Makefiles & C++ Preview

CSE 333 Summer 2020

Instructor: Travis McGaha

Teaching Assistants:

Jeter Arellano	Ramya Challa	Kyrie Dowling
Ian Hsiao	Allen Jung	Sylvia Wang

Poll Everywhere

pollev.com/cse33320su

About how long did HW 1 take?

A. 0-3 Hours
B. 3-6 Hours
C. 6-9 Hours
D. 9-12 Hours
E. 12+ Hours
F. I didn't submit (yet) / I prefer not to say

Side question:
What is your favourite programming language?

Administrivia

- Exercise 7 posted yesterday, due Monday
  - Read a directory and open/copy text files found there
    - Copy *exactly* and *only* the bytes in the file(s). No extra output.
  - Good warm-up for…

- Homework 2 due in two weeks (7/23)
  - File system crawler, indexer, and search engine
  - Spec and starter files will be pushed out tonight
  - Most find it more work then HW1. Get started early!

Lecture Outline

- **Make and Build Tools**
- Makefile Basics
- C++ Preview

`make`

- `make` is a classic program for controlling what gets (re)compiled and how
  - Many other such programs exist (*e.g.* `ant`, `maven`, IDE "projects")

- `make` has tons of fancy features, but only two basic ideas:
  1) Scripts for executing commands
  2) Dependencies for avoiding unnecessary work

- To avoid "just teaching `make` features" (boring and narrow), let's focus more on the concepts…

Building Software

- 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
  - Repetitive: gcc -Wall -g -std=c11 -o widget foo.c bar.c baz.c
    - Retype this every time: 😭
    - Use up-arrow or history: 😐 (still retype after logout)
    - Have an alias or bash script: 🙂
    - Have a Makefile: 😊 (you're ahead of us)

"Real" Build Process

- On larger projects, you can't or don't want to have one big (set of) command(s) that are all ran every time anything is changed. When thinking on how to do things "smarter" consider:
  1) It could be worse: If `gcc` didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
  2) Source files could have multiple outputs (*e.g.* javadoc). You may have to type out the source file name(s) multiple times
  3) You don't want to have to document the build logic when you distribute source code. It should be relatively simple for others to build.
  4) You don't want to recompile everything every time you change something (especially if you have 10^5-10^7 files of source code)
- A script can handle 1-3 (use a variable for filenames for 2), but 4 is trickier

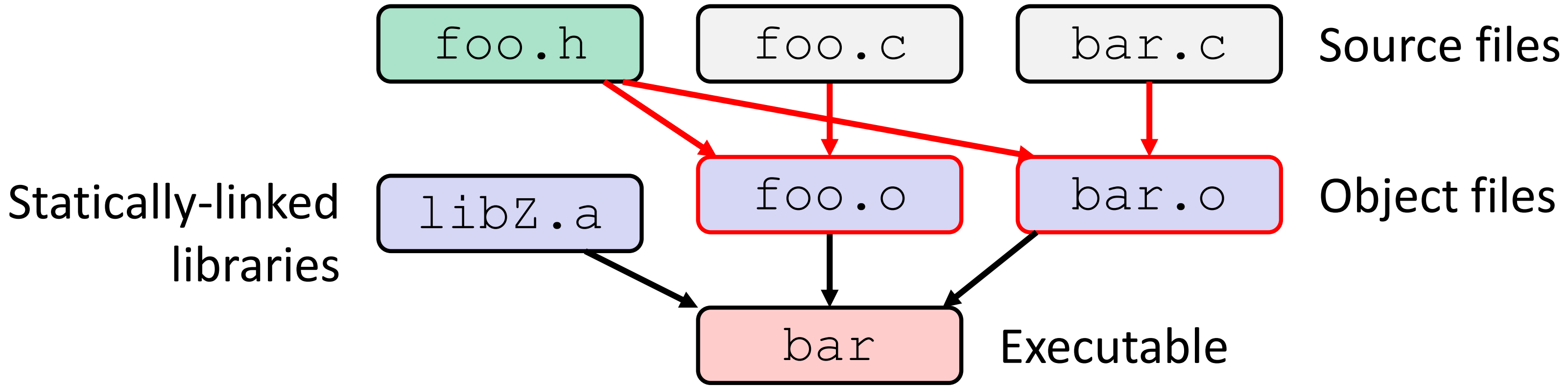
Recompilation Management

- The "theory" behind avoiding unnecessary compilation is a *dependency dag* (**d**irected, **a**cyclic **g**raph)

