C Wrap Up

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka Daniel Gearhardt
Ash Fujiyama Emily Shen

TAs:

Ahmed Abdellah
Angie Cao
August Fu
Caroline Begg
Cathy Cao
Claire Lu
Eric Sungwon Lee
Ethan Weisberg
Garrett O'Malley Kirsch
Hassan Rizwan
Iain Li
Jerry Wang
Juan Lopez
Keith Mathe
Maya Huizar
Meghana Vasireddy
Perrie Quek
Sidharth Roy
Sydnie-Shea Cohen
Vivi Li
Yousef AlRabiah

pollev.com/cis2400

- How are you? Any Questions from last lecture?

Upcoming Due Dates

- HW01 (Approx): Due This Past Friday
  - Remember, you can do it ***later in the semester***
  - don't fall behind please
- HW02 (C Strings!): Due This Friday
  - If you didn't get c-strings and how to use the heap you will after this!
- ***The lecture check was due right before lecture***!
  - No extensions except under extenuating circumstances
  - "I forgot" is not extenuating circumstances
  - ***They are always due right before Tuesday's Lecture!***

Recitation

- Wednesday's 4PM – 5PM
  - DRL 3C6
- Had about 10-15 Students…not bad
- Super useful if you need any extra help or want to review topics
- GDB, Valgrind, Makefiles, and etc. covered this week!
- There isn't enough time in lecture to cover these things in more depth; please go if you are still confused!

Lecture Outline

- Building a C-Program
  - Preprocessor
  - Single Program
  - Multi-File Program (Linking

Lecture Outline

- Building a C-Program
  - Preprocessor
  - Single Program
  - Multi-File Program (Linking)
- Recompilation
- Makefiles
  - Basics, Targets, Rules
  - Variables
  - Phony Targets
- Files (i/o)
  - Reading and writing to the terminal
- Generics
  - Void *

Building a C Program

```c
#define ten 10

int add(int x, int y, int z){
    return x + y + z;
}
int main(int argc, char *argv[]){
    int x = ten;
    int y = 11;
    int z = 12;
    int sum = ten; //comment here
    return 0;
}
```

small.c

- We can now compile C-programs from scratch!
- But what really happens during compilation?

```
clang-15 -Wall -save-temps -g3 -o small small.c
```

- -Wall
  - All warnings! (Produced the ~(!x) warning in HW02)
- -g / -g3
  - Compile and make it debuggable (-g3 means MAX debug info)

Building a C Program

```c
#define ten 10

int add(int x, int y, int z){
    return x + y + z;
}
int main(int argc, char *argv[]){
    int x = ten;
    int y = 11;
    int z = 12;
    int sum = ten; //comment here
    return 0;
}
```

small.c

- We can now compile C-programs from scratch!
- But what really happens during compilation?

```
clang-15 -Wall -save-temps -g3 -o small small.c
```

- -save-temps
  - Saves all temporary files created during compilation

Building a C Program

```c
#define ten 10

int add(int x, int y, int z){
    return x + y + z;
}
int main(int argc, char *argv[]){
    int x = ten;
    int y = 11;
    int z = 12;
    int sum = ten; //comment here
    return 0;
}
```

small.c

- -save-temps
  - Saves all temporary files created during compilation

```
-rwxr-xr-x 1 root root 10472 Sep 16 11:43 small
-rw-r--r-- 1 root root  2868 Sep 16 11:43 small.bc
-rw-r--r-- 1 root root   169 Sep 16 11:42 small.c
-rw-r--r-- 1 root root   255 Sep 16 11:43 small.i
-rw-r--r-- 1 root root  3328 Sep 16 11:43 small.o
-rw-r--r-- 1 root root 13499 Sep 16 11:43 small.s
```

C Preprocessor: small.i

- The C preprocessor is run over each source file in your program before it is passed to the compilation stage
  - The preprocessor stage handles various tedious text based operations.
- Some common preprocessor commands
  - #define – substitute the string literal with the following string
  - #include – literally include a file at this location

C Preprocessor: small.i

- Conditional compilation
  - you can include or exclude parts of the program according to various conditions.

```
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN

 the entire file

#endif /* FILE_FOO_SEEN */
```

The C preprocessor is programmed to notice when a header file uses this particular construct and handle it efficiently.

If a header file is contained entirely in a ”#ifndef” conditional, then it ***records that fact***.

If a subsequent ”#include” specifies the same file, and the macro in the ”#ifndef” is already defined, then the file is entirely skipped, without even reading it.

`#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 str 'hi!'` defines a symbol (a string!) and replaces later occurrences
    - Several others that we'll see soon…
  - Run on your behalf by `clang` during compilation
  - Note: `#include <foo.h>` looks in system (library) directories; `#include "foo.h"` looks first in current directory, then system

#include and the C Preprocessor

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `clang -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 Changes

small.c

```c
#define ten 10

int add(int x, int y, int z){
    return x + y + z;
}
int main(int argc, char *argv[]){
    int x = ten;
    int y = 11;
    int z = 12;
    int sum = ten; //comment here
    return 0;
}
```

small.i

```c
int add(int x, int y, int z){
    return x + y + z;
}
int main(int argc, char *argv[]){
    int x = 10;
    int y = 11;
    int z = 12;
    int sum = add(x, y, z);
    return 0;
}
```

Define macro is gone

'ten' is replaced with defined value

Comment is gone

Poll Everywhere

pollev.com/cis2400

- How many changes will be made after the preprocessor step?

```c
// Define a constant
#define MAX_SIZE 100

/* Define a macro for a greeting message */
#define GREETING "Hello, World!"

int main() {
    // Print a greeting message
    printf("%s\n", GREETING);

    // Print the maximum size
    printf("The maximum size is: %d\n", MAX_SIZE);

    /*
     * why did I become a programmer? Pls help.
     */

    return 0;
}
```

A. 5

B. 9

C. 10

D. 8

E. wtf

Poll Everywhere

pollev.com/cis2400

- How many changes will be made after the preprocessor step?

```c
// Define a constant
#define MAX_SIZE 100

/* Define a macro for a greeting message */
#define GREETING "Hello, World!"

int main() {
    // Print a greeting message
    printf("%s\n", GREETING);

    // Print the maximum size
    printf("The maximum size is: %d\n", MAX_SIZE);

    /*
     * why did I become a programmer? Pls help.
     */

    return 0;
}
```

A. 5

B. 9

C. 10

D. 8

E. wtf

Preprocessor: #include <stdlib.h>

- Including standard libraries instructs the preprocessor to *literally include the entire header file.*
- *Let's see this in action.*

Quick Aside: small.s

- The Assembly Language version of the file.
  - X86 (Intel Processors, e.g.)
  - Arm (M1 Chips, Raspberry Pi's, etc)
- We'll talk more about Assembly very soon!
  - (in a couple of weeks)

```
main:
.Lfunc_begin1:
        .loc    1 6 0 is_stmt 1
        .cfi_startproc
// %bb.0:
        sub     sp, sp, #64
        .cfi_def_cfa_offset 64
        stp     x29, x30, [sp, #48]
        add     x29, sp, #48
        .cfi_def_cfa w29, 16
        .cfi_offset w30, -8
        .cfi_offset w29, -16
        mov     w8, wzr
        str     w8, [sp, #12]
        stur    wzr, [x29, #-4]
        stur    w0, [x29, #-8]
        stur    x1, [x29, #-16]
        mov     w8, #10
```

Yup, it's ugly. I know

Multiple File Compilation (Linking)

- More complex programs require multiple files.
- Even printing to the terminal requires the import of the standard library
  - ***#include <stdio.h>*** `for` ***printf***
- Your own string library implementations will be ***#included*** in files which refer to those functions.

Multiple File Compilation (Linking)

utils.h

```
#ifndef UTILS_H
#define UTILS_H
```

utils.c

```
#include ”utils.h"

int fun1(int x, int y){
}
int fun2(int x, int y){
  ...
}
```

- Helper libraries don’t have a main function to “enter from”.
- These .c and .h files can still be partially compiled to have code for the subroutines defined here.
  - -c flag is used for partial compilation
  - ***clang-15 –c utils.c***
- This creates .o or .obj files.

