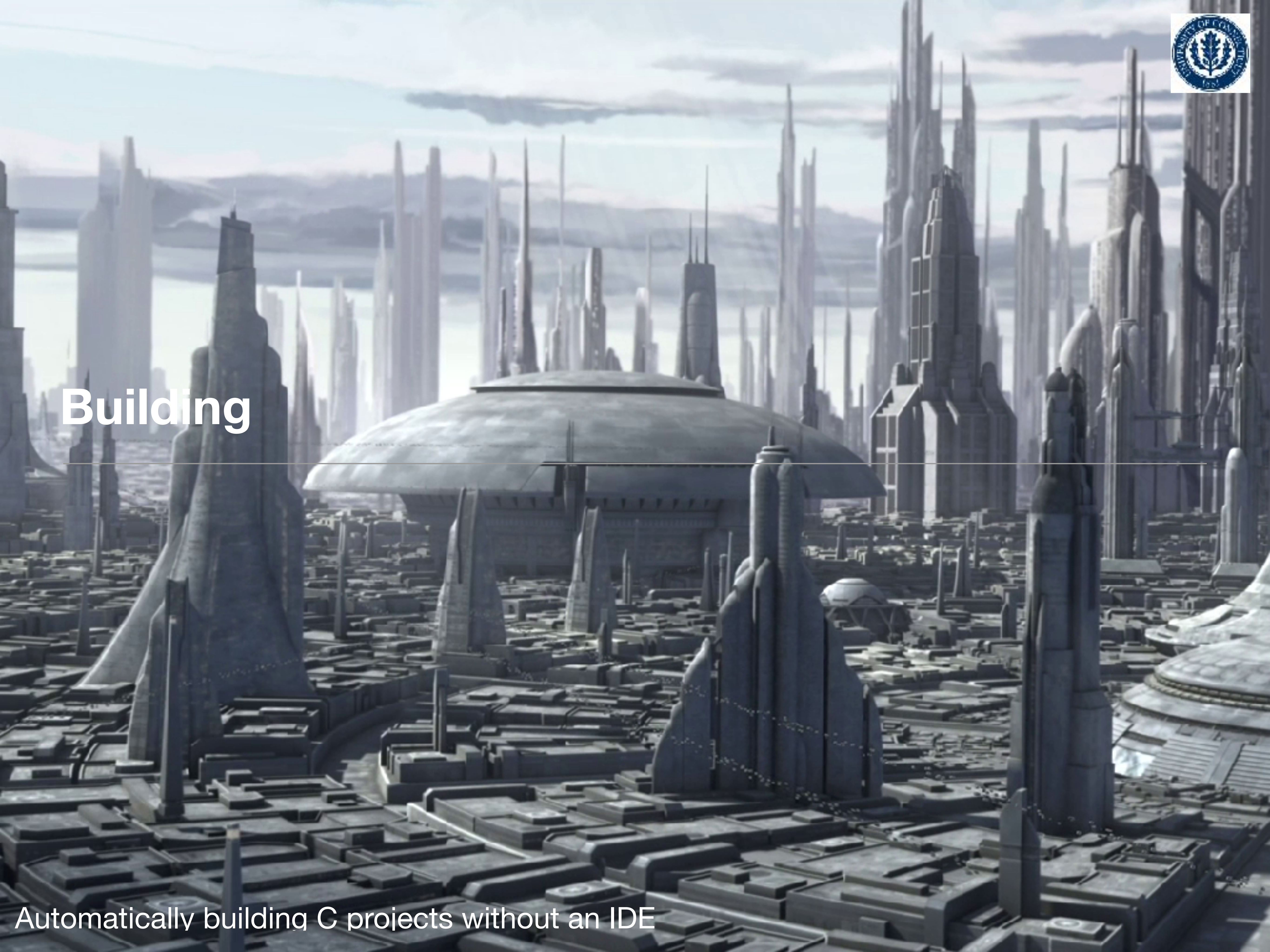


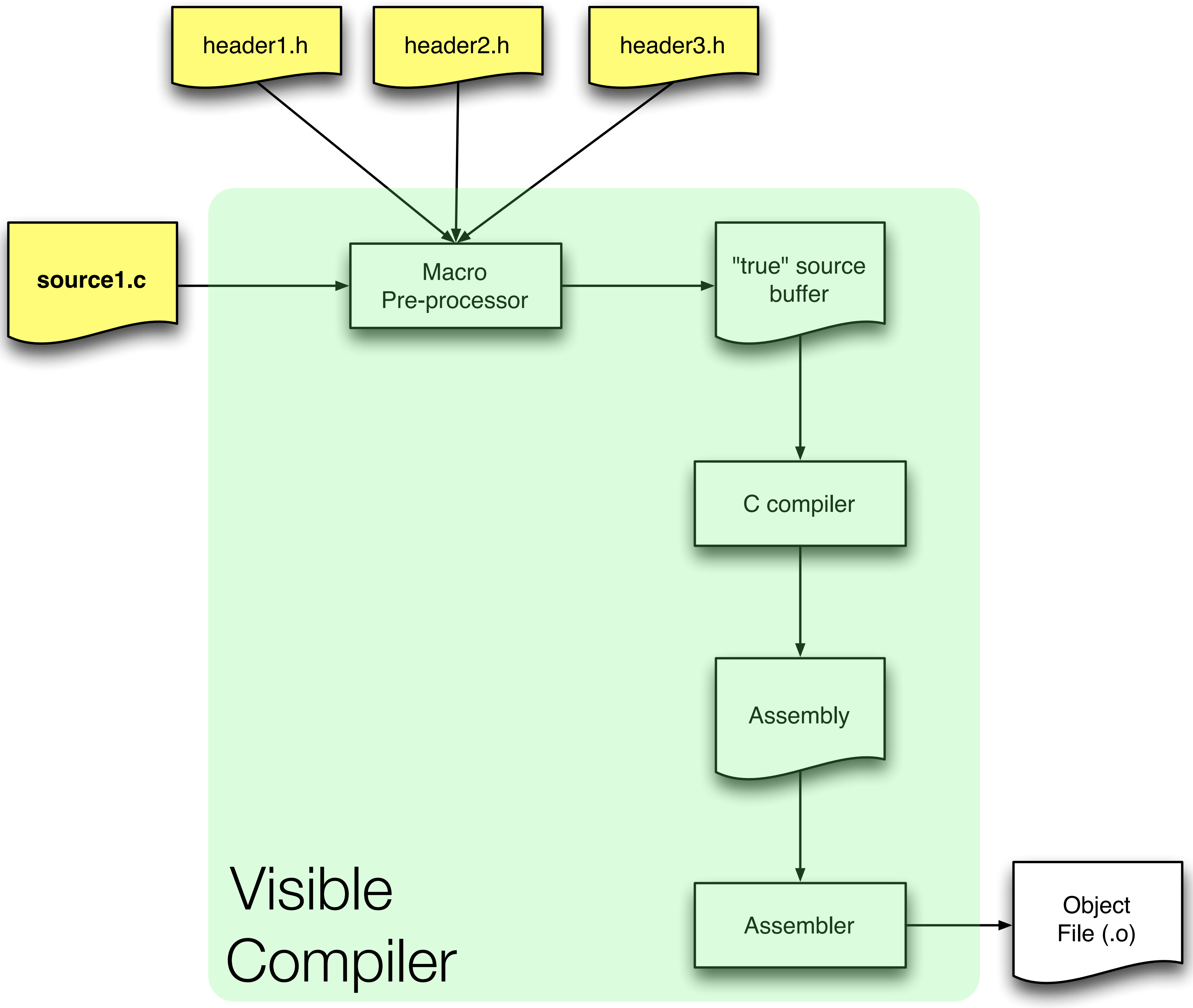
Building

Automatically building C projects without an IDE

Overview

- The C compiler workflow
  - Pre-processor
  - Compiler
- The linker
- The builder
  - Make
  - CMake
- Source code management

