C Tools & Compilation

Intro to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema	Andrew Rigas	Anisha Bhatia
Audrey Yang	Craig Lee	Daniel Duan
David LuoZhang	Eddy Yang	Ernest Ng
Heyi Liu	Janavi Chadha	Jason Hom
Katherine Wang	Kyrie Dowling	Mohamed Abaker
Noam Elul	Patricia Agnes	Patrick Kehinde Jr.
Ria Sharma	Sarah Luthra	Sofia Mouchtaris

Upcoming Due Dates

- HW07 (Deque & RPN) Due Friday 11/11 @ 11:59 pm
  - Assignments are increasing in time required to complete. Please try to not let the work accumulate

- Midsemester Survey Due Wednesday 11/9 @ 11:59 pm

- Students taking final exams through the Weingarten Center need to schedule by Nov 30th to guarantee extended time for an exam.

Lecture Outline

- **Memory Errors, Valgrind & gdb**
- C Header Files & Modules
- C compilation, definitions vs declarations, CPP
- Makefiles

Dynamic Memory Pitfalls

- Buffer Overflows
  - E.g. ask for 10 bytes, but write 11 bytes
  - Could overwrite information needed to manage the heap
  - Common when forgetting the null-terminator on malloc'd strings
- Not checking for **`NULL`**
  - Malloc returns NULL if out of memory
  - Should check this after every call to malloc
- Giving **`free()`** a pointer to the middle of an allocated region
  - Free won't recognize the block of memory and probably crash
- Giving free() a pointer that has already been freed
  - Will interfere with the management of the heap and likely crash
- **`malloc`** does NOT initialize memory
  - There are other functions like **`calloc`** that will zero out memory

Memory Leaks

- The most common Memory Pitfall
- What happens if we malloc something, but don’t free it?
  - That block of memory cannot be reallocated, even if we don’t use it anymore, until it is **`free`**d
  - If this happens enough, we run out of heap space and program may slow down and eventually crash

- Garbage Collection
  - Automatically “frees” anything once the program has lost all references to it
  - Affects performance, but avoid memory leaks
  - Java has this, C doesn’t

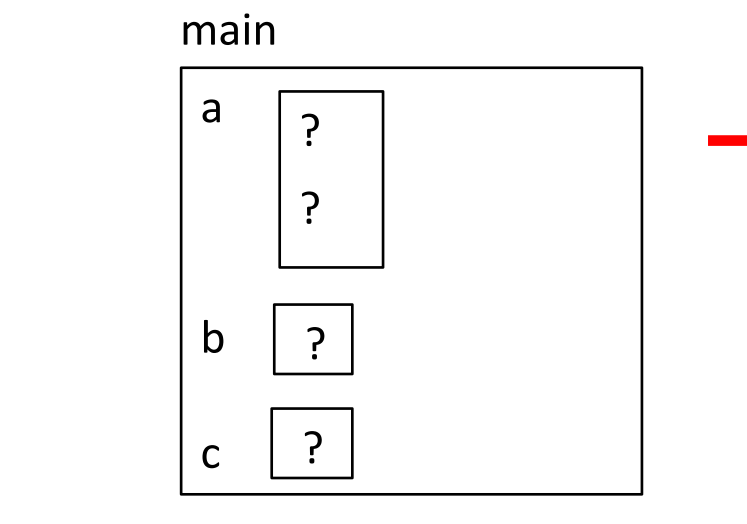
Poll Everywhere

pollev.com/tqm

- Which line below is first to (most likely) cause a crash?
  - Yes, there are a lot of bugs, but not all cause a crash ☺
  - See if you can find all the bugs!

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

1   a[2] = 5;
2   b[0] += 2;
3   c = b+3;
4   free(&(a[0]));
5   free(b);
6   free(b);
7   b[0] = 5;