Multiple File Compilation (Linking)

utils.h

```
#ifndef UTILS_H
#define UTILS_H
```

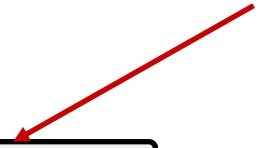
utils.c

```
#include "utils.h"

int fun1(int x, int y){
}
int fun2(int x, int y){
  ...
}
```

prog.c

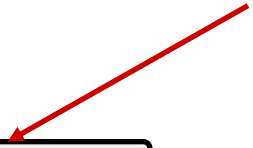
```
#include "utils.h"

int main(int argc, char *argv[]){
    fun1(10, 11);
    fun2(11, 12);
}
```

clang-15 –c utils.c



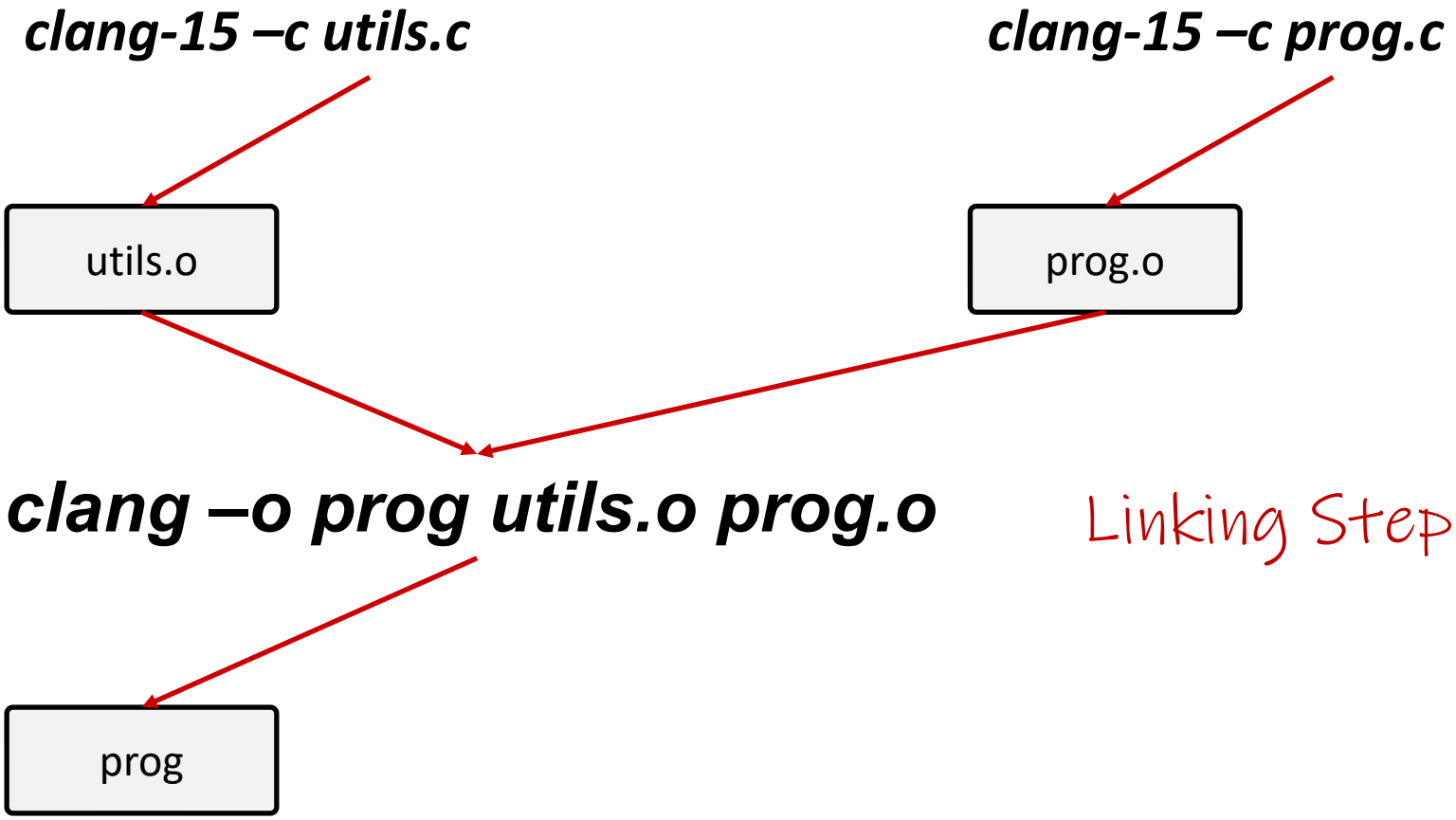
utils.o

clang-15 –c prog.c



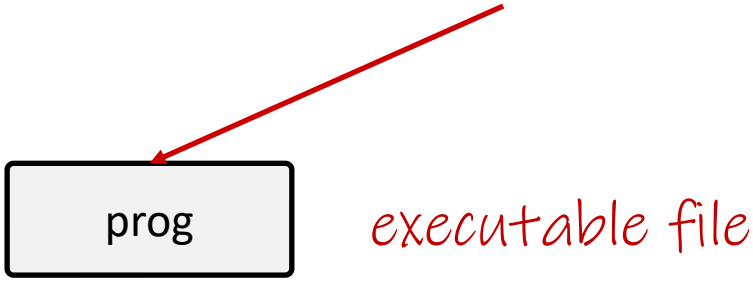
prog.o

Multiple File Compilation (Linking)



Multiple File Compilation (Linking)

clang –o prog utils.o prog.o Linking Step



- a program is combined with all of the libraries it calls to produce the final executable program.
- To do this the linker needs to find the code for every function that is called in the program and ***link*** them together appropriately to produce the final executable binary file.

Lecture Outline

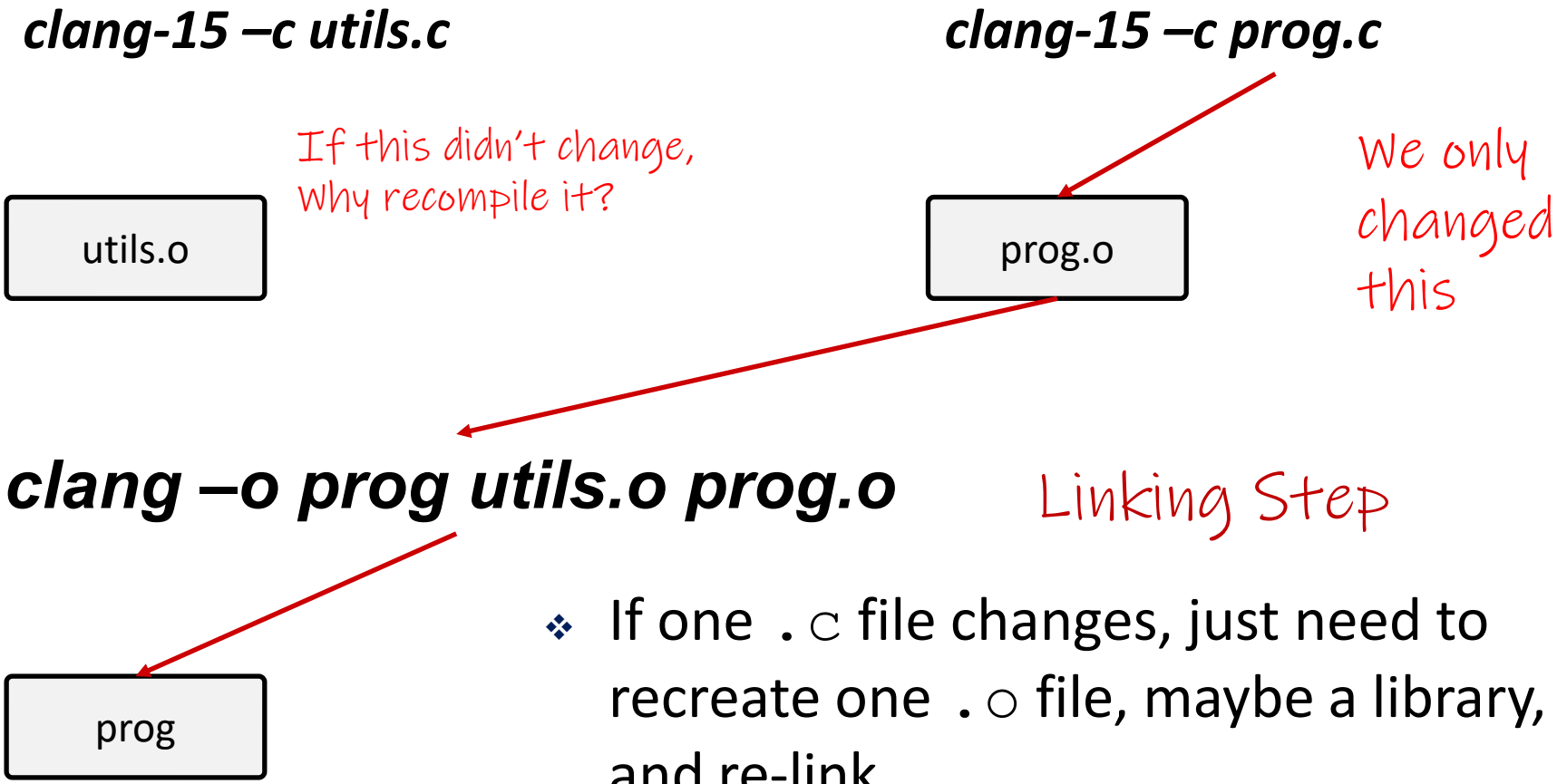
- Building a C-Program
  - Preprocessor
  - Single Program
  - Multi-File Program (Linking)
- Recompilation
- Makefiles
  - Basics, Targets, Rules
  - Variables
  - Phony Targets
- Files (i/o)
  - Reading and writing to the terminal
- Generics
  - Void *

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
  - 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"!