Compilation Workflow



Pre-processor

- Purpose
  - Carry out “macro expansion”
  - Concatenate all relevant header files
- Its output is *automatically* sent to the compiler
  - But you could save it to a file (option: -E)

Compiler

- By default
  - The compiler will compile every source file on the command line
  - And link all of them with the standard library
  - To produce an executable.
- But you can…
  - Compile one file at a time and create an object for each
  - Then link separately
  - Useful compiler option

-c	[only compile, do not link]
-g	[add debug information to the output]
-o	[send output to specific file]
-O1,-O2,-O3	[generate optimized code]
-S	[generate assembly code]

Assembler

- Automatically invoked by the compiler
- But you could do it by hand too!
    - Use a separate assembler called “as”

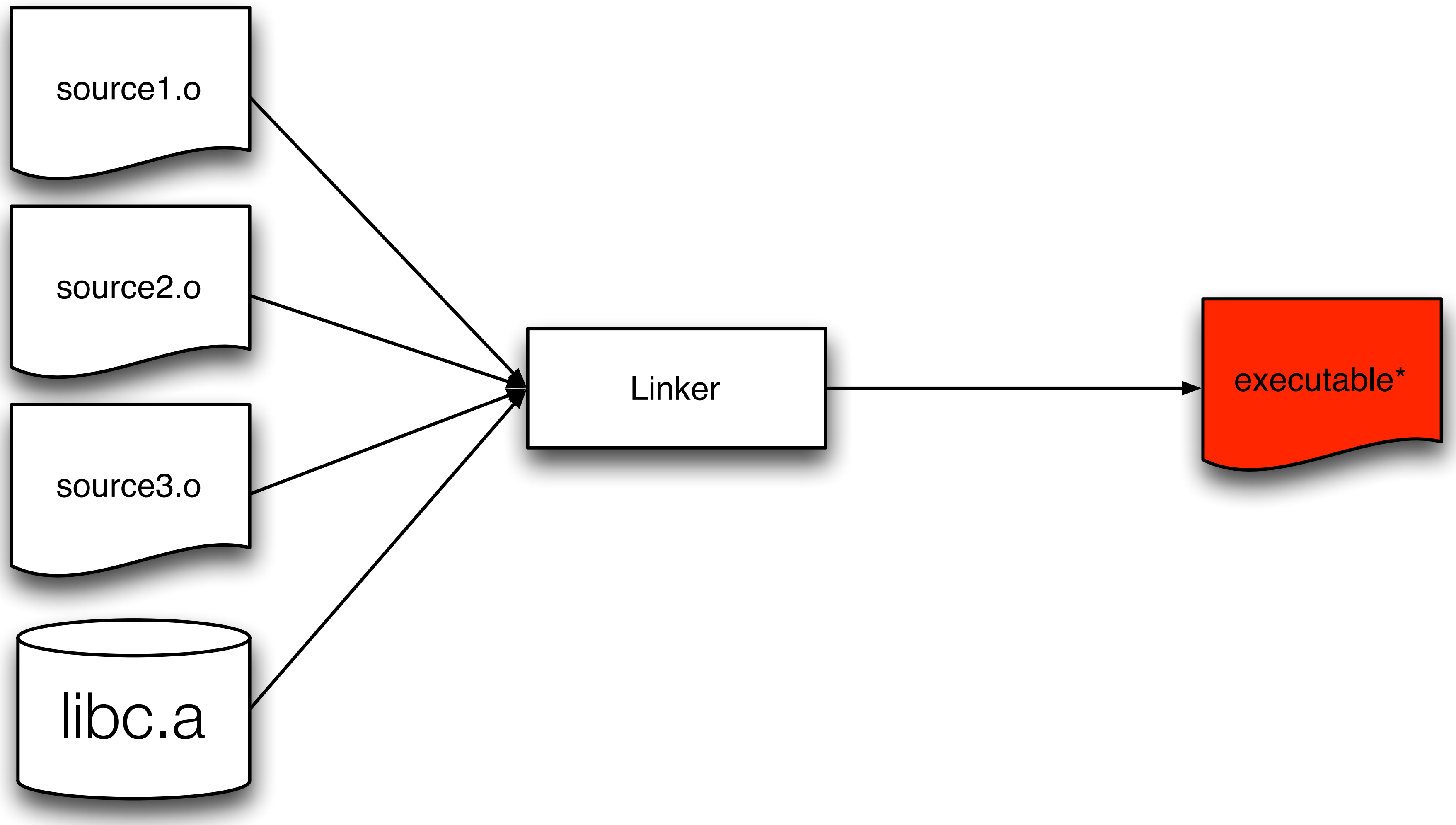
Overall…

- Four steps
    - Preprocess
    - Compile
    - Assemble
    - Link

```
cc -E -c hello.c   -o hello-bis.c

cc -S hello-bis.c -o hello-bis.s

as -c hello-bis.s -o hello-bis.o

cc hello-bis.o -o hello-bis
```