  return 0;
}
```

Memory Corruption - What Happens?



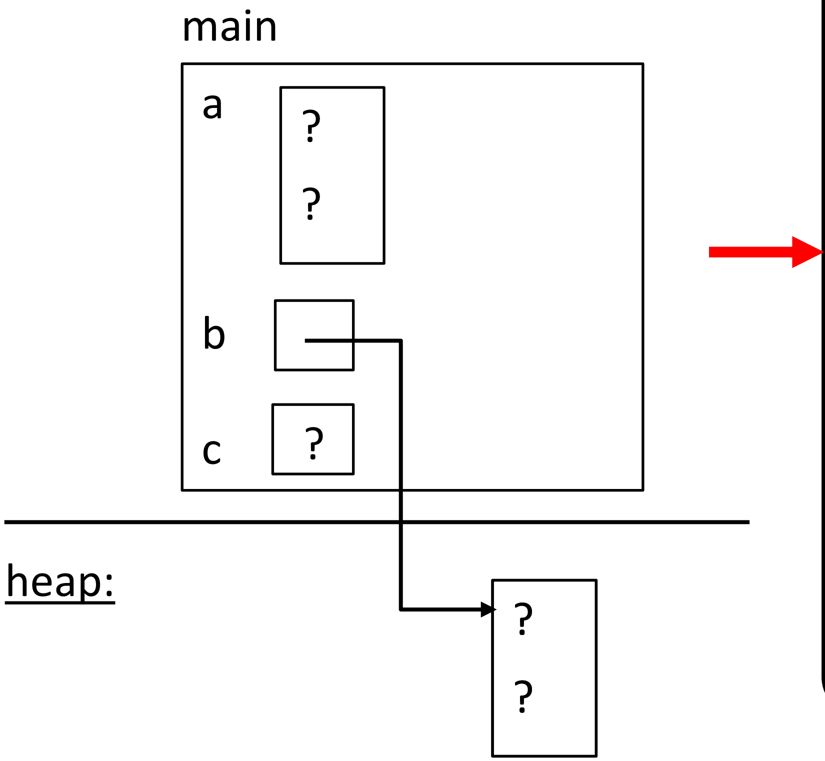
heap:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

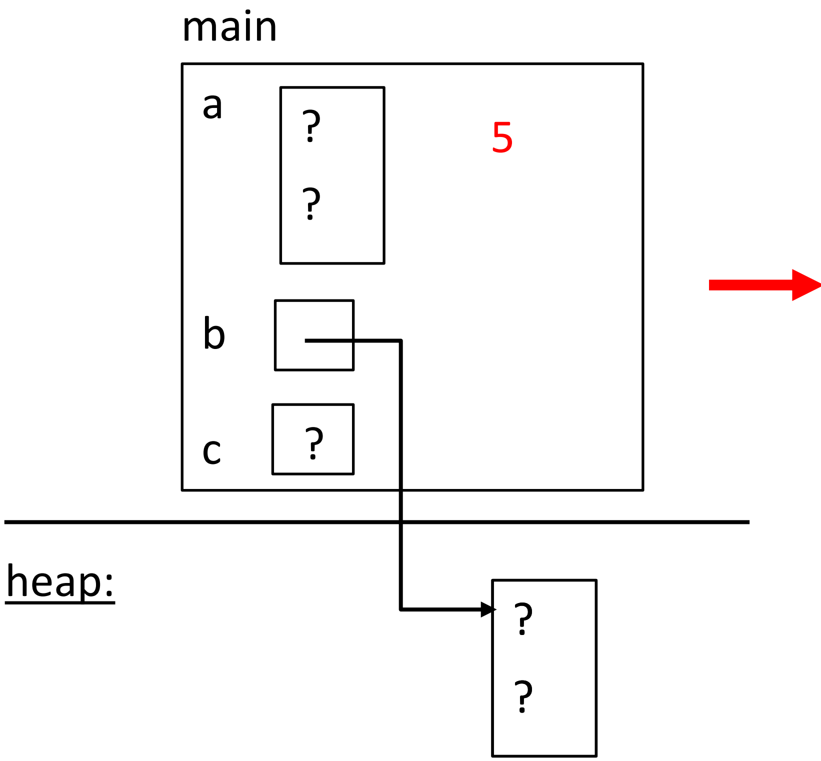



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

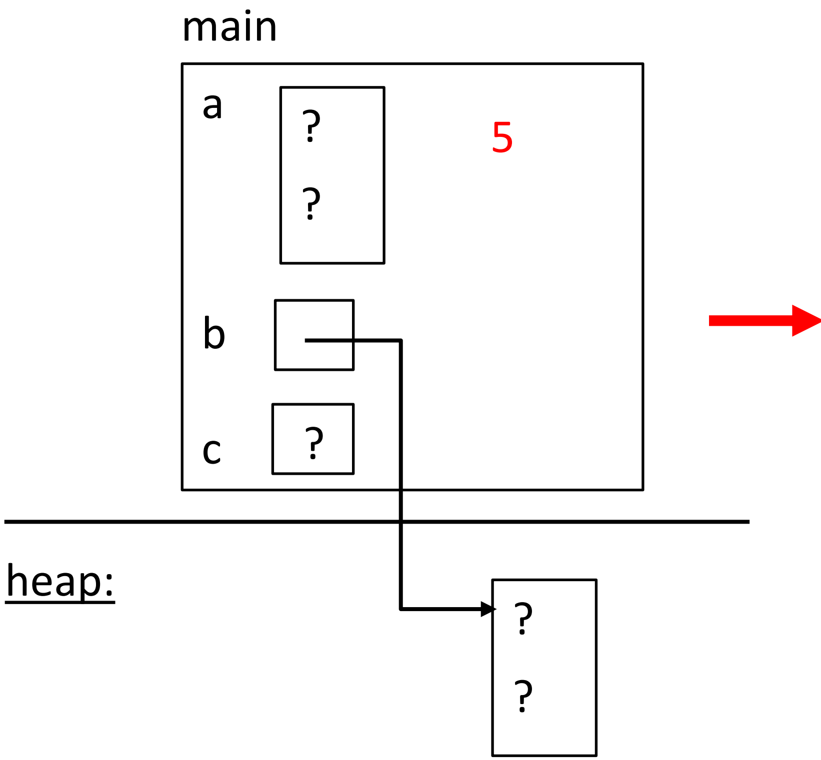



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

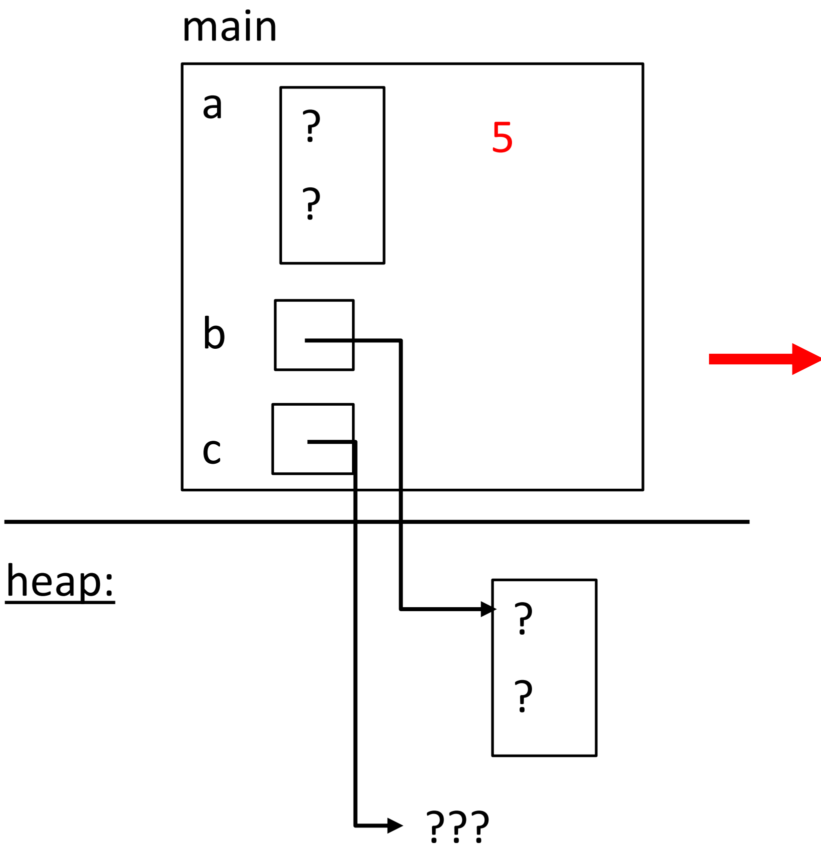



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