Recompilation



- 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

Lecture Outline

- Building a C-Program
  - Preprocessor
  - Single Program
  - Multi-File Program (Linking)
- Recompilation
- Makefiles
  - Basics, Targets, Rules
  - Variables
  - Phony Targets
- Files (i/o)
  - Reading and writing to the terminal
- Generics
  - Void *

Makefiles

- The ***make*** tool was used to simplify the development and building of multiple files.
- Allows you to specify dependencies between source files in your project and indicates how they should be compiled to produce the ***final result***
- Think of the Makefile as a recipe for building your final program from the source file

make Basics

- 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:

```
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)

Makefile Writing Tips

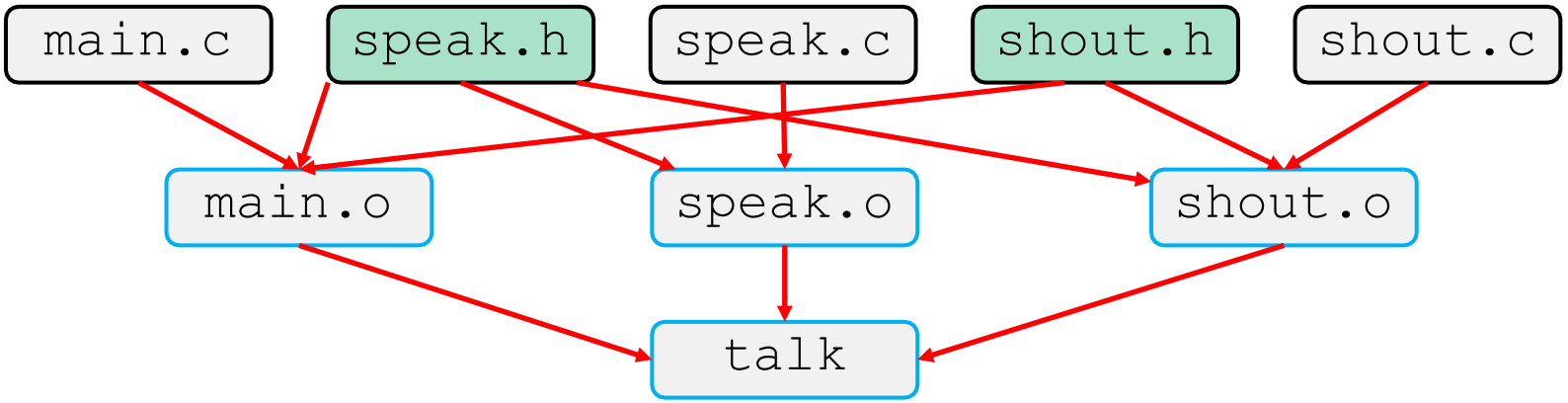
- *When creating a Makefile, first draw the dependencies!!!!*

- 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



main.c

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

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

speak.c

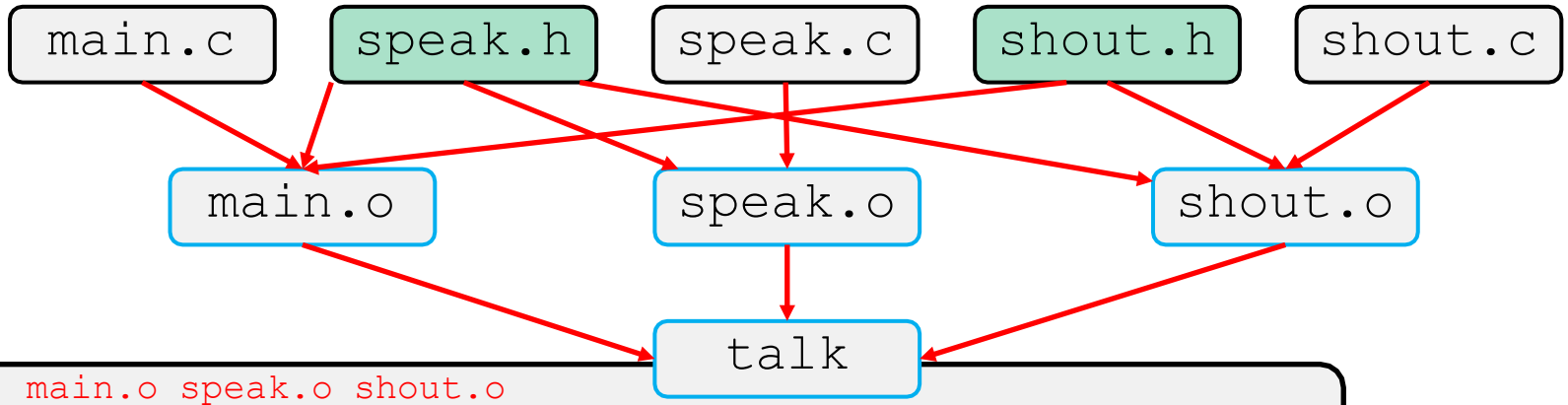
```
#include "speak.h"
...
```

shout.c

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

Writing a Makefile Example

- "`talk`" program




```
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
```

Makefiles

A "simple" Makefile for our previous example.

utils.h

```
#ifndef UTILS_H
#define UTILS_H
```

utils.c

```
#include "utils.h"

int fun1(int x, int y){
}
int fun2(int x, int y){
  ...
}
```

prog.c

```
#include "utils.h"

Int main(int argc, char *argv[]){
    fun1(10, 11);
    fun2(11, 12);
}
```

Makefiles

A "simple" Makefile for our previous example.

utils.h

```
#ifndef UTILS_H
#define UTILS_H
...
#endif
```

utils.c

```
#include "utils.h"

int fun1(int x, int y){
}
int fun2(int x, int y){
  ...
}
```

prog.c

```
#include "utils.h"

Int main(int argc, char *argv[]){
    fun1(10, 11);
    fun2(11, 12);
}
```

makefile

```
utils.o: utils.c utils.h
    clang -c utils.c

prog.o: prog.c utils.h
    clang -c prog.c

prog: prog.o utils.o
    clang -o prog prog.o utils.o

all: prog
```

Makefiles

A "simple" Makefile for our previous example.

makefile

```
utils.o: utils.c utils.h
    clang -c utils.c

prog.o: prog.c utils.h
    clang -c prog.c

prog: prog.o utils.o
    clang -o prog prog.o utils.o

all: prog
```

- This line specifies that **utils.o** depends on **utils.c** and **utils.h**.
- If either utils.c or utils.h is modified, utils.o will ***need*** to be recompiled.
- This saves time by not recompiling files that don't need to be recompiled.

Makefiles

A "simple" Makefile for our previous example.

makefile

```
utils.o: utils.c utils.h
    clang -c utils.c

prog.o: prog.c utils.h
    clang -c prog.c

prog: prog.o utils.o
    clang -o prog prog.o utils.o

all: prog
```

- This line compiles our first object file, ***utils.o.***

Makefiles

A "simple" Makefile for our previous example.

makefile

```
utils.o: utils.c utils.h
    clang -c utils.c

prog.o: prog.c utils.h
    clang -c prog.c

prog: prog.o utils.o
    clang -o prog prog.o utils.o

all: prog
```

- Same as before, prog.o depends on prog.c and utils.h.
- ***Then we partially compile prog.c into prog.o***

Makefiles

A "simple" Makefile for our previous example.

makefile

```
utils.o: utils.c utils.h
    clang -c utils.c

prog.o: prog.c utils.h
    clang -c prog.c

prog: prog.o utils.o
    clang -o prog prog.o utils.o

all: prog
```

- Finally, our last step where we link all object files together.
- The final executable **prog** depends on prog.o and utils.o.
- **If either of these changed before hand, we relink.**

Makefiles

A "simple" Makefile for our previous example.

makefile

```
utils.o: utils.c utils.h
    clang -c utils.c

prog.o: prog.c utils.h
    clang -c prog.c

prog: prog.o utils.o
    clang -o prog prog.o utils.o

all: prog
```

- When you type "make", it will create the file followed after the ***all***.
- So it will create ***prog***!

`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`)

"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

Makefiles

A "more complex" Makefile for our previous example.

makefile

```
# Define the compiler and flags
CC = clang
CFLAGS = -Wall -g

utils.o: utils.c utils.h
    $(CC) $(CFLAGS) -c utils.c

prog.o: prog.c utils.h
    $(CC) $(CFLAGS) -c prog.c

prog: prog.o utils.o
    $(CC) -o prog prog.o utils.o

all: prog

clean:
    rm -f prog *.o
```

- Here we define some **constants** or **macros** that we can refer to for the rest of the Makefile.
- Allows you to quickly change compiler name in every line or the flags you want to use.
- Every $(macro) is replaced with the corresponding defined constants.

Makefiles

A "more complex" Makefile for our previous example.

makefile

```
# Define the compiler and flags
CC = clang
CFLAGS = -Wall -g

utils.o: utils.c utils.h
    $(CC) $(CFLAGS) -c utils.c

prog.o: prog.c utils.h
    $(CC) $(CFLAGS) -c prog.c

prog: prog.o utils.o
    $(CC) -o prog prog.o utils.o

all: prog

clean:
    rm -f prog *.o
```

- A new phony target: 'clean'
- Typing 'make clean' runs this.
- Removes all generated .o files and the final 'prog' executable created.

rm stands for 'remove' and the flag '-f' means forcefully.

Revenge of the Funny Characters

- Special variables:
  - **$@** for target name
  - **$^** for all sources
  - **$<** for left-most source (first 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 $<
```

Makefiles

A complex Makefile for our previous example.

makefile

```
# Define the compiler and flags
CC = clang
CFLAGS = -Wall -g

# Define the target executable and dependencies
TARGET = prog
DEPS = utils.h

# Pattern rule for compiling .c files into .o files

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

$(TARGET): prog.o utils.o
    $(CC) -o $@ $^

all: $(TARGET)

clean:
    rm -f prog *.o
```

$@ - The Target Filename

prog: prog.o utils.o
$@ would refer to prog.

Makefiles

A complex Makefile for our previous example.

makefile

```
# Define the compiler and flags
CC = clang
CFLAGS = -Wall -g

# Define the target executable and dependencies
TARGET = prog
DEPS = utils.h

# Pattern rule for compiling .c files into .o files

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

$(TARGET): prog.o utils.o
    $(CC) -o $@ $^

all: $(TARGET)

clean:
    rm -f prog *.o
```

$@ - The Target Filename

prog: prog.o utils.o
$@ would refer to prog.

