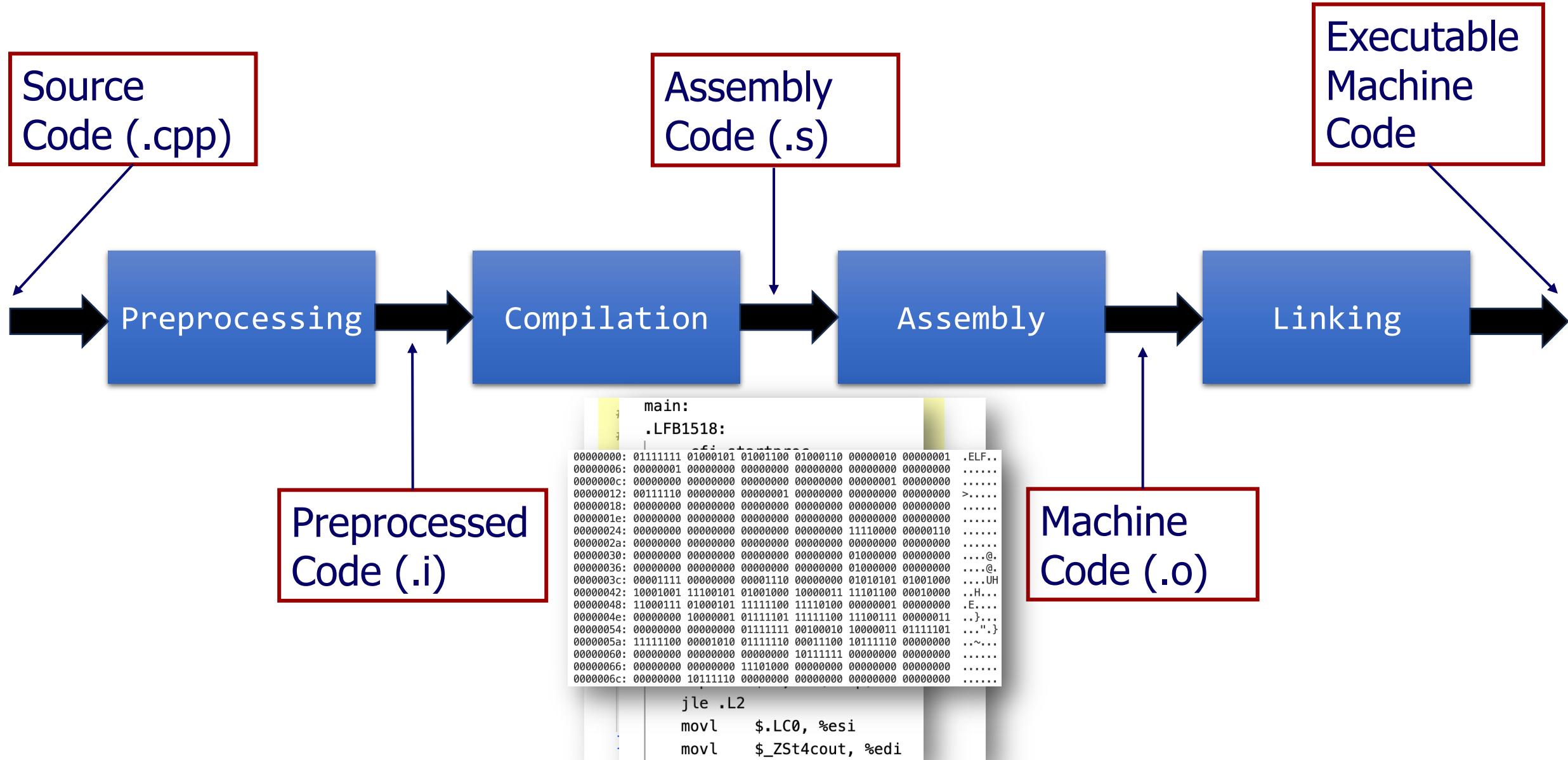
```
g++ project.cpp -o project
```



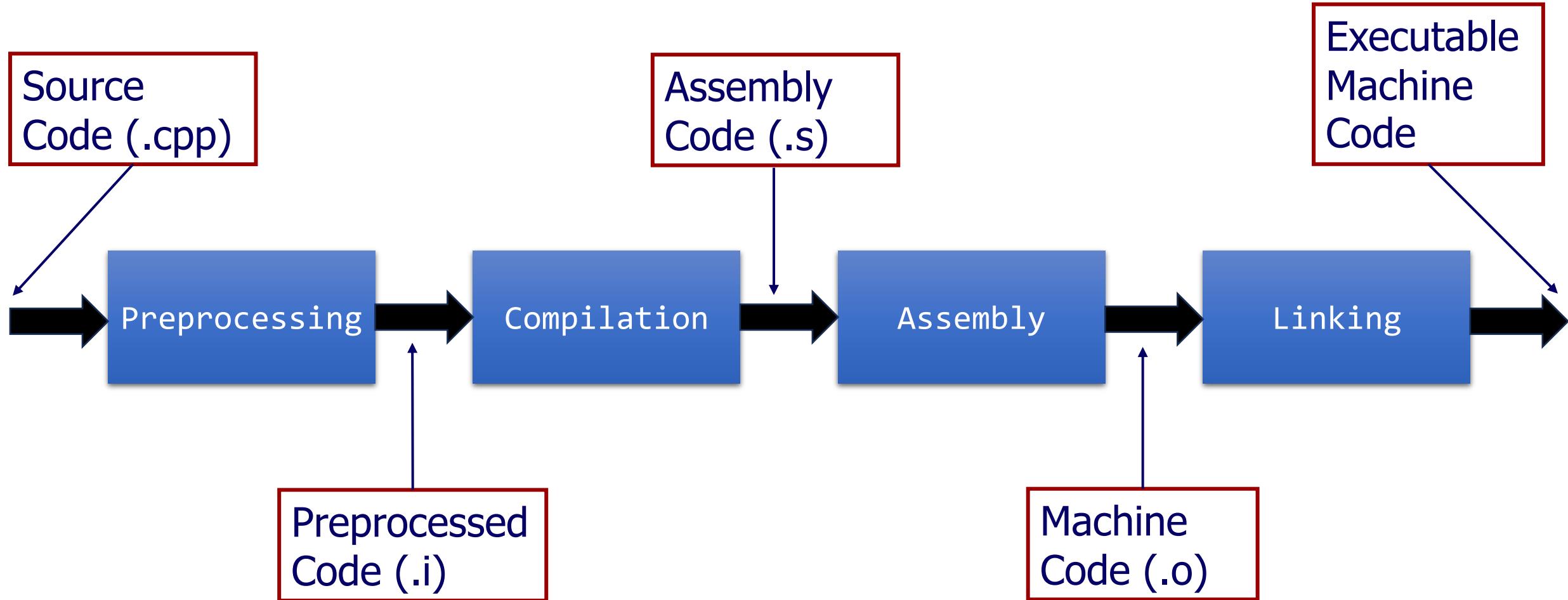
C++ Compilation Pipeline



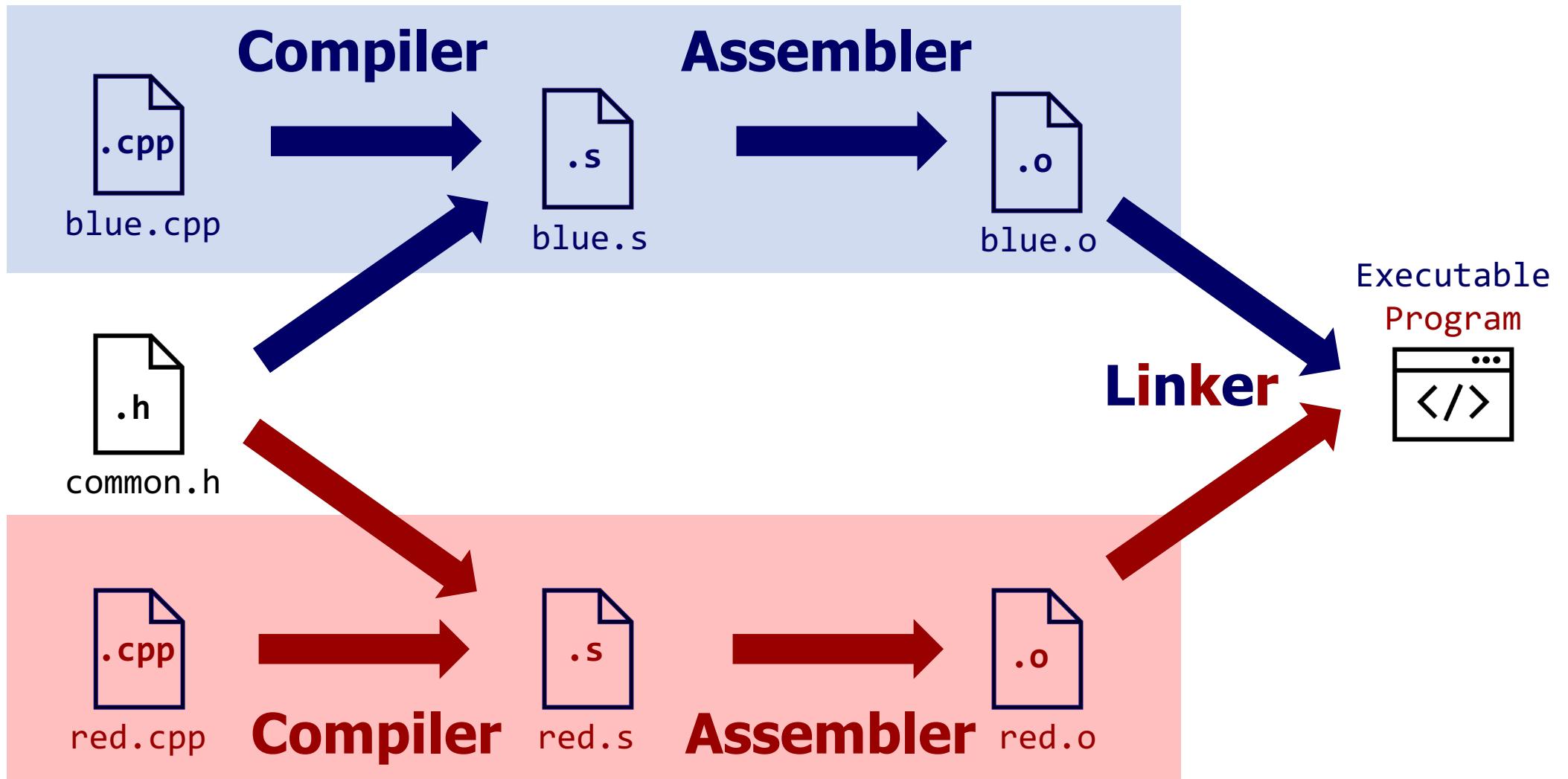
C++ Compilation Pipeline



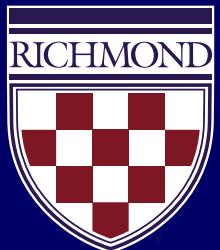
C++ Compilation Pipeline



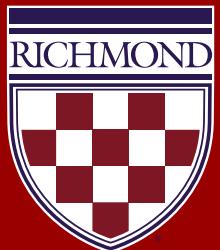
C++ Compilation Pipeline



Ask a question

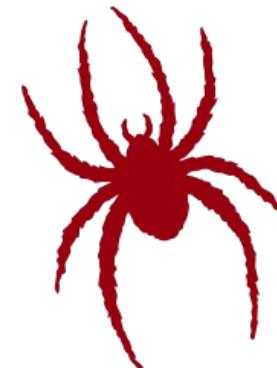


Give it a try!



Today – Build Automation

- ~~Compilation pipeline~~
- Build automation with **make**



make

- The build tool **make** is a classic program for controlling what gets (re)compiled and how
 - Many other such programs exist (e.g. ant, maven, IDE “projects”)
- Two basic ideas of **make**:
 1. Scripts for executing commands
 2. Dependencies for avoiding unnecessary work

make

- Programmers spend a lot of time “**building**”
 - Creating programs from source code
 - Both programs that they write, and other people write
- Programmers like to **automate** repetitive tasks
 - `g++ -Wall functions.cpp calculate.cpp -o calculate`
 - Retype this every time → 😢
 - Use up-arrow or history → 😐
 - Create an alias or bash script → 😊
 - Use **make** → 😃