- To create a target t, you need sources $s_1, s_2, \ldots, s_n$ and a command c that directly or indirectly uses the sources
  - It t is newer than every source (file-modification times), assume there is no reason to rebuild it
  - Recursive building: if some source s_i is itself a target for some other sources, see if it needs to be rebuilt…
  - Cycles "make no sense"!

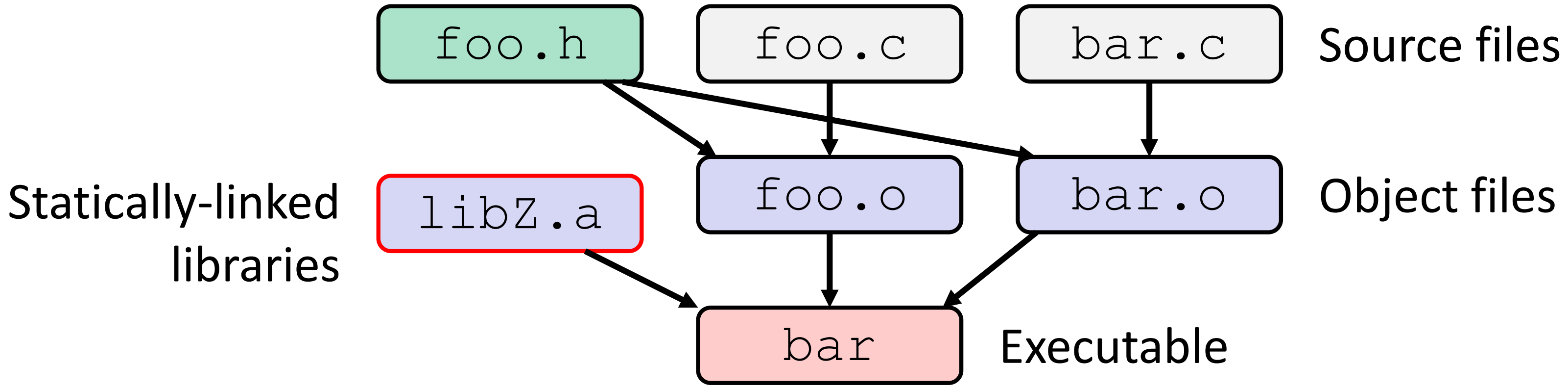
Theory Applied to C

NOTE: .c and .h files will always be sources, never targets.



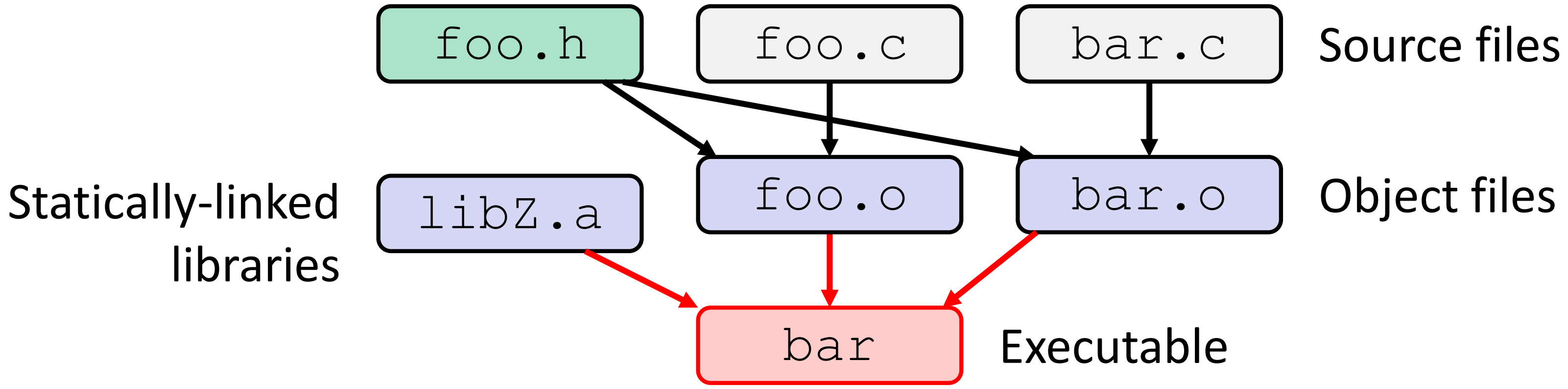
- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)