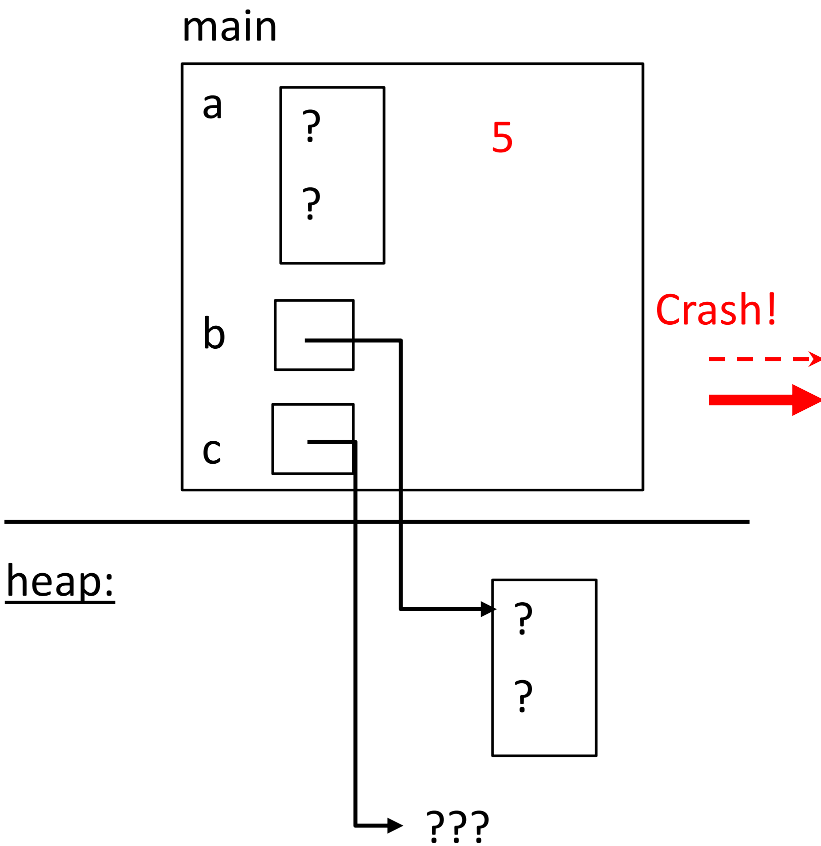



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

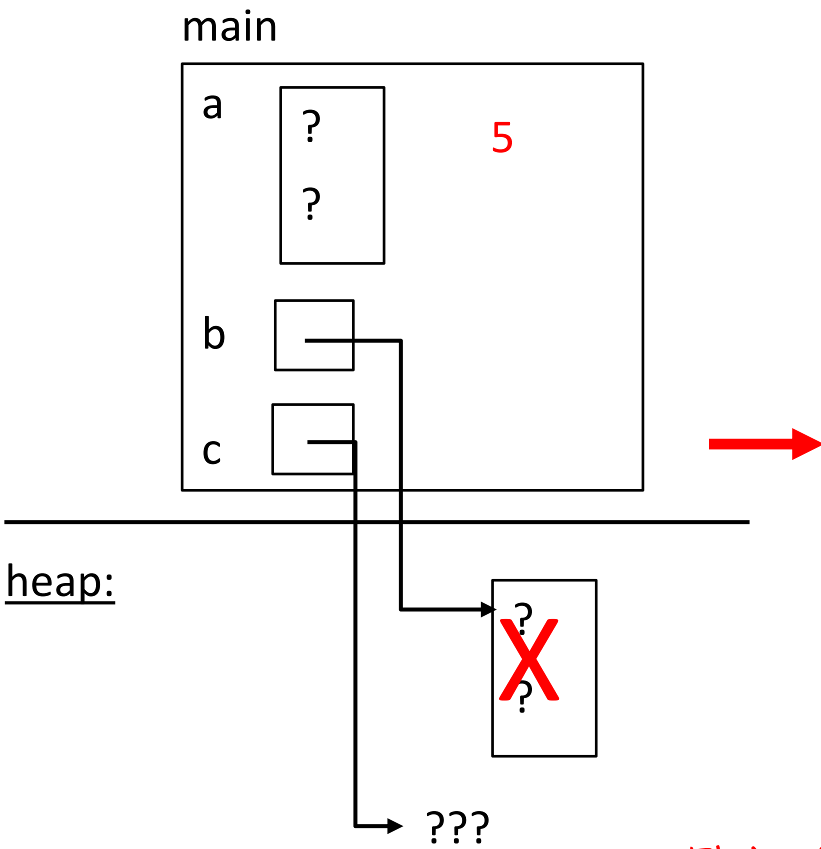



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?



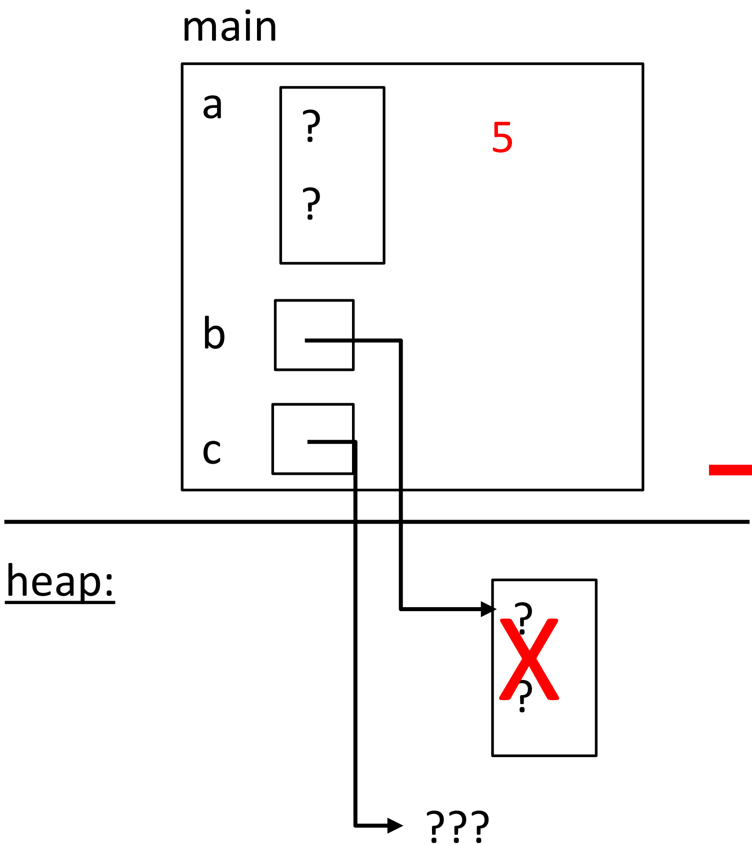

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

This "double free" would also cause the program to crash

memcorrupt.c

Memory Corruption - What Happens?

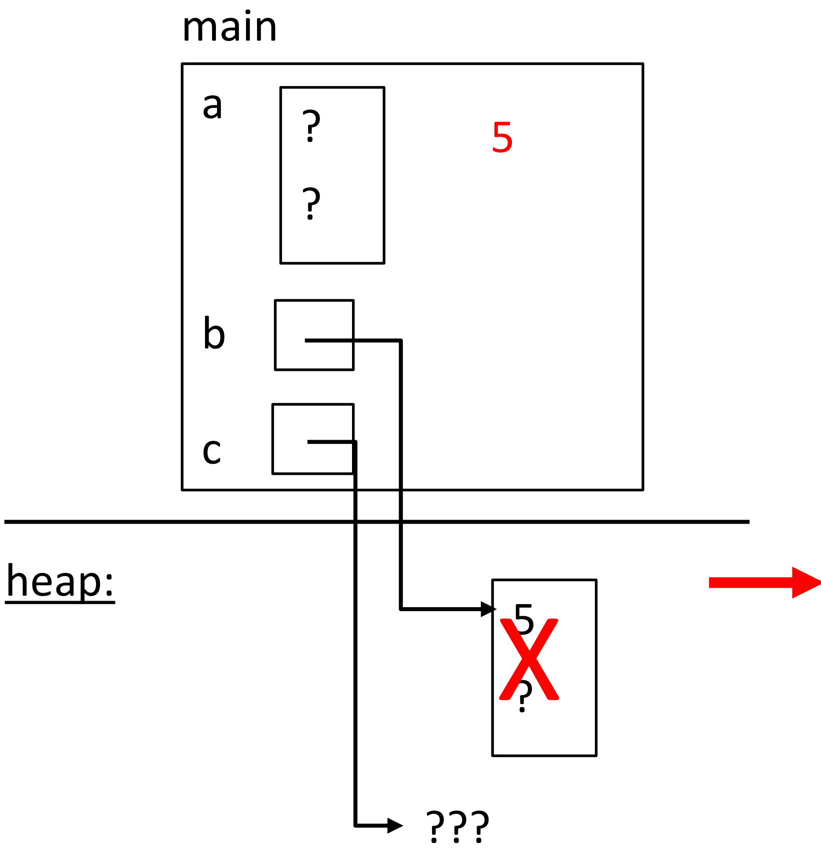