Creating a “Real” Build Process

- On larger projects, you don’t want to have one big set of commands to run every time anything is changed
- When thinking on how to do things “smarter” consider:
 1. It could be worse: If `g++` didn’t combine steps for you, you’d need to preprocess, compile, and link on your own
 2. Source files could have multiple outputs (e.g. generated docs)
 3. Your source code should be relatively simple for others to build
 4. You don’t want to recompile everything every time you make a change

Recompilation Management

- The theory behind avoiding unnecessary compilation is a dependency graph
- To create a build target **t**, you need sources **s₁, s₂, … , s_n** and a command **c** that uses the sources
 - If **t** is newer than every source (file-modification times), then there is no reason to rebuild it

```
drwxr-x---. 2 dbalash acs 176 Oct 23 10:52 .
drwxr-x---. 5 dbalash acs 103 Oct 23 09:02 ..
-rwxr-x---. 1 dbalash acs 17976 Oct 23 10:52 calculate
-rw-r-----. 1 dbalash acs 271 Oct 23 10:45 calculate.cpp
-rw-r-----. 1 dbalash acs 1480 Oct 23 10:51 calculate.o
-rw-r-----. 1 dbalash acs 617 Oct 23 10:45 functions.cpp
-rw-r-----. 1 dbalash acs 261 Oct 23 10:45 functions.h
-rw-r-----. 1 dbalash acs 1360 Oct 23 10:51 functions.o
```

Sources

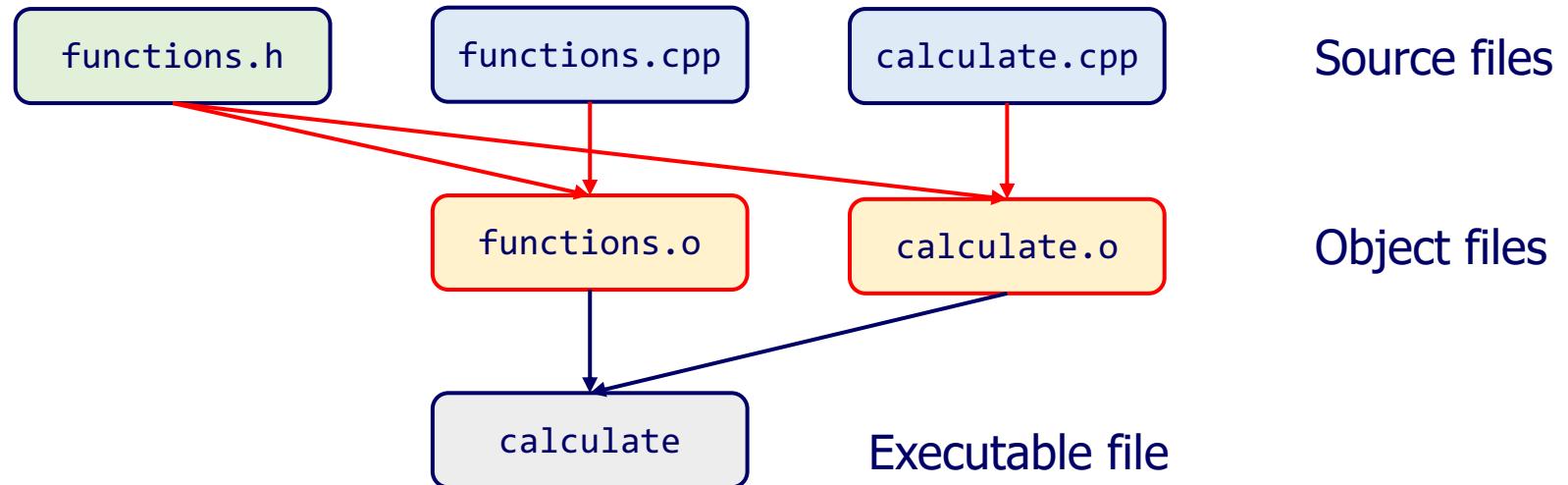
Target

Recompilation Management

- The theory behind avoiding unnecessary compilation is a dependency graph
- To create a build target **t**, you need sources **s₁, s₂, … , s_n** and a command **c** that uses the sources
 - If **t** is newer than every source (file-modification times), then there is no reason to rebuild it
 - Recursive building: if the source **s_i** is itself a build target of some other sources, check to see if it needs to be rebuilt