The Linker’s true power…

- Purpose
  - Pull together the object files and libraries

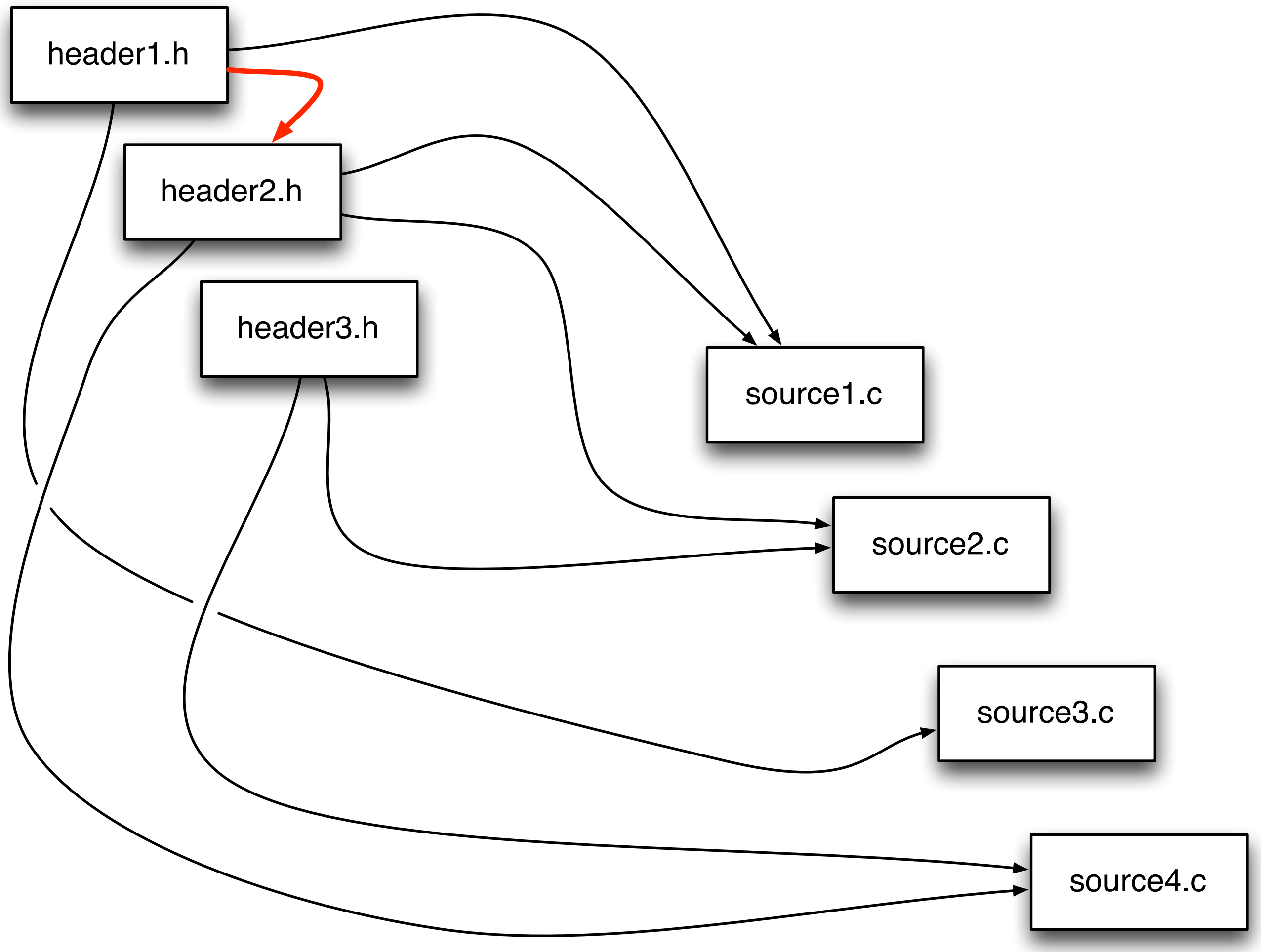


Automation with Makefile

- Purpose
  - Avoid typing all the commands by hand
  - Write a “recipe” file that does all the steps
    - Compile each source file
    - Link the executable
    - Factor out the options
    - Clean up the source tree

Gluing headers

- Consider a collection of
  - C source files
  - C header files
- Purpose of header files?
  - Publicize APIs for C sources
- Two types of header files
  - Standard
    - stdio,stdlib,….
  - User defined
    - For your own sources!



Use case

- Imagine building a reusable stack of integers!
    - Three files
        - Header file to publish the stack API. (stack.h)
        - C file to implement the stack API (stack.c)
        - C file to test the stack (stacktest.c)

Makefile

- Purpose
    - Define a collection of variables
    - Define a collection of rules
- All the rules have the same “shape”
    - What target is to be “build”
    - What it depends on
    - The command to build the target from the dependencies
        - The command can use variables

```
target : dependencies
    command
```

Makefile

- Bare bone example
  - Defines variable of object files
  - Defines compiler flag variable
  - Defines the all: target
  - Defines a target for the binary
  - Defines a target for each object
  - Defines a target to cleanup
- Special variables
  - $@ [name of target]
  - $< [name of dependency]
  - $^ [the entire list of dependencies]

```
OFILES = stack.o stacktest.o
CFLAGS = -g
all: stacktest