```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?




```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Motivation

- The assignments will start getting bigger and are more open ended. Lots of potential for bugs

- **<u>Debugging is a skill that you will need throughout your programming career</u>**

- gdb (GNU Debugger) is a debugging tool
  - Very useful in tracking undefined behavior

- Valgrind
  - Checks for various memory errors
  - IF you have odd behavior, valgrind may point out the cause.

Segmentation Fault

- C doesn't tell you much when it crashes, usually just prints "Segmentation fault (Core Dumped)"

- Causes:
  - Dereferencing an unitialized pointer
  - Dereferencing NULL
  - Using a previously freed pointer
  - Writing beyond the bounds of an array
  - …

- GDB is incredibly useful for debugging a segmentation fault

GDB “Cheat Sheet”

- run <command_line_args>
  - Runs the program with specified command line arguments
- backtrace
  - Prints out the “trace” of where functions were invoked to get to the current spot in the program
- up/down
  - Can be used to look at the function who called us/we are calling
- print <expression>
  - Prints out a value so that we can examine it
- quit
  - Quit the program

GDB "Cheat Sheet" Part 2

- tui enable
  - Used to enable the Text User Interface
- step
  - Move forward a line, steps into a function if we call one
- next
  - Moves forward a line, doesn't step into a function if called
- continue
  - Run until we crash, hit a breakpoint, or program finishes
- breakpoints
  - Next slide

gdb breakpoints

- Usage:
  - break <function_name>
  - break <filename:line#>
    - Example: `break main.c:20`
  - info break
    - Prints out information of all breakpoints
  - del <id>
    - Deletes the breakpoint with specified num.
    - Get breakpoint num with `info break`

Valgrind

- Tool used for identifying memory errors
- **Will be used on your HW submissions going forward**
- Detects:
  - Use of uninitialized memory
  - Reading/writing memory after it has been freed
  - Reading/writing to the end of malloc'd blocks
  - Reading/writing to inappropriate areas on the stack
  - Memory leaks where pointers to malloc'd blocks are lost
- Run with