$^ - All Source Files

prog: prog.o utils.o
$^ would refer to prog.o utils.o
.

clang -o prog prog.o utils.o.

Makefiles

A complex Makefile for our previous example.

makefile

```
# Define the compiler and flags
CC = clang
CFLAGS = -Wall -g

# Define the target executable and dependencies
TARGET = prog
DEPS = utils.h

# Pattern rule for compiling .c files into .o files

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

$(TARGET): prog.o utils.o
    $(CC) -o $@ $^

all: $(TARGET)

clean:
    rm -f prog *.o
```

%.o –
This matches any file ending in .o
It means "any object file."

%.c –
This matches any file ending in .c
It means "any c file."

Note: they will always match
e.g. foo.o : foo.c

$<
This means the first dependent file in the list.

clang -Wall -g -c utils.c -o utils.o

And more…

- There are a lot of "built-in" rules – see documentation
- 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…)

Lecture Outline

- Building a C-Program
  - Preprocessor
  - Single Program
  - Multi-File Program (Linking)
- Recompilation
- Makefiles
  - Basics, Targets, Rules
  - Variables
  - Phony Targets
- Files (i/o)
  - Reading and writing to the terminal
- Generics
  - Void *

Files

- In C files are very ***simple*** objects – they consist of a sequence of bytes.
  - Can be ACII values, UTF-8, machine code to execute, the name of the person who broke your heart last summer in the note's app of your macbook pro, really anything: All up to interpretation.

In general, files start off 'closed'.

Files

- To read from a file (i.e. to see what it contains)
  - You need to open it first.
  - Then you can read from it
- To write to a file (i.e. change its contents or add to it)
  - You need to open() it first.
  - Then you can write() to it.
- When done reading/writing to a file
  - You need to then close() it (in some cases…)

Files: How to refer to them?

- In Unix and Unix like systems, ***File ** are pointers that refer to ***file handlers***
- You can use these to refer to ***files*** that you've opened.
- Some files are already open for you when you run your program.

The terminal itself is treated as a file

- you can "read from" and "write to" it

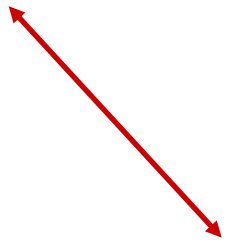
- Three file handlers your programs will always have
  - stdin : standard-input (terminal)
  - stdout : standard-output (terminal, for output)
  - stderr : standard-error (terminal, for error message

Reading and Writing from Terminal

stdin, stdout, and stderr

defined in `#include <stdio.h>`

- `printf("hi\n")` and `fprintf(stdout,“hi\n”);`
  - These are equivalent!
  - The f in fprintf stands for file.



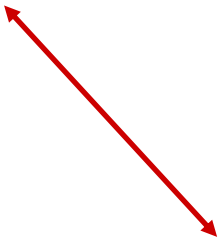
Here, we specify the File Handler to write to.

Reading and Writing from Terminal

stdin, stdout, and stderr

defined in `#include <stdio.h>`

- `fprintf(stderr,“Your progeram failed.\n”);`



Here, we specify to use the stderr **File Handler** to indicate an error.

Reading and Writing from Terminal

Reading from stdin is not so straight forward.

```
int getline(char ** line_buf, int *num_bytes, FILE *handler)
```

- `line_buf`
  - The ***address of char * you declare***.
  - getline will modify this pointer to hold a pointer to a ***heap allocated*** array that holds the line that was read if line_buf is null.
- `num_bytes`
  - The address of an int you declare.
  - num_bytes is the length of the array you pass in via line_buf
  - If it is 0, it will be updated to the size of the array it heap allocates for you.

Reading and Writing from Terminal

Reading from stdin is not so straight forward.

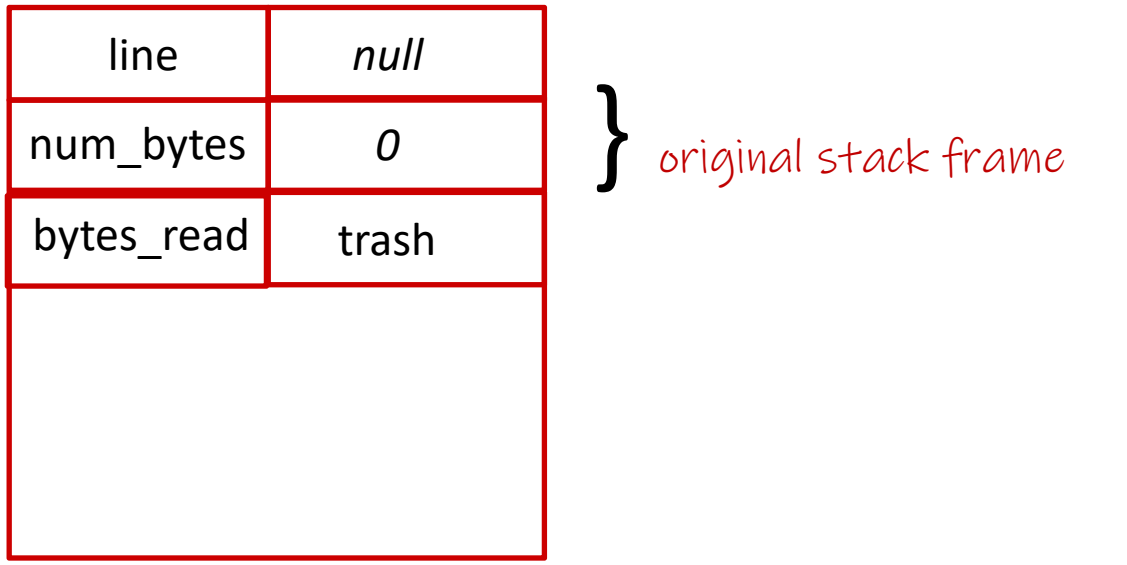
```
int getline(char ** line_buf, int *num_bytes, FILE *handler)
```

- `handler`
  - The file handler / file you want to "read from"
  - For us, *stdin* is the file handler to grab input from terminal.
- Returns the number of bytes put in line_buf.

Reading and Writing from Terminal

int getline(char ** line_buf, int *num_bytes, FILE *handler)

```
char *line = NULL;
int num_bytes = 0;
int bytes_read = getline(&line, &num_bytes, stdin));

//should check to see if line is non null first…

free(line); //free the line when done with it.
```

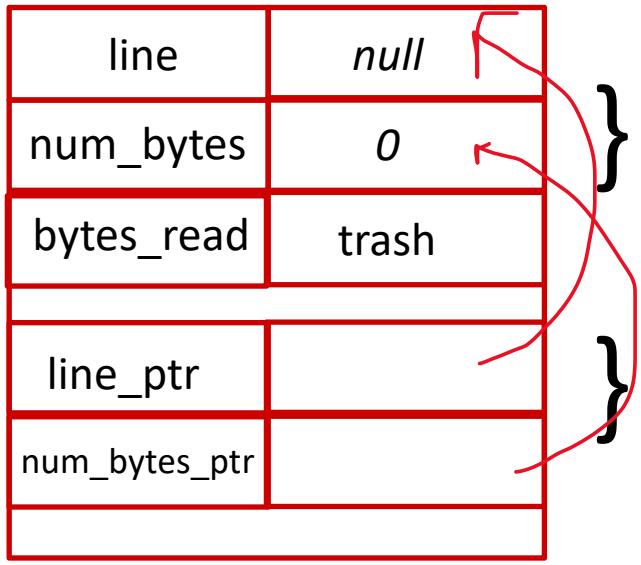


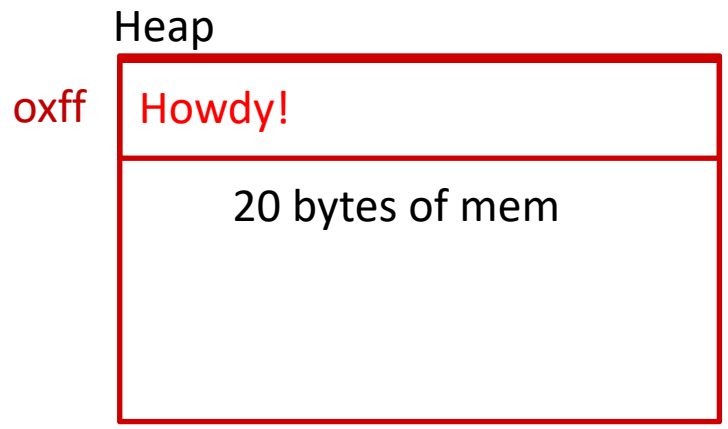
Reading and Writing from Terminal

int getline(char ** line_buf, int *num_bytes, FILE *handler)

```
char *line = NULL;
int num_bytes = 0;
int bytes_read = getline(&line, &num_bytes, stdin));

//should check to see if line is non null first…

free(line); //free the line when done with it.
```