stacktest: $(OFILES)
   $(CC) $(OFILES) -o $@

stack.o: stack.c
   $(CC) -c $(CFLAGS) $<

stacktest.o: stacktest.c
   $(CC) -c $(CFLAGS) $<

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

Refining the Makefile

- Objective
  - Avoid repeating “boilerplate” rules
  - Write “generic” rule instead

```
OFILES = stack.o stacktest.o
CFLAGS = -g
all: stacktest

stacktest: $(OFILES)
   $(CC) $(OFILES) -o $@

%.o : %.c
   $(CC) -c $(CFLAGS) $<

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

Any **.o** can be derived from the corresponding **.c** with the command below

Refining the Makefile (2)

- Objective
  - Make already defines “boilerplate” rules for us for common source files!
  - Only define the first target (all:) and the rule for the executable

```
OFILES = stack.o stacktest.o
CFLAGS = -g
all: stacktest

stacktest: $(OFILES)
   $(CC) $(OFILES) -o $@

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

CMake

- Challenge
  - Makefiles are *platform dependent*
  - Different options on Linux, Windows, Mac, ….
- Solution
  - Use a *Makefile generator*
  - It takes care of
    - platform dependencies
    - boilerplate bits and pieces

Example

- Step 1
  - Create CMakeList.txt

```
cmake_minimum_required(VERSION 2.8.9)
project (hello)
add_executable(hello hello.c)
```

- Step 2
  - Run cmake
- Step 3
  - Use the generated makefile! (by running ‘make’).

Tutorials

- Loads of them online:
  - https://cmake.org/cmake/help/latest/guide/tutorial/index.html
  - https://cliutils.gitlab.io/modern-cmake/
  - https://mirkokiefer.com/cmake-by-example-f95eb47d45b1
- And loads more….