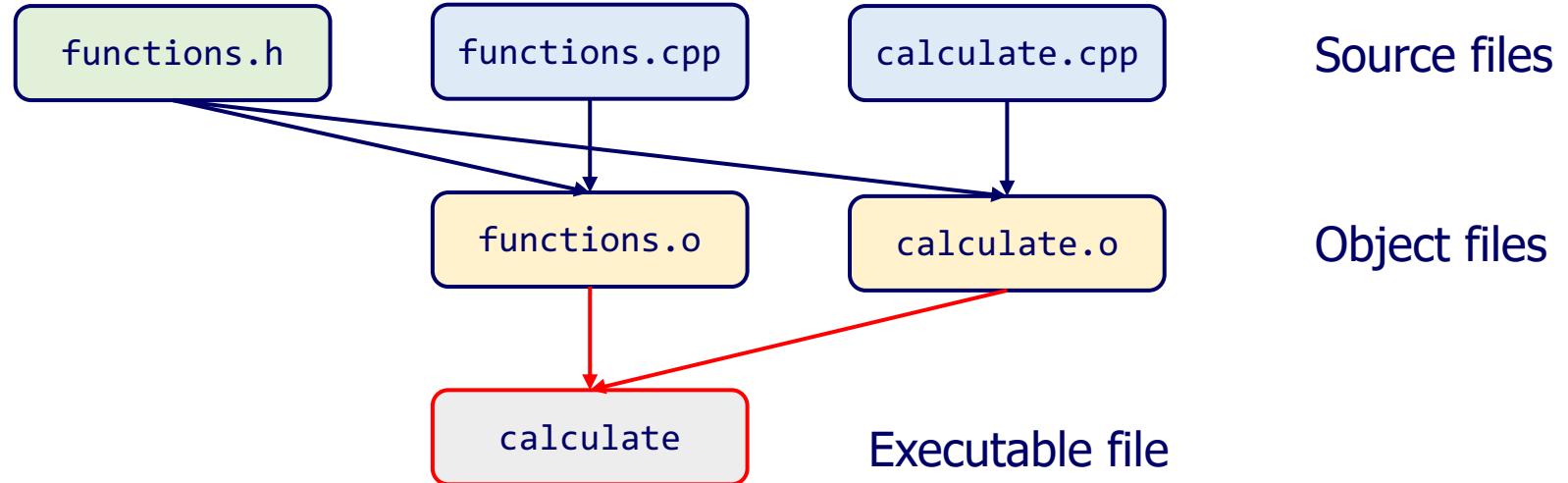
```
drwxr-x---. 2 dbalash acs 176 Oct 23 10:52 .
drwxr-x---. 5 dbalash acs 103 Oct 23 09:02 ..
-rwxr-x---. 1 dbalash acs 17976 Oct 23 10:52 calculate ← Target
-rw-r-----. 1 dbalash acs 271 Oct 23 10:45 calculate.cpp ← Sources
-rw-r-----. 1 dbalash acs 1480 Oct 23 10:51 calculate.o ← Sources
-rw-r-----. 1 dbalash acs 617 Oct 23 10:45 functions.cpp ← Sources
-rw-r-----. 1 dbalash acs 261 Oct 23 10:45 functions.h ← Sources
-rw-r-----. 1 dbalash acs 1360 Oct 23 10:51 functions.o ← Sources
```