You type "howdy!" in terminal

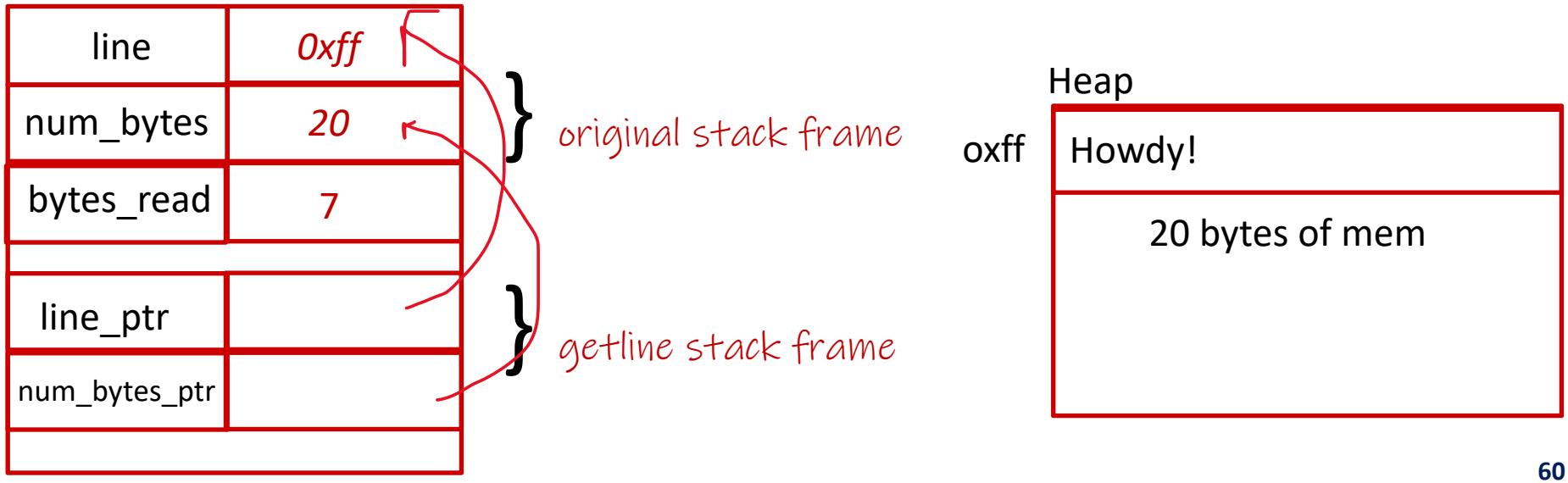




Reading and Writing from Terminal

int getline(char ** line_buf, int *num_bytes, FILE *handler)

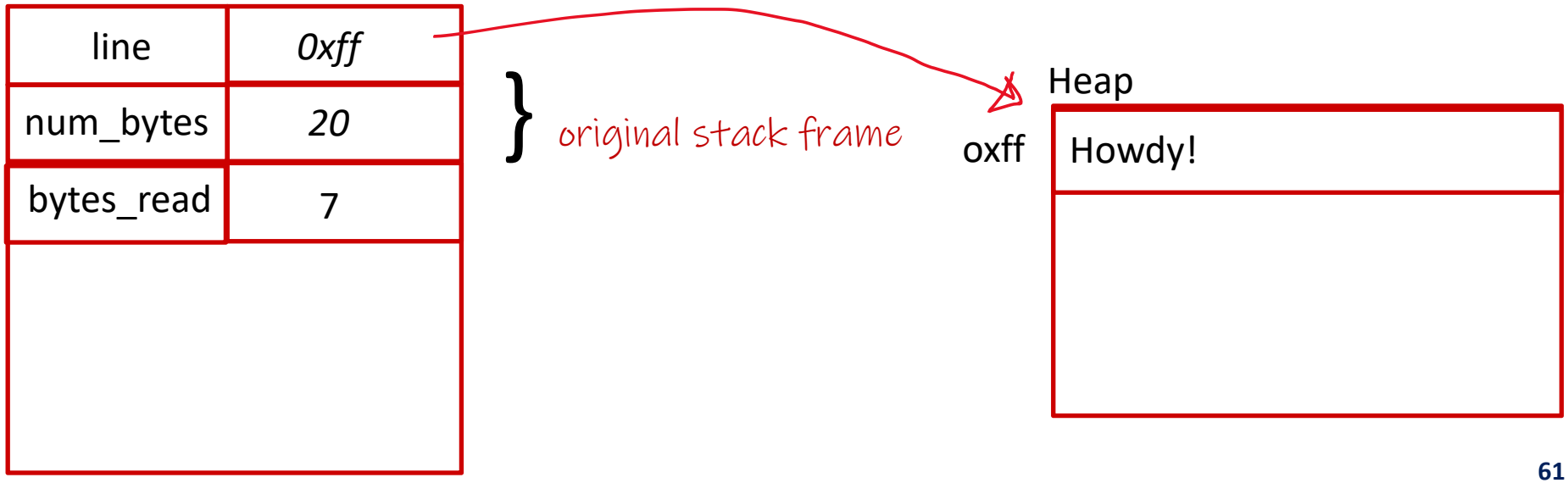
```
char *line = NULL;
int num_bytes = 0;
int bytes_read = getline(&line, &num_bytes, stdin));

//should check to see if line is non null first…

free(line); //free the line when done with it.
```



Reading and Writing from Terminal

int getline(char ** line_buf, int *num_bytes, FILE *handler)

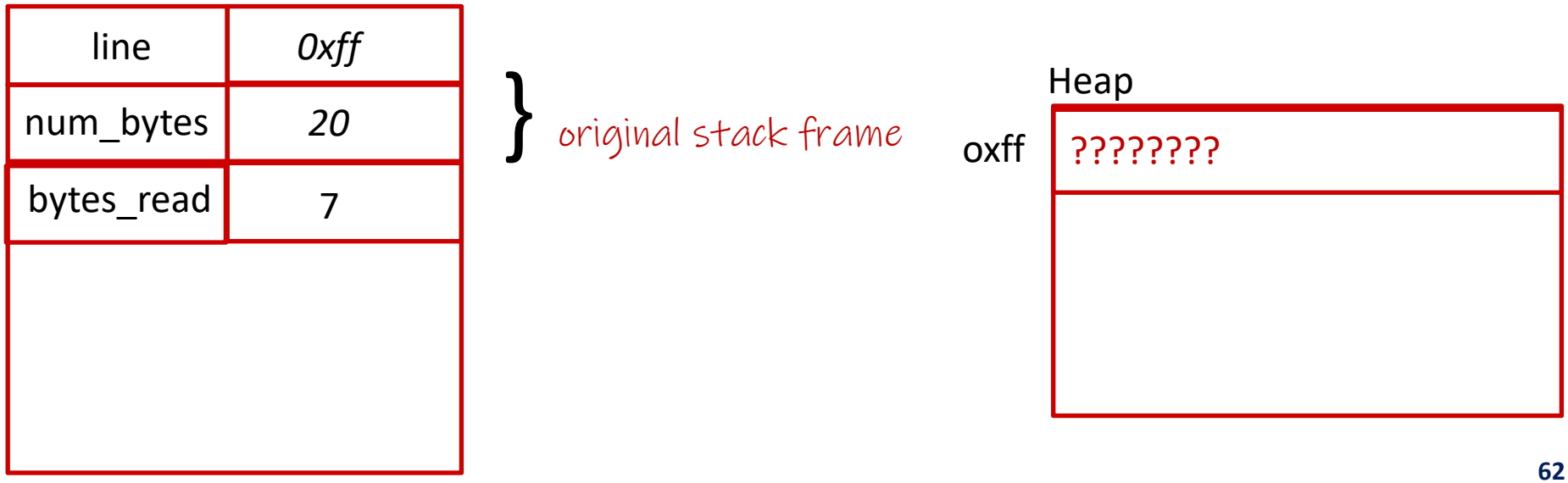
```
char *line = NULL;
int num_bytes = 0;
int bytes_read = getline(&line, &num_bytes, stdin));

//should check to see if line is non null first…

free(line); //free the line when done with it.
```



Reading and Writing from Terminal

int getline(char ** line_buf, int *num_bytes, FILE *handler)

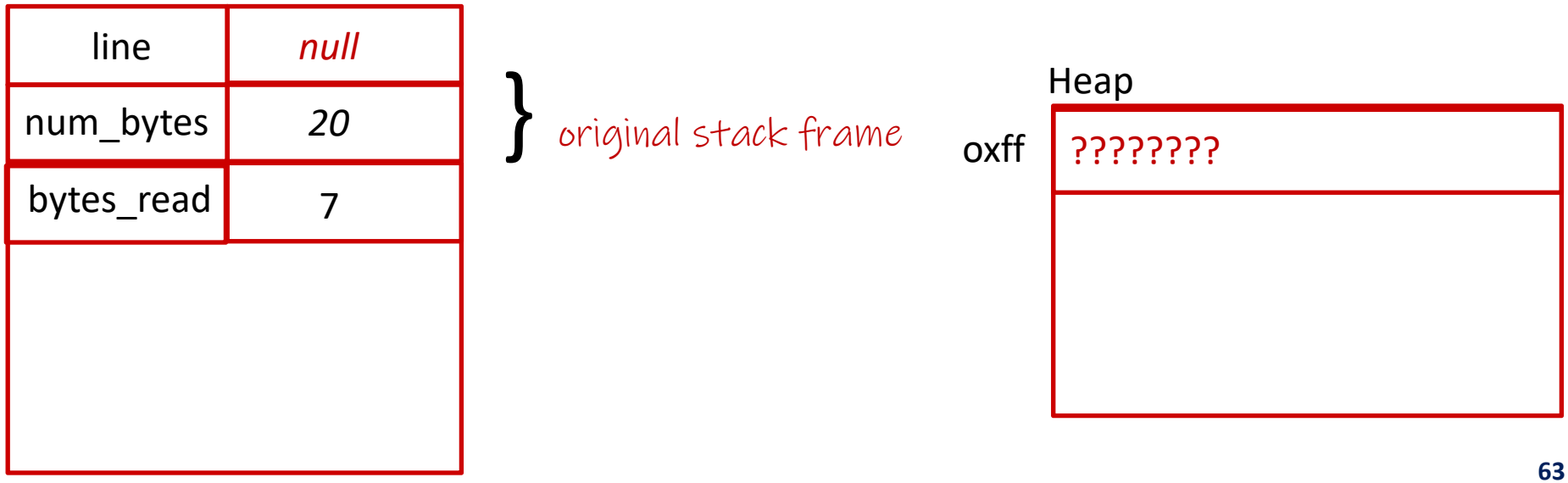
```
char *line = NULL;
int num_bytes = 0;
int bytes_read = getline(&line, &num_bytes, stdin));

//should check to see if line is non null first…

free(line); //free the line when done with it and set to null
line = null;
```