Theory Applied to C



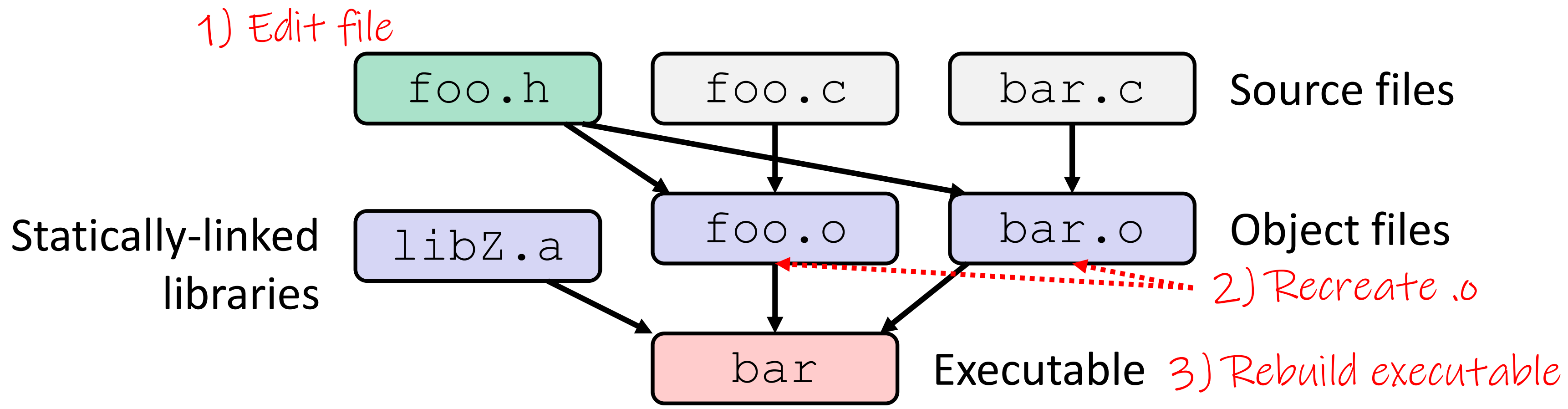
- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- An archive (library, `.a`) depends on included `.o` files

Theory Applied to C



- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- An archive (library, `.a`) depends on included `.o` files
- Creating an executable ("linking") depends on `.o` files and archives
  - Archives linked by `-L<path> -l<name>` (*e.g.* `-L. -lfoo` to get `libfoo.a` from current directory)

Theory Applied to C



- If one `.c` file changes, just need to recreate one `.o` file, maybe a library, and re-link
- If a `.h` file changes, may need to rebuild more
- Many more possibilities!