Example: C++ Build



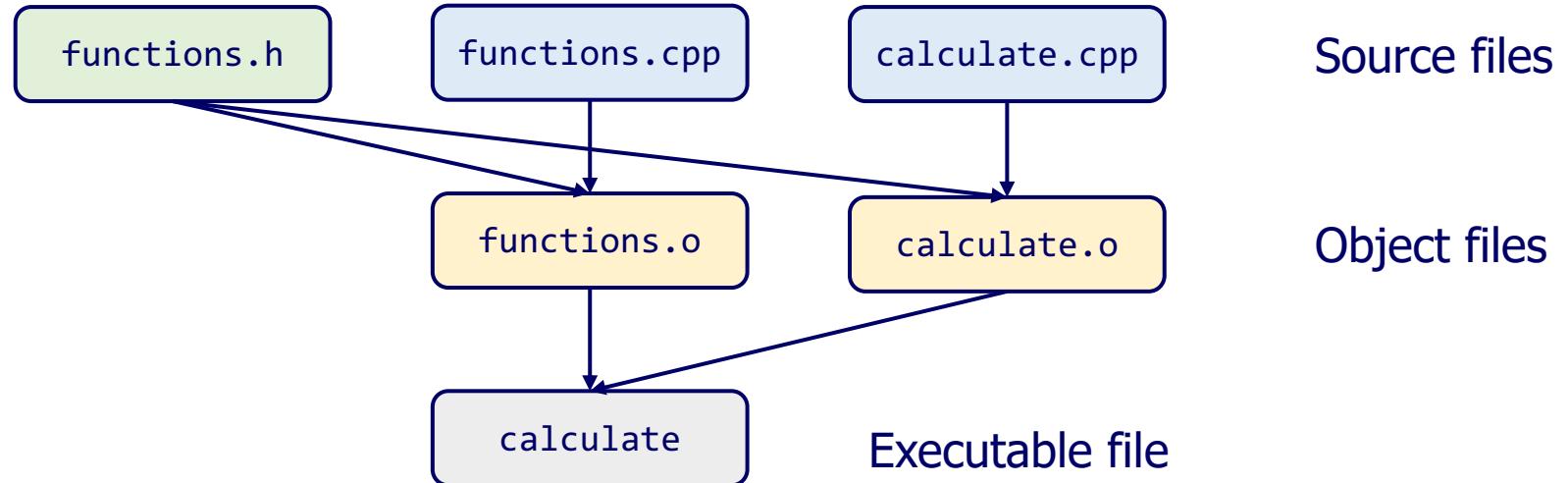
- Compiling a `.cpp` file creates a `.o` file
- The `.o` depends on the `.cpp` and all included files (`.h`)

Example: C++ Build



- Creating an executable
- Linking depends on `.o` files

Example: C++ Build



- If one **.cpp** file changes, we just need to rebuild one **.o** file
- If a **.h** file changes, may need to rebuild more

Using the `make` command

```
$ make -f <MakefileName> target
```

- Defaults:
 - If no `-f` specified, use a file named **Makefile** in current directory
 - If no **target** specified, will use the first one in the makefile

Makefiles

- A makefile contains a bunch of **triples**:

target: sources

← Tab → command

- Colon after target is required
- Command lines must start with a **TAB**, not spaces
- Multiple commands for same target are executed in order
 - Can split commands over multiple lines by ending lines with '\'

- Example:

functions.o: functions.cpp
g++ -c functions.cpp

Makefile Variables

- You can define variables in a makefile:
 - All values are strings of text
 - Variable names are case-sensitive and can't contain `:`, `#`, `=`, or whitespace

- Example:

```
CC = g++
CFLAGS = -Wall -g
OBJFILES = functions.o calculate.o

calculate: $(OBJFILES)
    $(CC) $(CFLAGS) -o calculate $(OBJFILES)
```

- Advantages:

- Easy to change things (especially in multiple commands)
- Can also specify on the command line:
 - (e.g. make calculate CC=clang CFLAGS=-g)

Phony Targets

- “Phony Target”: a make target whose command will never create the target

```
OBJFILES = functions.o calculate.o
```

```
clean:
```

```
    rm $(OBJFILES) calculate
```

- The **clean** target is a convention:
 - Remove generated files to “start over” from just the source
 - It’s “phony” because the target doesn’t exist and there are no sources, but it works because:
 - The target doesn’t exist, so it must be “remade” by running the command

All Target

- **all** target
 - Lists all the final products as sources, so “make all” builds everything

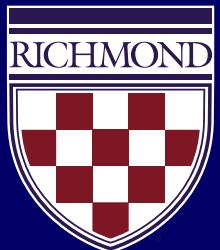
```
all: calculate functions.o calculate.o  
      # notice no commands this time
```

```
calculate: functions.o calculate.o  
          g++ calculate.o functions.o -o calculate
```

```
functions.o: functions.cpp  
           g++ -c functions.cpp
```

```
calculate.o: calculate.cpp  
           g++ -c calculate.cpp
```

Ask a question



Give it a try!