```
valgrind --leak-check=full ./executable
```

Brief GDB & Valgrind Demo: Seg Faults

- IF NOTHING ELSE FROM GDB: GDB is very useful for finding a segmentation fault
  - Run the code on gdb till segmentation fault
  - Type in the command <backtrace>

- Commands:
  - gdb ./executable
  - run
  - backtrace

Lecture Outline

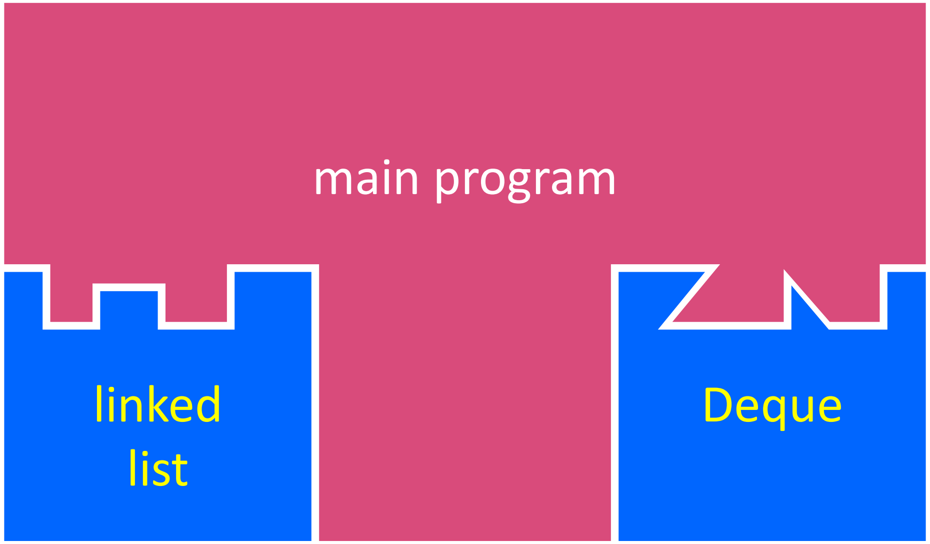
- Memory Errors, Valgrind & gdb
- **C Header Files & Modules**
- C compilation, definitions vs declarations, CPP
- Makefiles

Multi-File C Programs

- Let's create a queue *module*
  - A module is a self-contained piece of an overall program
    - Has externally visible functions that customers can invoke
    - Has externally visible `typedef`s, and perhaps global variables, that customers can use
    - May have internal functions, `typedef`s, or global variables that customers should *not* look at
  - The module's *interface* is its set of public functions, `typedef`s, and global variables

Modularity

- The degree to which components of a system can be separated and recombined
  - "Loose coupling" and "separation of concerns"
  - Modules can be developed independently
  - Modules can be re-used in different projects



C Header Files

- Header: a file whose only purpose is to be `#include`'d
  - Generally has a filename `.h` extension
  - Holds the variables, types, and function prototype declarations that make up the interface to a module
  - There are <system-defined> and "programmer-defined" headers

#include <stdio.h>   #include "my_linkedlist.h"

- Main Idea:
  - Every `name.c` is intended to be a module that has a `name.h`
  - `name.h` declares the interface to that module
  - Other modules can use `name` by `#include`-ing `name.h`
    - They should assume as little as possible about the implementation in `name.c`

C Module Conventions (1 of 2)

- File contents:
  - `.h` files only contain *declarations*, never *definitions*
  - `.c` files never contain prototype declarations for functions that are intended to be exported through the module interface
  - Public-facing functions are **`ModuleName_functionname`**`()` and take a pointer to "`this`" as their first argument

  Deque_Push_Back

  The "object" we are calling the function on

- Including:
  - ***NEVER*** `#include` a `.c` file – only `#include .h` files
  - `#include` all of headers you reference, even if another header (transitively) includes some of them
- Compiling:
  - Any `.c` file with an associated `.h` file should be able to be compiled into a `.o` file
    - The `.c` file should `#include` the `.h` file; the compiler will check definitions and declarations for consistency

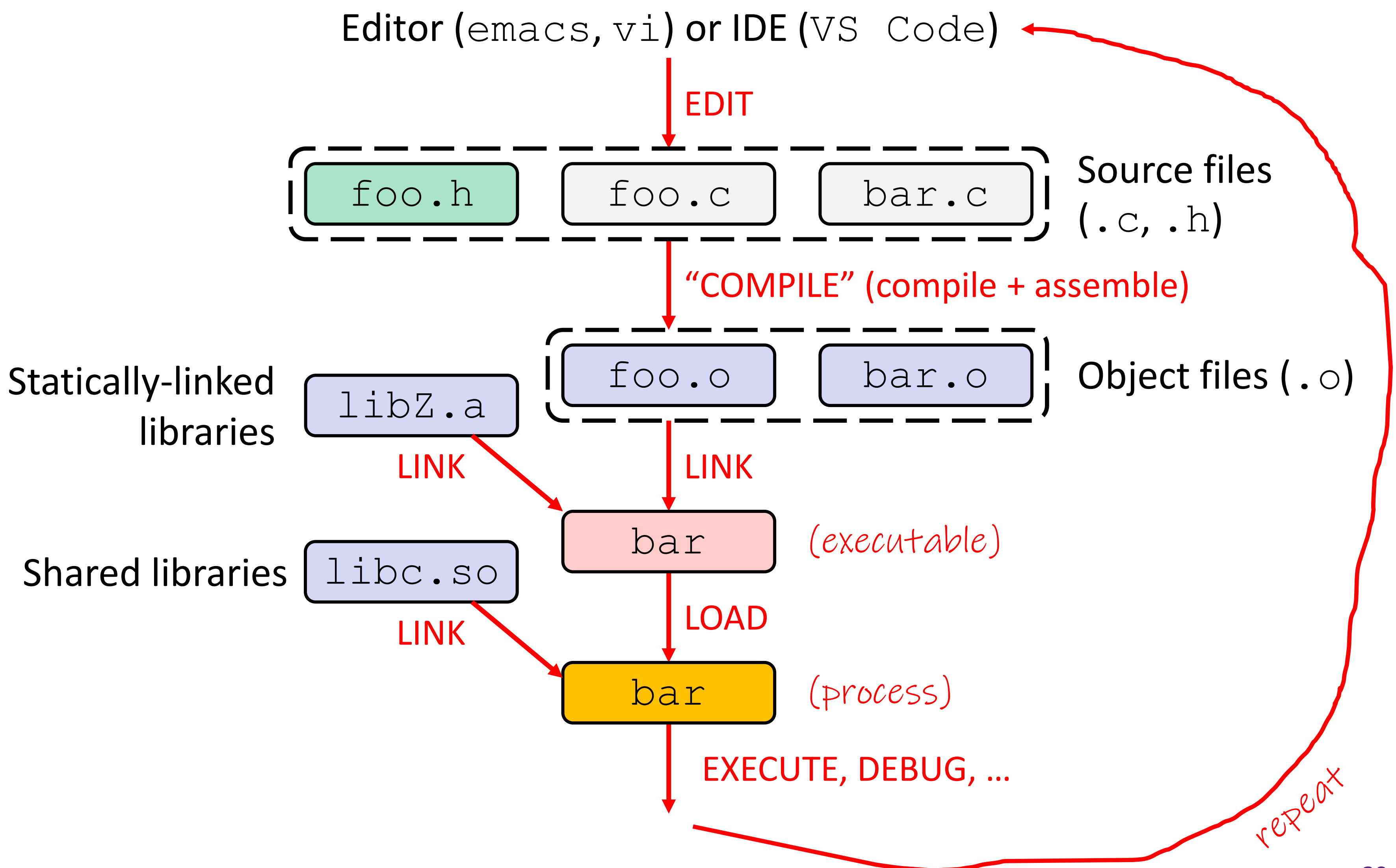
C Module Conventions (2 of 2)

- Commenting:
  - If a function is declared in a header file (`.h`) and defined in a C file (`.c`), *the header needs full documentation because it is the public specification*
    - Don't copy-paste the comment into the C file (don't want two copies that can get out of sync)
  - If prototype and implementation are in the same C file:
    - School of thought #1: Full comment on the prototype at the top of the file, no comment (or "declared above") on code
    - School of thought #2: Prototype is for the compiler and doesn't need comment; comment the code to keep them together

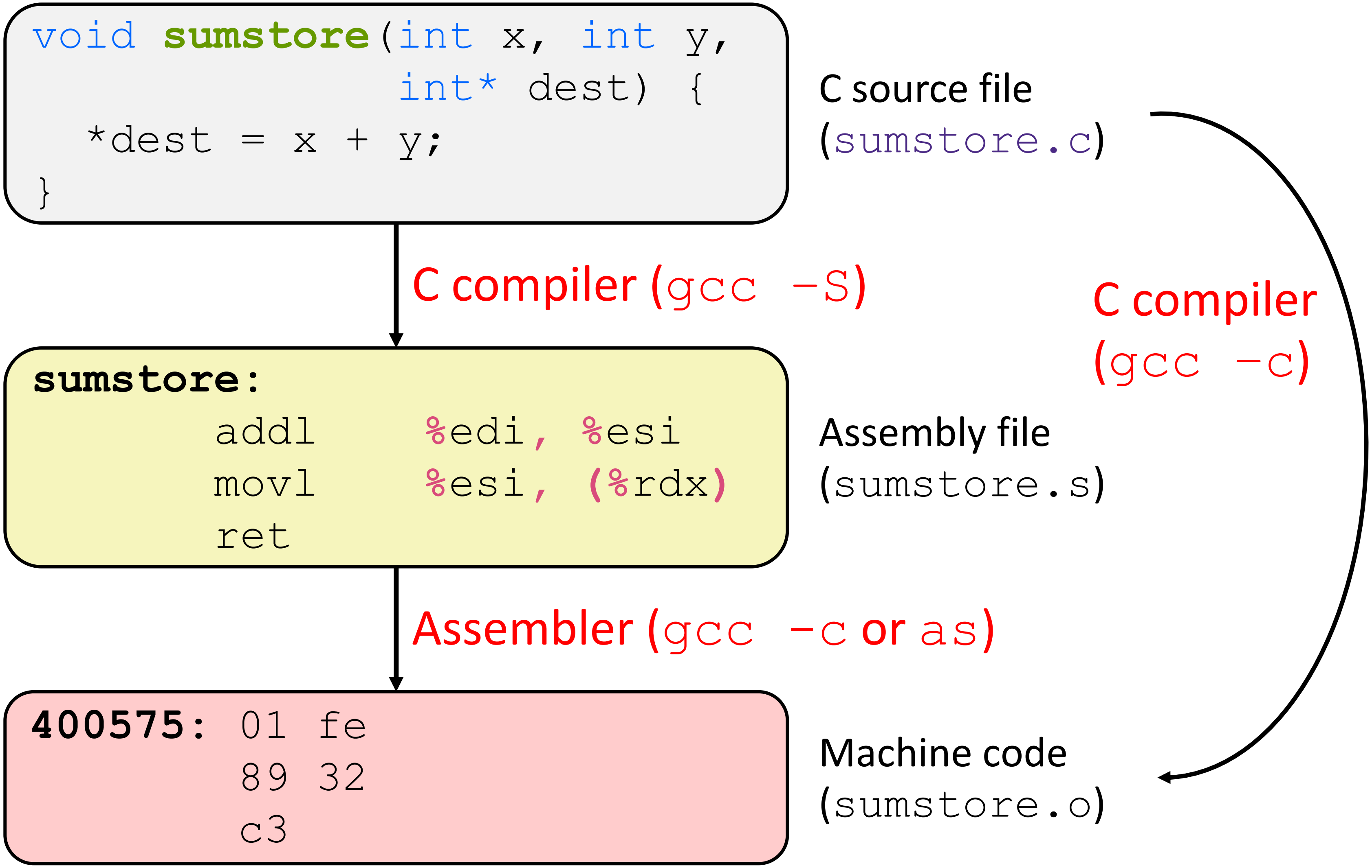
Lecture Outline

- Memory Errors, Valgrind & gdb
- C Header Files & Modules
- **C compilation, definitions vs declarations, CPP**
- Makefiles

C Workflow


Editor (emacs, vi) or IDE (VS Code)
EDIT
foo.h
foo.c
bar.c
Source files (.c, .h)
"COMPILE" (compile + assemble)
Statically-linked libraries
libZ.a
foo.o
bar.o
Object files (.o)
LINK
LINK
bar
(executable)
Shared libraries
libc.so
LOAD
LINK
bar
(process)
EXECUTE, DEBUG, …
repeat

C to Machine Code



Function Definitions

- Generic format:

```
returnType fname(type param1, …, type paramN) {
    // statements
}
```

```
// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i = 1; i <= max; i++) {
    sum += i;
  }

  return sum;
}
```

Function Ordering

- You *shouldn't* call a function that hasn't been declared yet

C compiler goes line by line

```
int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return EXIT_SUCCESS;
}

// sum of integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;

  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}
```

"What is sumTo()?"

Solution 1: Reverse Ordering

- Simple solution; however, imposes ordering restriction on writing functions (who-calls-what?)

```
// sum of integers from 1 to max
int sumTo(int max) {          ← defined
  int i, sum = 0;

  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}

int main(int argc, char** argv) {     Seen later
  printf("sumTo(5) is: %d\n", sumTo(5)); ←
  return EXIT_SUCCESS;
}
```

Solution 2: Function Declaration

- Teaches the compiler arguments and return types; function definitions can then be in a logical order
  - Function comment usually by the *prototype*

Parameter names optional

(1) Declare functions first

(2) Main function

(3) Define functions later

```
// sum of integers from 1 to max
int sumTo(int);  // func prototype

int main(int argc, char** argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return EXIT_SUCCESS;
}

int sumTo(int max) {
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}
```

Function Declaration vs. Definition

- C makes a careful distinction between these two

- Definition: the thing itself
  - *e.g.* code for function, variable definition that creates storage
  - Must be **exactly one** definition of each thing (no duplicates)

- Declaration: description of a thing
  - *e.g.* function prototype, external variable declaration
    - Often in header files and incorporated via `#include`
    - Should also `#include` declaration in the file with the actual definition to check for consistency
  - Needs to appear in **all files** that use that thing
    - Should appear before first use

More on header files & #include in this lecture

#include and the C Preprocessor

- The C preprocessor (`cpp`) transforms your source code before the compiler runs – it's a simple copy-and-replace text processor(!) with a memory
  - Input is a C file (text) and output is still a C file (text)
  - Processes the directives it finds in your code (`#directive`)
    - *e.g.* `#include "ll.h"` is replaced by the post-processed content of `ll.h`
    - *e.g.* `#define PI 3.1415` defines a symbol (a string!) and replaces later occurrences
    - Several others that we'll see soon…
  - Run on your behalf by `gcc` during compilation
  - Note: `#include <foo.h>` looks in system (library) directories; `#include "foo.h"` looks first in current directory, then system

C Preprocessor Example

- What do you think the preprocessor output will be?

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

Keep in mind:
1. Pre-processor goes line by line
2. builds up "state" as it processes directives

C Preprocessor Example

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c
```

C Preprocessor Example

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

Pre-processor state

FOO	1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c
```

C Preprocessor Example

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

Pre-processor state

FOO	1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c
```

C Preprocessor Example

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c
```

C Preprocessor Example

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c

typedef long long int verylong;
```

C Preprocessor Example

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
~~#define BAR 2 + FOO~~

typedef long long int verylong;
```

cpp_example.h

```
~~#define FOO 1~~

~~#include "cpp_example.h"~~

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c

typedef long long int verylong;
int main(int argc, char **argv) {
```

C Preprocessor Example

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c

typedef long long int verylong;
int main(int argc, char **argv) {
  int x = 1;
```

C Preprocessor Example

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c

typedef long long int verylong;
int main(int argc, char **argv) {
  int x = 1;
  int y = 2 + 1;
```

C Preprocessor Example

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c

typedef long long int verylong;
int main(int argc, char **argv) {
  int x = 1;
  int y = 2 + 1;
  verylong z = 1 + 2 + 1;
```

C Preprocessor Example

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use gcc -E)
  - "-P" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```c
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```c
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;  // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c

typedef long long int verylong;
int main(int argc, char **argv) {
  int x = 1;
  int y = 2 + 1;
  verylong z = 1 + 2 + 1;
  return 0;
}
```

Program Using a Linked List

Need definitions for code to compile

```c
#include <stdlib.h>
#include <assert.h>
#include "ll.h"

Node* Push(Node* head,
           void* element) {
  ... // implementation here
}
```

ll.c

```c
typedef struct node_st {
  void* element;
  struct node_st* next;
} Node;

Node* Push(Node* head,
           void* element);
```

ll.h

```c
#include "ll.h"

int main(int argc, char** argv) {
  Node* list = NULL;
  char* hi = "hello";
  char* bye = "goodbye";

  list = Push(list, (void*)hi);
  list = Push(list, (void*)bye);

  ...

  return 0;
}
```

example_ll_customer.c

Compiling the Program

- Four parts:
  - 1/2) Compile `example_ll_customer.c` into an object file
  - 2/1) Compile `ll.c` into an object file
  - 3) Link both object files into an executable
  - 4) Test, Debug, Rinse, Repeat

```
1 bash$ gcc -Wall -g -c example_ll_customer.c
2 bash$ gcc -Wall -g -c ll.c
3 bash$ gcc -g -o example_ll_customer ll.o example_ll_customer.o
4 bash$ ./example_ll_customer
  Payload: 'yo!'
  Payload: 'goodbye'
  Payload: 'hello'
4 bash$ valgrind -leak-check=full ./example_ll_customer
  ... etc ...
```

But There's a Problem with `#include`

- What happens when we compile `foo.c`?

```c
struct pair {
  int a, b;
};
```

pair.h

```c
#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);
```

util.h

```c
#include "pair.h"
#include "util.h"

int main(int argc, char** argv) {
  // do stuff here
  ...
  return 0;
}
```

foo.c

A Problem with `#include`

- What happens when we compile `foo.c`?