Lecture Outline

- Make and Build Tools
- **Makefile Basics**
- C++ Preview

Using make

```
bash$ make -f <makefileName> target
```

- Defaults:
  - If no `-f` specified, use a file named `Makefile` in current dir
  - If no `target` specified, will use the first one in the file
  - Will interpret commands in your default shell
    - Set `SHELL` variable in makefile to ensure

- Target execution:
  - Check each source in the source list:
    - If the source is a target in the makefile, then process it recursively
    - If some source does not exist, then error
    - If any source is newer than the target (or target does not exist), run `command` (presumably to update the target)

make Basics

- A makefile contains a bunch of triples:

```
① target: sources ②
← Tab → command ③
```

  - 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:

```
foo.o: foo.c foo.h bar.h
	gcc -Wall -o foo.o -c foo.c
```

`make` Variables

- You can define variables in a makefile:
  - All values are strings of text, no "types"
  - Variable names are case-sensitive and can't contain ':', '#', '=', or whitespace
- Example:

```
CC = gcc
CFLAGS = -Wall -std=c11
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
        $(CC) $(CFLAGS) -o widget $(OBJFILES)
```

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

"Phony" Targets

- "Phony Target": a make target whose command will never create the target.

```
OBJFILES = foo.o bar.o baz.o
clean:
	rm $(OBJFILES) widget *~
```

- `clean` 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
- Have several uses, such as: "all"
  - Lists all of the "final products" as sources, so "make all" builds everything.

"all" Example

"make all" check everything

```
all: prog B.class someLib.a
    # notice no commands this time

prog: foo.o bar.o main.o
      gcc -o prog foo.o bar.o main.o

B.class: B.java
      javac B.java

someLib.a: foo.o baz.o
      ar r foo.o baz.o

foo.o: foo.c foo.h header1.h header2.h
      gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

Makefile writing tips

STYLE TIP

- When creating a Makefile, draw the dependencies!!!!
- C Dependency Rules:
  - `.c` and `.h` files are never targets, only sources.
  - Each .c file will be compiled into a corresponding `.o` file
    - Header files will be implicitly used via `#include`
  - Executables will typically be built from one or more `.o` file
- Good Conventions:
  - Include a ‘clean’ rule
  - If you have more than one ‘final target’ have an ‘all’ rule
  - Put your singular ‘final target’ or ‘all’ as the first target.

Writing a Makefile Example

- "`talk`" program (find files on web with lecture slides)

`main.c` `speak.h` `speak.c` `shout.h` `shout.c`

speak.c

```
#include "speak.h"
...
```

main.c

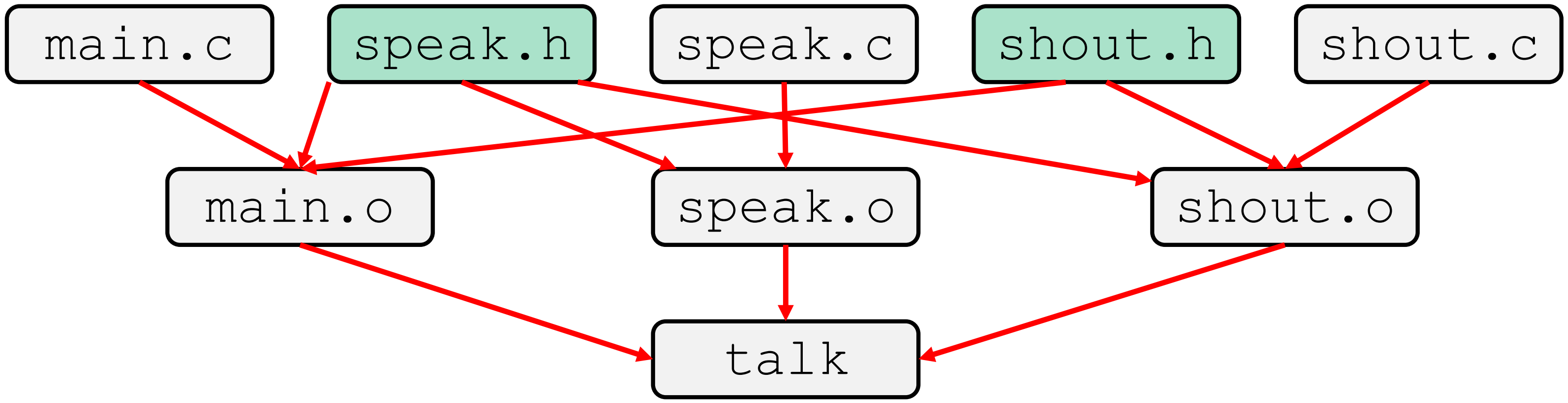
```
#include "speak.h"
#include "shout.h"