Reading and Writing from Terminal

int getline(char ** line_buf, int *num_bytes, FILE *handler)

```
char *line = NULL;
int num_bytes = 0;
int bytes_read = getline(&line, &num_bytes, stdin));

//should check to see if line is non null first…

free(line); //free the line when done with it and set to null
line = null;
```



Reading Variables from Format String

- **int sscanf(char *_str_, char * _format_, ...);**
  - Allows us to insert values from a string into corresponding variables.
  - Best shown by example.

Reading Variables from Format String

- **int sscanf(char *_str_, char * _format_, ...);**

```
int main() {
    char str[] = "123 45.6 hello";
    int num1;
    float num2;
    char word[20];

    // Read formatted input from the string
    int count = sscanf(str, "%d %f %s", &num1, &num2, word);
    ...
}
```

- After this
  - num1 will be 123
  - num2 will be 45.6
  - word will be “hello”
- count will be the number of variable correctly assigned
  - here it will be 3

stdio.h

- There are many more functions in the standard library
- Check them out at your own discretion…

Lecture Outline

- Building a C-Program
  - Preprocessor
  - Single Program
  - Multi-File Program (Linking)
- Recompilation
- Makefiles
  - Basics, Targets, Rules
  - Variables
  - Phony Targets
- Files (i/o)
  - Reading and writing to the terminal
- Generics
  - Void *

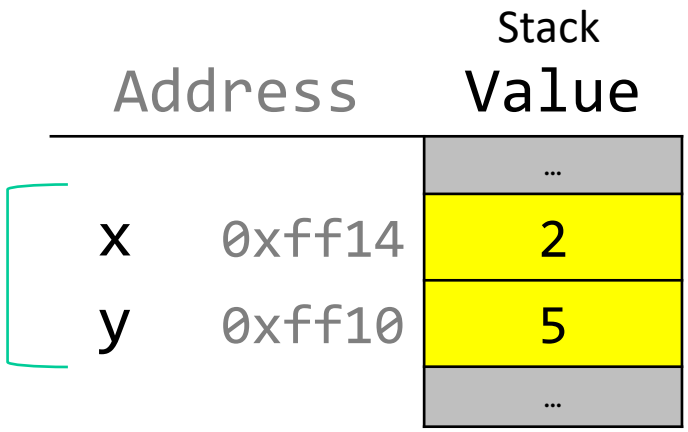
Generics

- We always want to write code that is as general-purpose as possible.
- Generic code reduces code duplication and means you can make improvements and fix bugs in one place rather than many.
- Generics is used throughout C for functions to sort any array, search any array, free arbitrary memory, and more.
- How can we write generic code in C? (Pointers are key!)

Void *: Generic Swap

- You're asked to write a function that swaps two ints.

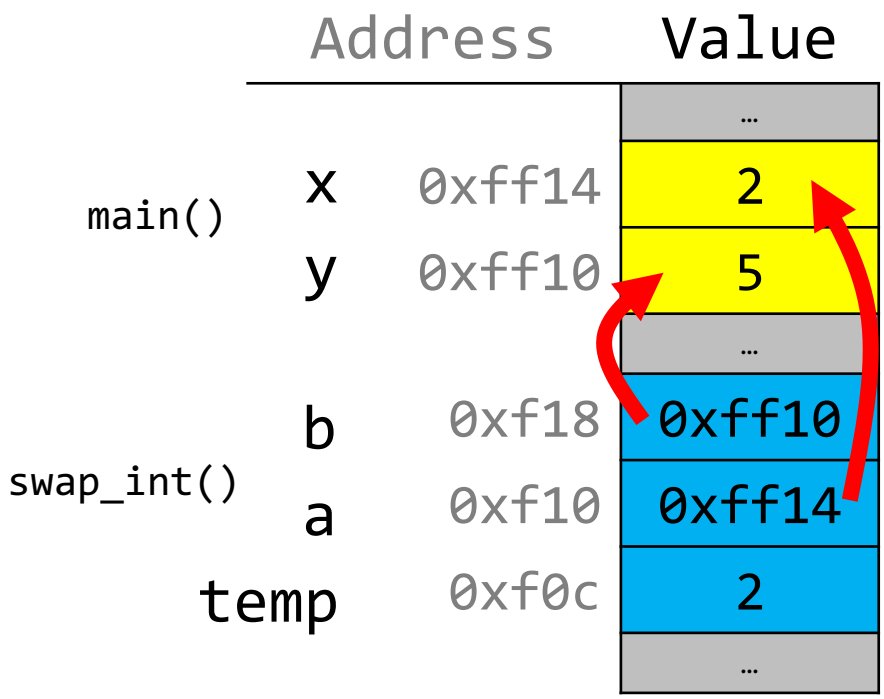
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```



Void *: Generic Swap

- You're asked to write a function that swaps two ints.

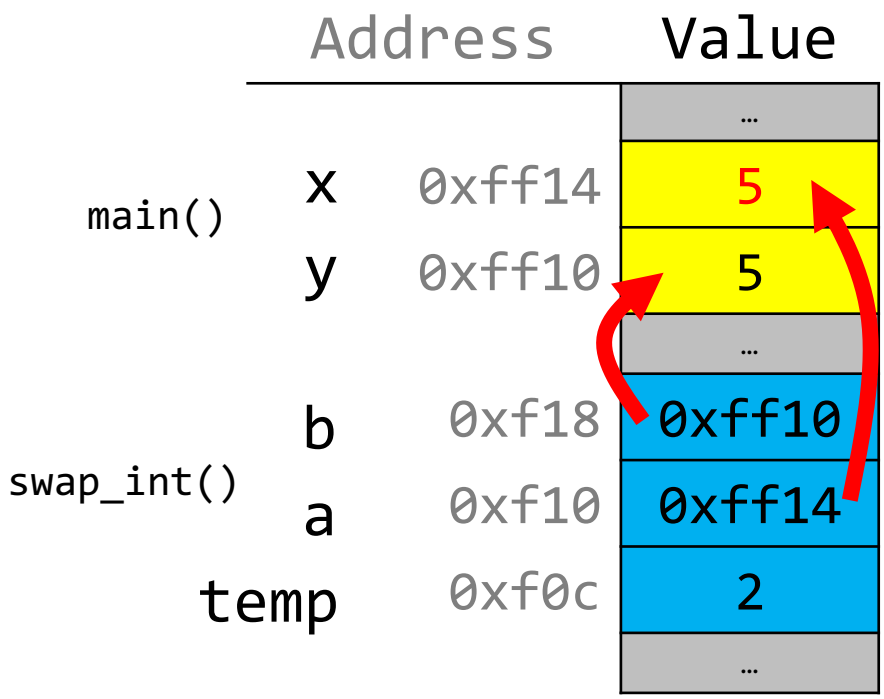
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```



Void *: Generic Swap

- You're asked to write a function that swaps two ints.

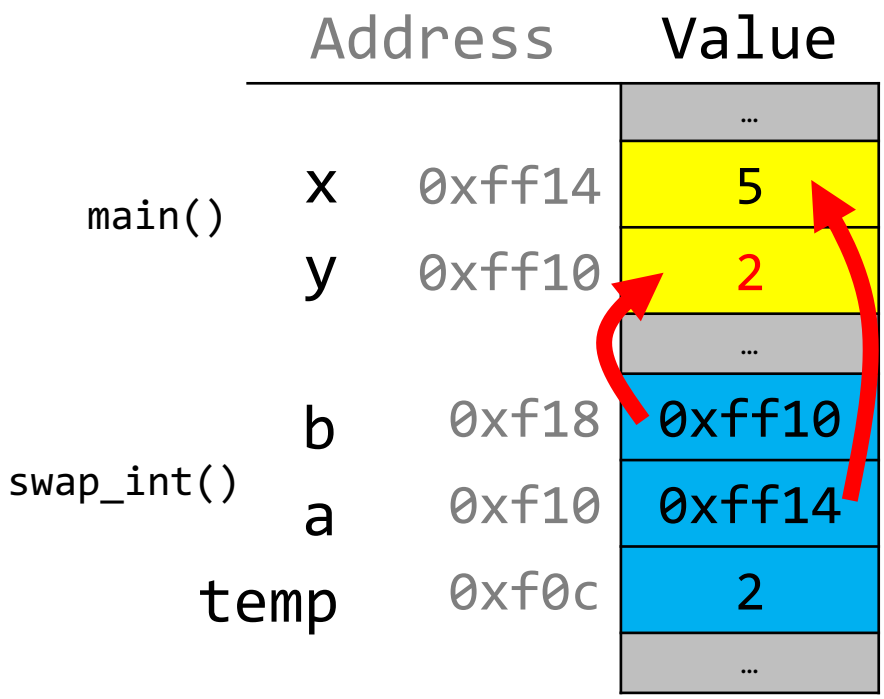
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```



Void *: Generic Swap

- You're asked to write a function that swaps two ints.