```
bash$ gcc -Wall -g -o foo foo.c
In file included from util.h:1:0,
                 from foo.c:2:
pair.h:1:8: error: redefinition of 'struct pair'
 struct pair { int a, b; };
        ^
In file included from foo.c:1:0:
pair.h:1:8: note: originally defined here
 struct pair { int a, b; };
        ^
```

- `foo.c` includes `pair.h` twice!
  - Second time is indirectly via `util.h`
  - Struct definition shows up twice
    - Can see using `cpp`

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state


```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

pair.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

util.h

foo.c

```
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state


```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

pair.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

util.h

foo.c

```
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state


```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

pair.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

util.h

foo.c

```
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state

PAIR_H_	defined

```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif  // PAIR_H_
```

pair.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif  // UTIL_H_
```

util.h

foo.c

```
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state

PAIR_H_	defined

```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

pair.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

util.h

foo.c

```
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state

PAIR_H_	defined

pair.h

```
#ifndef PAIR_H_
~~#define PAIR_H_~~

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

util.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

foo.c

```
~~#include "pair.h"~~
~~#include "util.h"~~

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (#define) in combination with conditional compilation (#ifndef and #endif)

Pre-processor state

PAIR_H_	defined

pair.h
```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

util.h
```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

foo.c
```
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state

PAIR_H_	defined
UTIL_H_	defined

pair.h
```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

util.h
```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

foo.c
```
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state

PAIR_H_	defined
UTIL_H_	defined

pair.h

```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

util.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

foo.c

```
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state

PAIR_H_	defined
UTIL_H_	defined

```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

pair.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

util.h

foo.c

```
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state

PAIR_H_	defined
UTIL_H_	defined

```c
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

pair.h

```c
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

util.h

foo.c

```c
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Header Guards

- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

Pre-processor state

PAIR_H_	defined
UTIL_H_	defined

```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

pair.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif   // UTIL_H_
```

util.h

foo.c

```
#include "pair.h"
#include "util.h"

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

Preprocessor Tricks: Constants

- A way to deal with "magic constants"

```
int globalbuffer[1000];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * 3.1415;
  *area = rad * 3.1415 * 3.1415;
}
```

Bad code
(littered with magic constants)

```
#define BUFSIZE 1000
#define PI 3.14159265359

int globalbuffer[BUFSIZE];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * PI;
  *area = rad * PI * PI;
}
```

Better code

Lecture Outline

- Memory Errors, Valgrind & gdb
- C Header Files & Modules
- C compilation, definitions vs declarations, CPP
- **Makefiles**

`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=c17 -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 run every time you change anything. 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; make it 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)

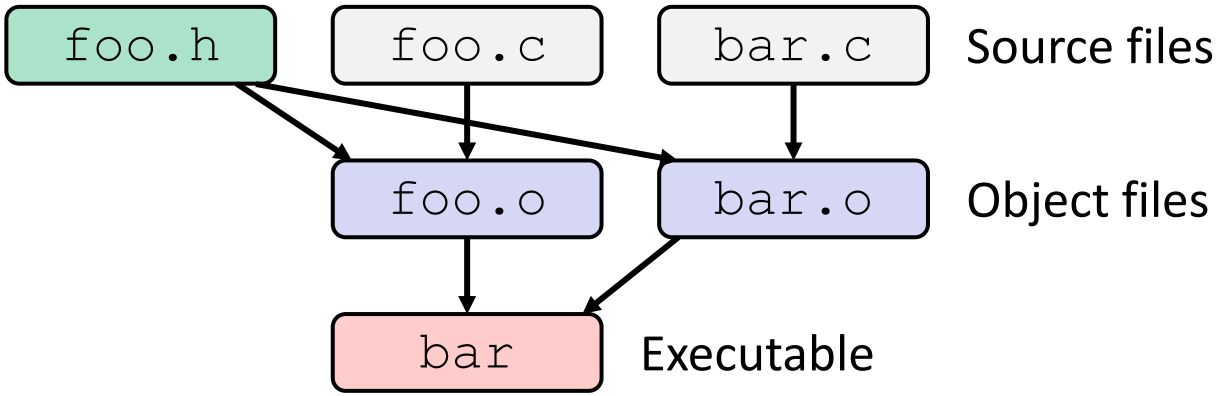
- A script can handle 1-3 (use a variable for filenames for 2), but 4 is trickier

Recompilation Management

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

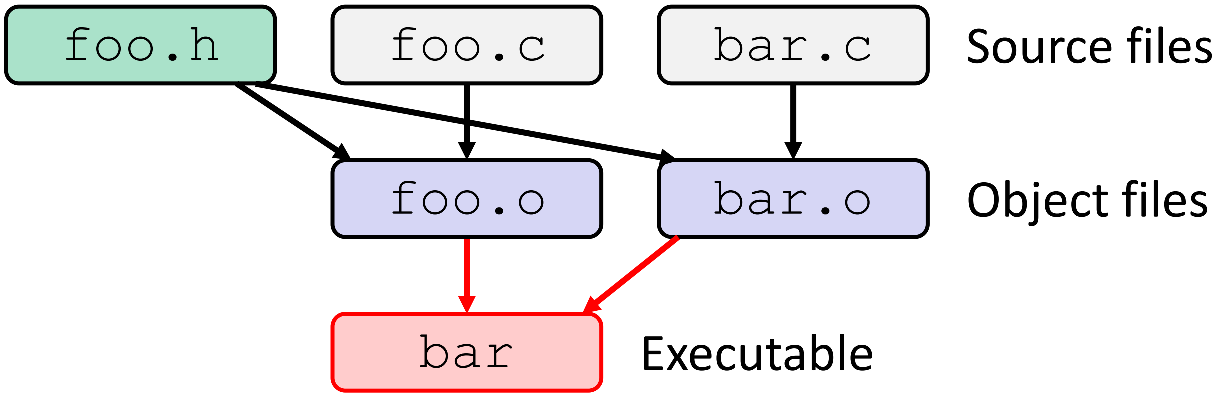
- To create a target ① t, you need sources ② $s_1, s_2, \ldots, s_n$ and a command ③ c that directly or indirectly uses the sources
  - If t is newer than every source (file-modification times), assume there is no reason to rebuild it
  - Recursive building: if some source s_i is itself a target for some other sources, see if it needs to be rebuilt…
  - Cycles "make no sense"!

Theory Applied to C



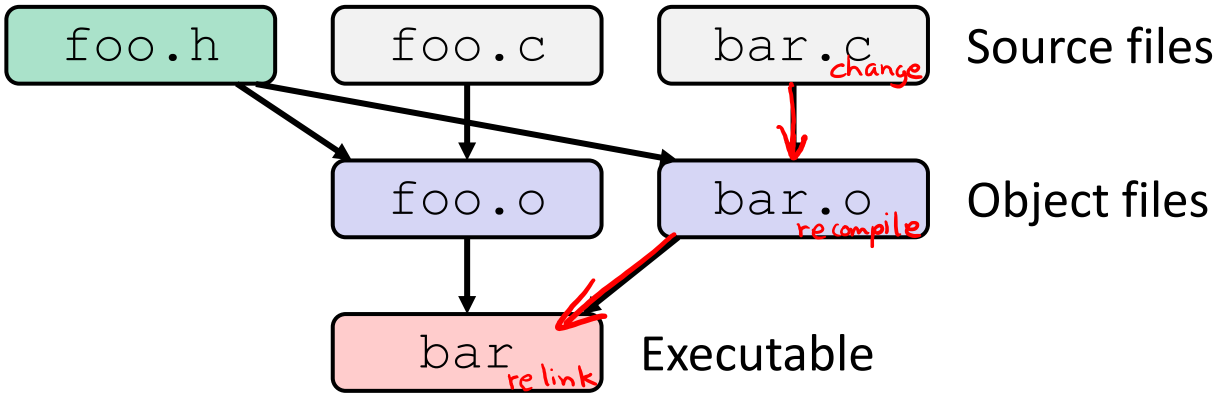
- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)

Theory Applied to C



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

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!

`make` Basics

- A makefile contains a bunch of triples:

```
①target: 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
```

Using `make`

```
bash$ make <target>
```

- Defaults:
  - If no `target` specified, will use the first one in the file
  - Will interpret commands in your default shell

- 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)

"Phony" Targets

- A make target whose command does not create a file of the target's name (*i.e.*, a "recipe")
  - As long as target file doesn't exist, the command(s) will be executed because the target must be "remade"

- *e.g.*, target `clean` is a convention to remove generated files to "start over" from just the source

```
clean:
        rm foo.o bar.o baz.o widget *~
```

- *e.g.*, target `all` is a convention to build all "final products" in the makefile
  - Lists all of the "final products" as sources

"all" Example

```
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

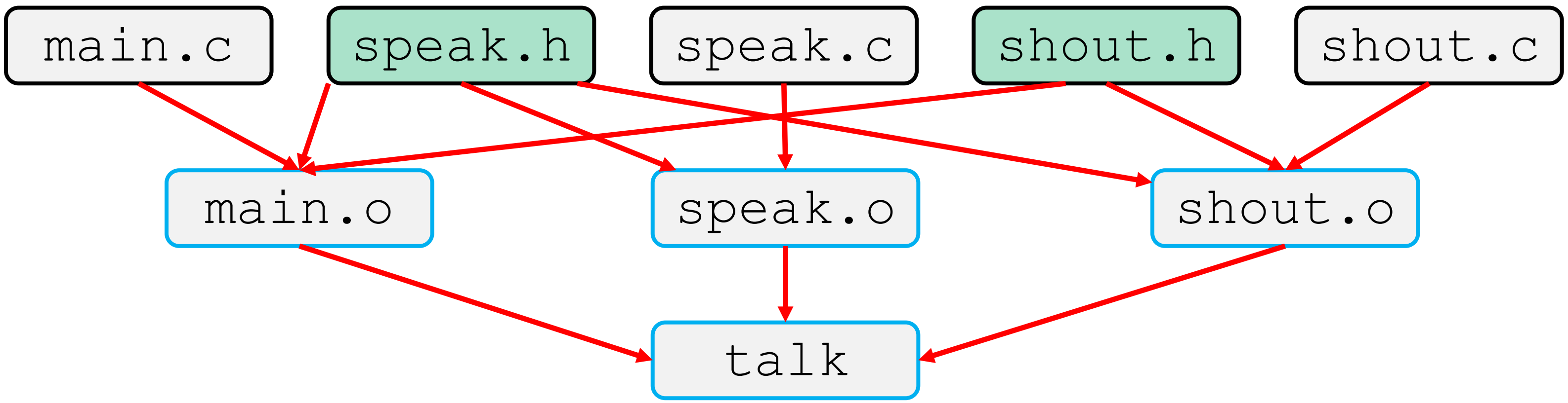
- *When creating a Makefile, first draw the dependencies!!!!*

- 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,” include an `all` rule
  - The first/top target should be your singular “final target” or `all`

Writing a Makefile Example

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



main.c

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

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

speak.c

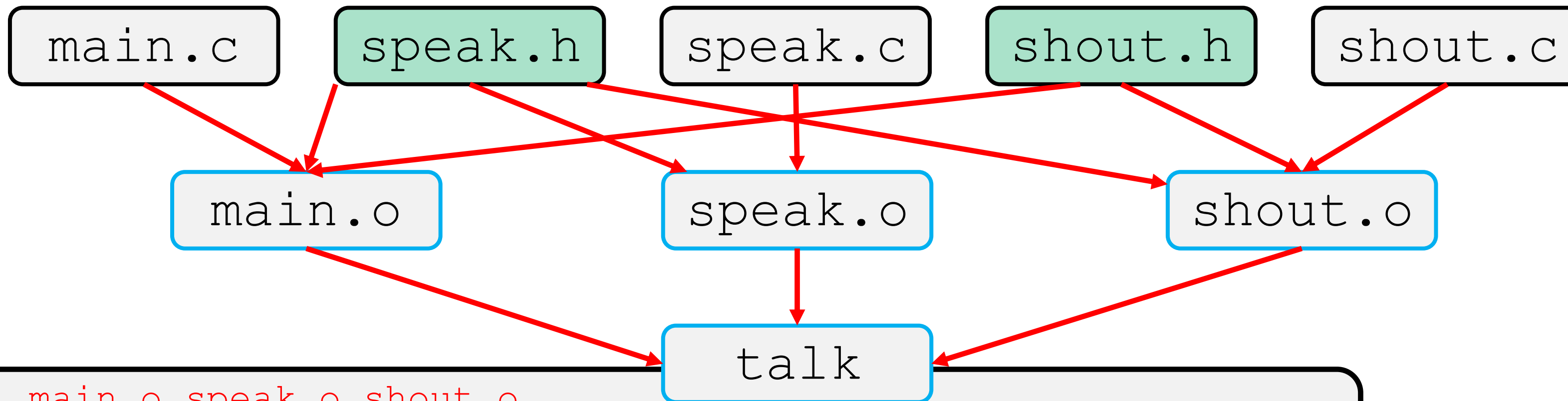
```
#include "speak.h"
...
```

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 -g -Wall -o talk main.o speak.o shout.o

main.o: main.c speak.h shout.h
        gcc -g -Wall -c main.c

speak.o: speak.c speak.h
        gcc -g -Wall -c speak.c

shout.o: shout.c speak.h shout.h
        gcc -g -Wall -c shout.c

clean:
        rm talk *.o
```

`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=c17
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
        $(CC) $(CFLAGS) -o widget $(OBJFILES)
```

- Advantages:
  - Easy to change things (especially in multiple commands)
    - It’s common to use variables to hold lists of filenames
  - Can also specify/overwrite variables on the command line: (*e.g.*, `make CC=clang CFLAGS=-g`)

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…)