int main(int argc, char** argv) {…
```

shout.c

```
#include "speak.h"
#include "shout.h"
...
```

Writing a Makefile Example

- "`talk`" program (find files on web with lecture slides)



```
talk: main.o speak.o shout.o
          gcc $(CFLAGS) -o talk main.o speak.o shout.o

main.o: main.c speak.h shout.h
          gcc $(CFLAGS) -c main.c

speak.o: speak.c speak.h
          gcc $(CFLAGS) -c speak.c

shout.o: shout.c shout.h speak.h
          gcc $(CFLAGS) -c shout.c

clean:
          rm talk *.o
```

Revenge of the Funny Characters

- Special variables:
  - `$@` for target name
  - `$^` for all sources
  - `$<` for left-most source
  - Lots more! – see the documentation

- Examples:

```
# CC and CFLAGS defined above
widget: foo.o bar.o
      $(CC) $(CFLAGS) -o $@ $^
foo.o: foo.c foo.h bar.h
      $(CC) $(CFLAGS) -c $<
```

And more…

- There are a lot of "built-in" rules – see documentation
- There are "suffix" rules and "pattern" rules
  - Example:

```
%.class: %.java
        javac $<   # we need the $< here
```

- Remember that you can put *any* shell command – even whole scripts!
- You can repeat target names to add more dependencies
- Often this stuff is more useful for reading makefiles than writing your own (until some day…)

On the topic of compilation…

This is cool trivia, but totally unnecessary to know. Hopefully you enjoy it :)

- What's the best compilation message?

```
bash$ gcc -g - Wall -std=c11 -o msg msg.c
msg.c:1:6: warning: 'main' is usually a
                   function [-Wmain]
```

- C will let you do many things, many of which you should not do.
  - This is an example of something you should **NEVER** do.
- You can write a program that runs and prints!
  - Not portable though…

not_a_function.c