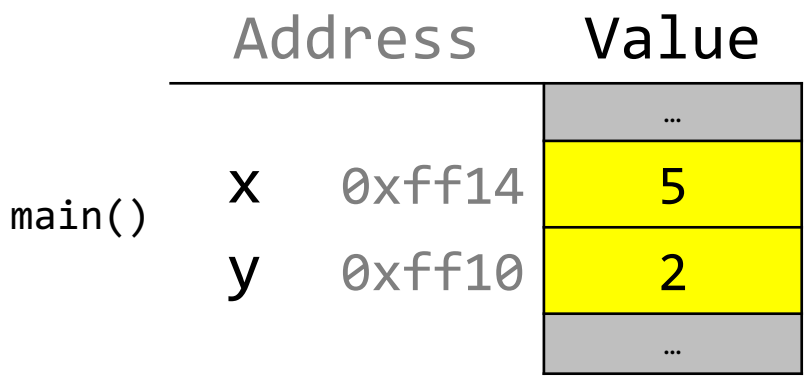
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```



Void *: Generic Swap

- You're asked to write a function that swaps two ints.

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```



Average PM: Nevermind, I meant shorts not ints…

Ok not that bad of a fix…

Void *: Generic Swap

- You're asked to write a function that swaps two shorts.

```
void swap_short(short *a, short *b) {
    short temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    short x = 2;
    short y = 5;
    swap_short(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Everything that was an int, we need to change to be a short!

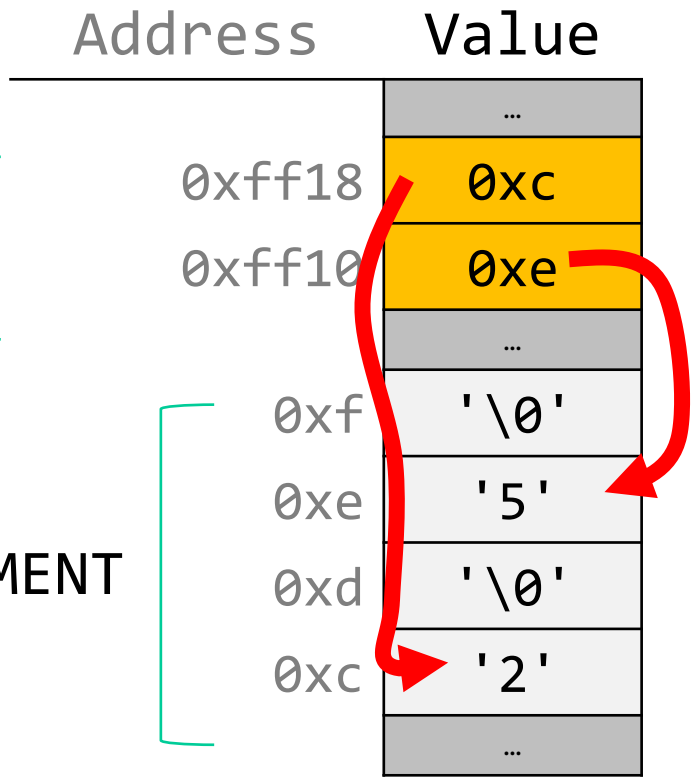
Average PM: You know what, I goofed. We're going to use strings. Could you write something to swap those?

Um sure……

Void *: Generic Swap

- You're asked to write a function that swaps two srtings.

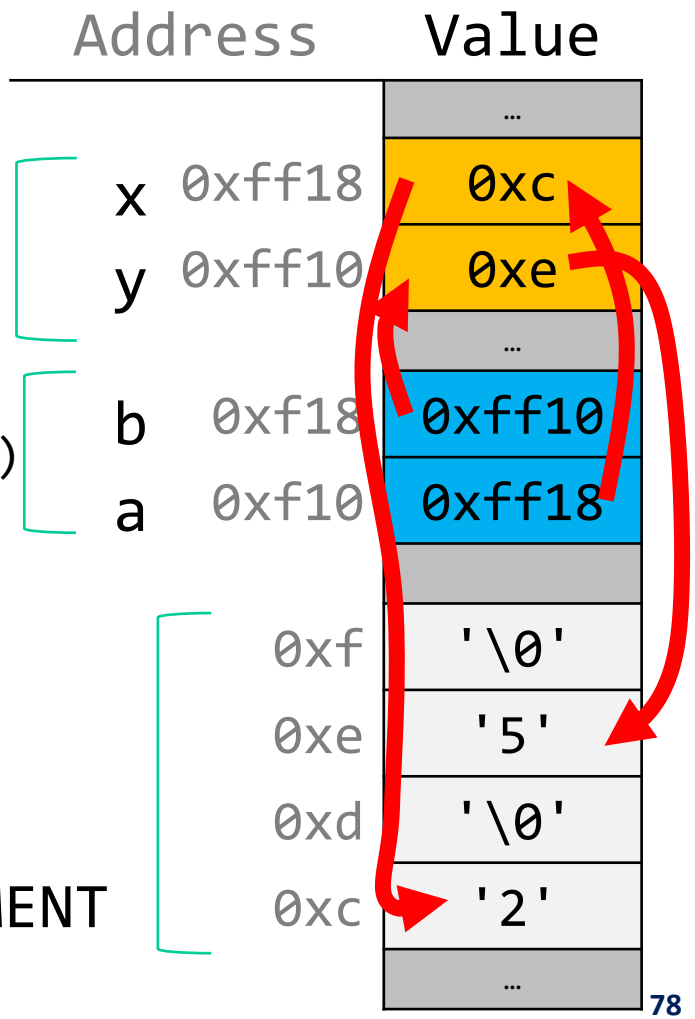
```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



Void *: Generic Swap

- You're asked to write a function that swaps two shorts.

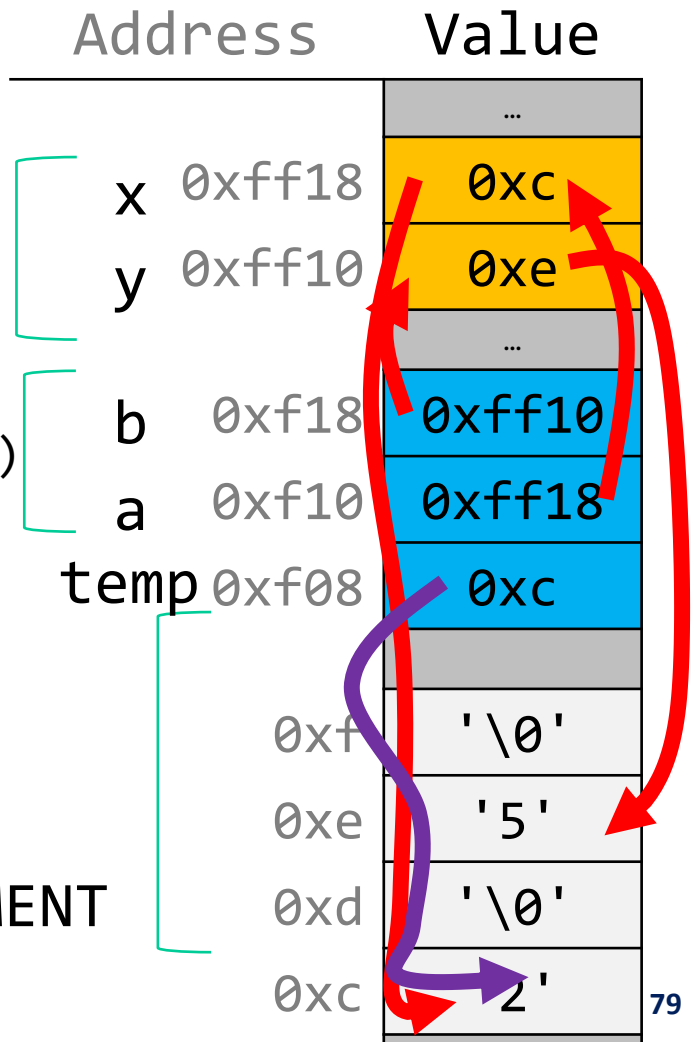
```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



Void *: Generic Swap

- You're asked to write a function that swaps two shorts.

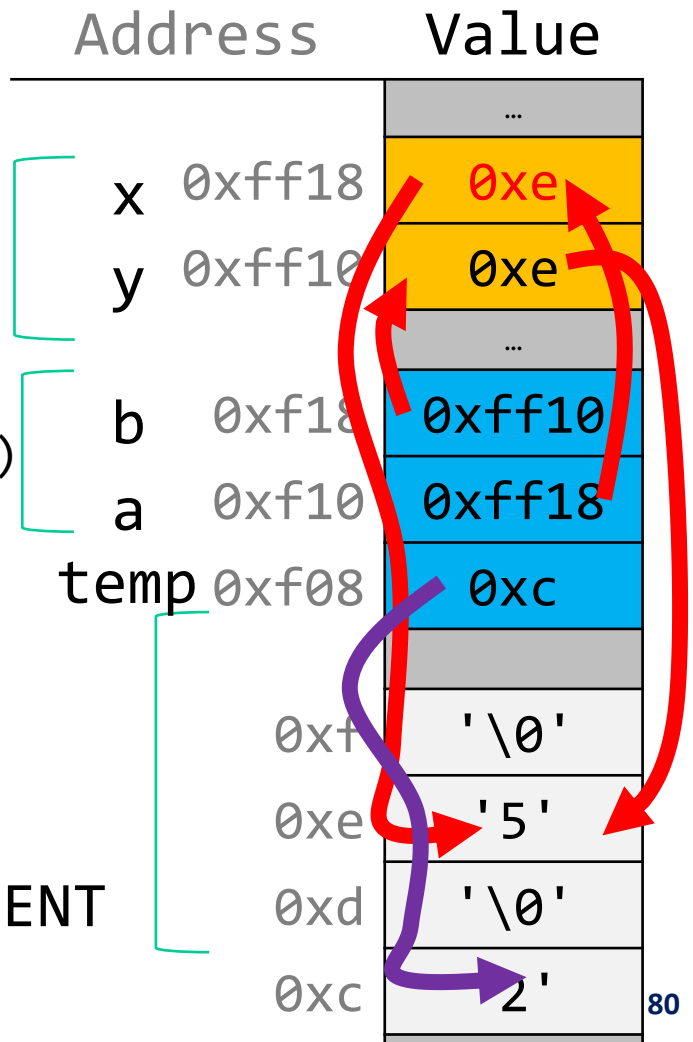
```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



Void *: Generic Swap

- You're asked to write a function that swaps two shorts.

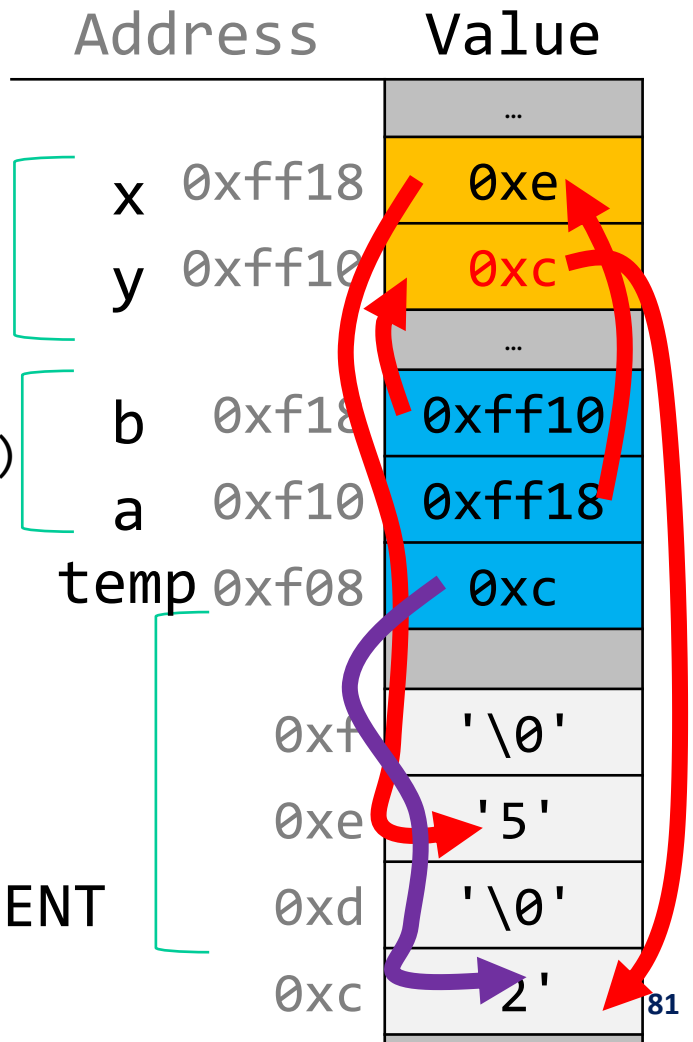
```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



Void *: Generic Swap

- You're asked to write a function that swaps two shorts.

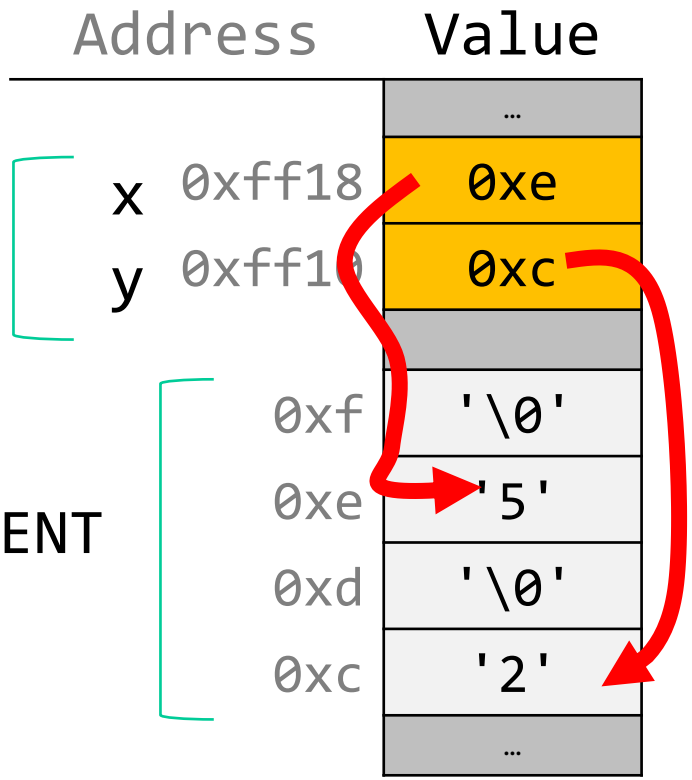
```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



Void *: Generic Swap

- You're asked to write a function that swaps two shorts.

```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



Average PM: You know what, we have 60 different custom structs. Can you write a custom swap for each one?

Umm I quit…

Imagine a World Where…

We could write *one* function to swap two values of any single type?

```
void swap_int(int *a, int *b) { … }
void swap_float(float *a, float *b) { … }
void swap_size_t(size_t *a, size_t *b) { … }
void swap_double(double *a, double *b) { … }
void swap_string(char **a, char **b) { … }
void swap_mystruct(mystruct *a, mystruct *b)
{ … }
…
```

Generic Swap

```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void swap_short(short *a, short *b) {
    short temp = *a;
    *a = *b;
    *b = temp;
}

void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}
```

All 3:

- Take pointers to values to swap
- Create temporary storage to store one of the values
- Move data at **b** into where **a** points
- Move data in temporary storage into where **b** points

General Swap Outline

```
void swap(pointer to data1, pointer to data2) {
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

We don't know the sizes of the pointers now! Would be wrong to use **char ***, **int ***, etc.

Introducing Void *

```
void swap(void *data1, void * data2) {
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

Void *: a generic pointer to some memory address.

Big Note: We have no idea what type we are pointing at.

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

If we don't know the data type, we don't know how many bytes it is. Let's take that as another parameter.

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    void temp; ???
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

Let's start by making space to store the temporary value.
How can we make **nbytes** of temp space?

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

Chars are only 1 byte! So let's make an nbyte char array…

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
```

Now, how can we copy in what **data1** points to into **temp**?

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    temp = *data1ptr; ???

    // copy data2 to location of data1
    // copy data in temporary storage to location of data2

}
```

We can't dereference a **void** * (or set an array equal to something). C doesn't know what it points to! Therefore, it doesn't know how many bytes it should grab.

memcpy

```
void *memcpy(void *dest, const void *src, size_t n);
```

- **memcpy** is a function that copies a specified number of bytes at one address to another address.
- It copies the next **n** bytes that **src** points to to the location pointed to by **dest**. (It also returns **dest**). It does not support regions of memory that overlap.

```
int x = 5;
int y = 4;
memcpy(&x, &y, sizeof(x)); // just like x = y;
```

memmove

```
void *memmove(void *dest, const void *src, size_t n);
```

- **memmove** is the same as memcpy but supports overlapping regions of memory. (Unlike its name implies, it still "copies").
- It copies the next **n** bytes that **src** points to to the location pointed to by **dest**. (It also returns **dest**).

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);

    // copy data2 to location of data1
    // copy data in temporary storage to location of data2

}
```

We can copy the bytes ourselves into temp! This is equivalent to **temp = *data1ptr** in non-generic versions, but this works for *any* type of *any* size.

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);

    // copy data2 to location of data1
    // copy data in temporary storage to location of data2

}
```

How can we copy data2 to the location of data1?

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);

    // copy data in temporary storage to location of data2

}
```

How can we copy data2 to the location of data1?
memcpy!

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}
```

Our last step, copy data in **temp** into **data2**

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}
```

```
int x = 2;
int y = 5;
swap(&x, &y, sizeof(x));
```

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}
```

```
short x = 2;
short y = 5;
swap(&x, &y, sizeof(x));
```

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}
```

```
char *x = "2";
char *y = "5";
swap(&x, &y, sizeof(x));
```

Introducing Void *

```
void swap(void *data1, void * data2, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}
```

```
mystruct x = {…};
mystruct y = {…};
swap(&x, &y, sizeof(x));
```

C Generics and Void *

- We can use **void *** and **memcpy** to handle memory as generic bytes.
- If we are given where the data of importance is, and how big it is, we can handle it!

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}
```

That's all folks!

- Goodluck on HW 02!
- See y'all on Thursday!

Have a great weekend!