```
const int main[] = {
  -443987883, 440, 113408, -1922629632,
  4149, 2013696, 84869120, 15544,
  266023168, 1818576901, 1126199148, 857752915,
  1699479555, 1936269379, 1701344288, 1936024096,
  -1878384268, 521126749, 1096237312, 1447165833
}
```

Credit to this blog:
http://jroweboy.github.io/c/asm/2015/01/26/when-is-main-not-a-function.html

Lecture Outline

- Make and Build Tools
- Makefile Basics
- **C++ Preview**

Programming Terminology Review

- **Encapsulation and Abstraction:** Hiding implementation details (restricting access) and associating behaviors (methods) with data

- **Polymorphism:** The provision of a single interface to entities of different types

- **Generics:** Algorithms written in terms of types *to-be-specified-later*

Encapsulation and Abstraction (C)

- Used header file conventions and the `static` specifier to separate "private" functions, definitions, and constants from "public"

- Used forward-declared `struct`s and opaque pointers (*i.e.* `void*`) to hide implementation-specific details

- Can't associate behaviors with encapsulated state
  - Functions that operate on a `LinkedList` not actually tied to the struct

Really difficult to mimic – implemented primarily via coding conventions

Encapsulation and Abstraction (C++)

- Support for classes and objects!
  - Public, private, and protected access specifiers
  - **Methods** and **instance variables** ("`this`")
  - (Multiple!) inheritance

- Polymorphism
  - *Static polymorphism:* multiple functions or methods with the same name, but different argument types (overloading)
    - Works for all functions, not just class members
  - *Dynamic (subtype) polymorphism:* derived classes can override methods of parents, and methods will be dispatched correctly

Generics (C)

- Generic linked list and hash table by using `void*` payload
- Function pointers to generalize different behavior for data structures
  - Comparisons, deallocation, pickling up state, etc.

Emulated generic data structures primarily by disabling type system

Generics (C++)

- Templates facilitate generic data types
  - *Parametric polymorphism:* same idea as Java generics, but different in details, particularly implementation
    - A vector of `int`s: `vector<int> x;`
    - A vector of `float`s: `vector<float> x;`
    - A vector of (vectors of `float`s): `vector<vector<float>> x;`

- Specialized casts to increase type safety

Namespaces (C)

- Names are global and visible everywhere
  - Can use `static` to prevent a name from being visible outside a source file (as close as C gets to "private")
- Naming conventions help avoid collisions in the global namespace
  - *e.g.* `LinkedList_Allocate`, `HTIterator_Next`, etc.

Avoid collisions primarily via coding conventions

Namespaces (C++)

- Explicit namespaces!
  - The linked list module could define an "`LL`" namespace while the hash table module could define an "`HT`" namespace
  - Both modules could define an Iterator class
    - One would be globally named `LL::Iterator` and the other would be globally named `HT::Iterator`
- Classes also allow duplicate names without collisions
  - Classes can also define their own pseudo-namespace, very similar to Java static inner classes

Standard Library (C)

- C does not provide any standard data structures
  - We had to implement our own linked list and hash table
- Hopefully, you can use somebody else's libraries
  - But C's lack of abstraction, encapsulation, and generics means you'll probably end up tweak them or tweak your code to use them

YOU implement the data structures that you need

Standard Library (C++)

- **Generic containers:** bitset, queue, list, associative array (including hash table), deque, set, stack, and vector
  - And iterators for most of these

- **A `string` class:** hides the implementation of strings

- **Streams:** allows you to stream data to and from objects, consoles, files, strings, and so on

- **Generic algorithms:** sort, filter, remove duplicates, etc.

Error Handling (C)

- Error handling is a pain
- Define error codes and return them
  - Either directly return or via a "global" like `errno`
  - No type checking: does `1` mean `EXIT_FAILURE` or `true`?

- Customers and implementors need to constantly test return values
  - *e.g.* if `a()` calls `b()`, which calls `c()`
    - `a` depends on `b` to propagate an error in `c` back to it

Error handling is a pain – mixture of coding conventions and discipline

Error Handling (C++)

- Supports exceptions!
  - `try` / `throw` / `catch`
  - If used with discipline, can simplify error processing
  - If used carelessly, can complicate memory management
    - Consider: `a()` calls `b()`, which calls `c()`
      - If `c()` throws an exception that `b()` doesn't catch, you might not get a chance to clean up resources allocated inside `b()`

- We will largely avoid in 333
  - You still benefit from having more interpretable errors!
  - But much C++ code still needs to work with C & old C++ libraries, so still uses return codes, **`exit`**`()`, etc.

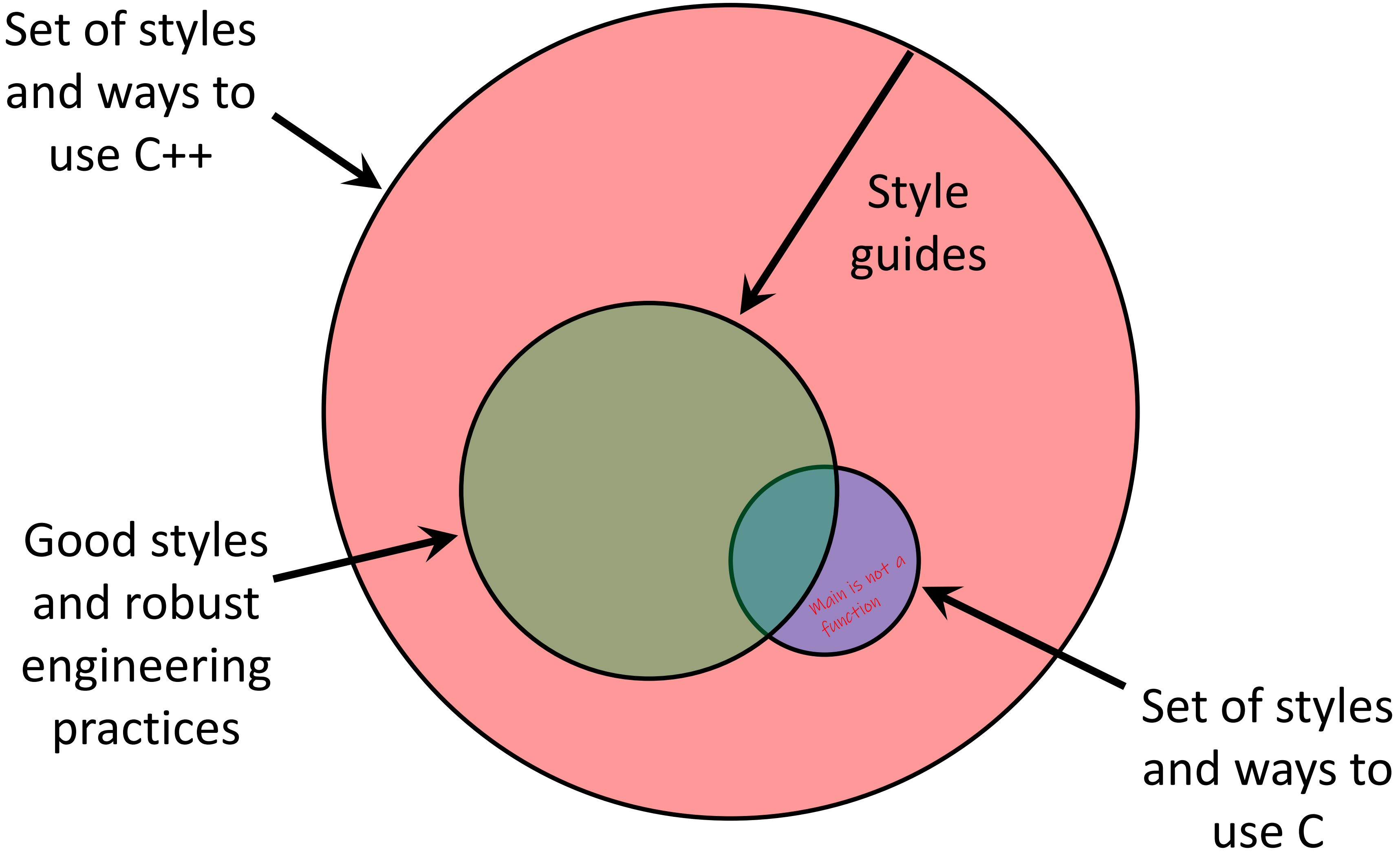
Some Tasks Still Hurt in C++

- Memory management
  - C++ has no garbage collector
    - You still have to manage memory allocation & deallocation and track
    - It's still possible to have leaks, double frees, and so on
  - But there are some things that help
    - "Smart pointers"
      - Classes that encapsulate pointers and track reference counts
      - Deallocate memory when the reference count goes to zero
    - C++'s constructors and destructors permit a pattern known as "Resource Allocation Is Initialization" (RAII)
      - Useful for releasing memory, locks, database transactions, etc.

Some Tasks Still Hurt in C++

- C++ doesn't guarantee type or memory safety
  - You can still:
    - Forcibly cast pointers between incompatible types
    - Walk off the end of an array and smash memory
    - Have dangling pointers
    - Conjure up a pointer to an arbitrary address of your choosing

How to Think About C++


Set of styles and ways to use C++
Style guides
Good styles and robust engineering practices
Main is not a function
Set of styles and ways to use C

Or…

In the hands of a disciplined programmer, C++ is a powerful tool

But if you're not so disciplined about how you use C++…