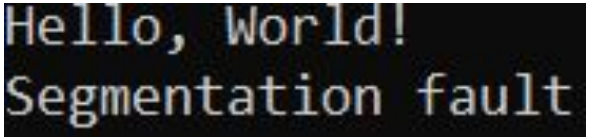
CSE 333 Section 2

Structs and Debugging



UNIVERSITY of WASHINGTON

Checking In & Logistics

Do you have any questions, comments, or concerns?

Exercises going ok?

Lectures making sense?

Due Monday (1/17): Exercise 3 @ 11 am

Due Thursday (1/20): Homework 1 @ 11:59 pm – START EARLY!

Structs and Typedef Review

Defining Structs

- To define a struct, we use the `struct` statement, which typically has a name (a tag) and must have one or more data members
  - This defines a new data type!

```
struct word_st {
  char* word;
  int   count;
};
struct word_st my_word;
```

Typedef

- The C Programming language provides the keyword `typedef`, which defines an alias (alternate name) for an existing data type
  - This can be used in combination with a `struct` statement

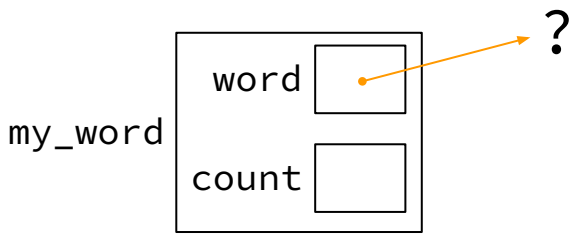
```
typedef struct word_st {
  char* word;
  int   count;
} WordCount;
WordCount my_word;
```

```
struct word_st {
  char* word;
  int   count;
};
typedef struct word_st WordCount;
WordCount my_word;
```

Structs and Memory Diagrams

- Struct instance is a box, with individual boxes for fields inside of it, labelled with field names
  - Even though we know that field ordering is guaranteed, we can be loose with where we place the fields in our diagram

```
typedef struct word_st {
   char* word;
   int   count;
} WordCount;
 WordCount my_word;
```

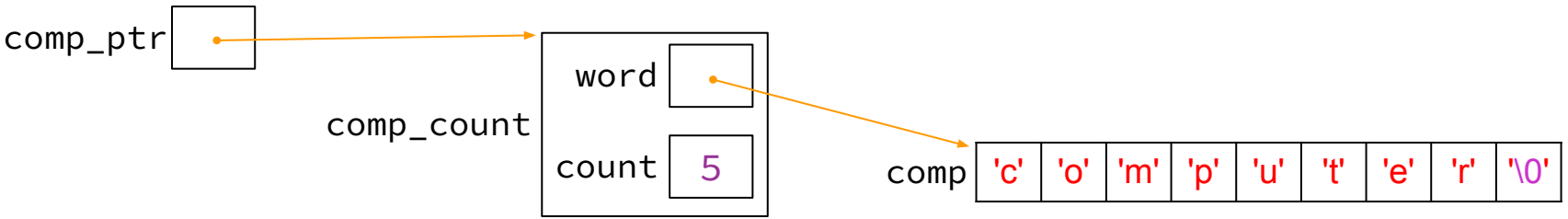


Structs and Pointers

- "`.`" to access field from struct instance
- "`->`" to access field from struct pointer

```c
typedef struct word_st {
  char* word;
  int   count;
} WordCount;
```

```c
char comp[] = "computer";
WordCount comp_count;
WordCount* comp_ptr = &comp_count;
comp_count.word = comp;
comp_ptr->count = 5;
```



Passing Structs

- Assignment copies over all of the field values
  - Unlike reference copying in Java

- Structs are *pass-by-copy* (as arguments and return values)
  - Can imitate pass-by-reference by passing pointer to struct instance instead

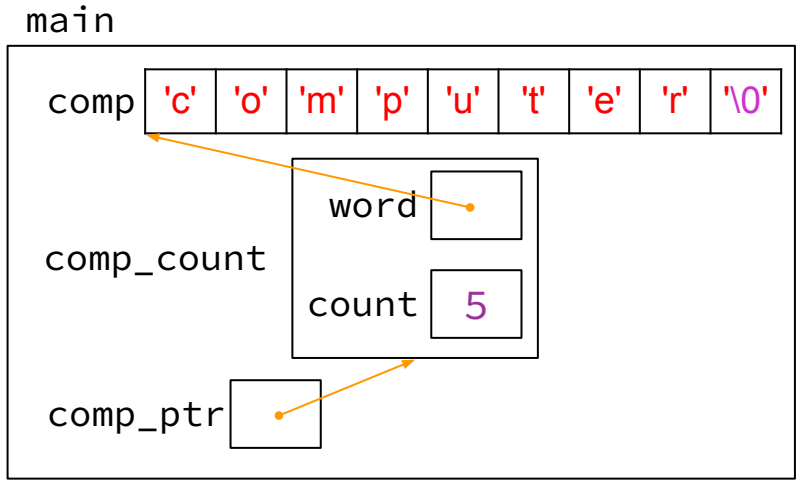
Exercise 1

Complete the Memory Diagram

Note: boxes with a function name above are local variables on the stack

```
int main(int argc, char* argv[]) {
  char comp[] = "computer";
  WordCount comp_count = {comp, 5};
  WordCount* comp_ptr = &comp_count;

  printf("1. %s, %d\n", comp_ptr->word,
         comp_ptr->count);
  ...
}
```

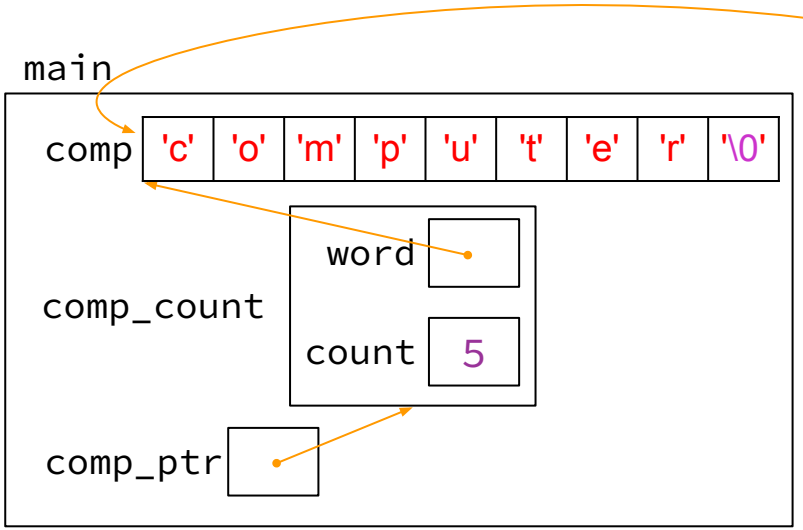


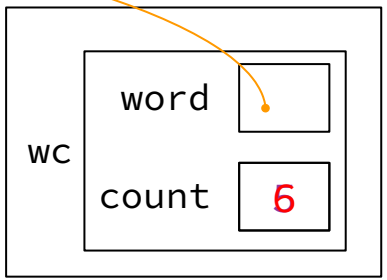
Console output

```
1. computer, 5
```

```
    // continued main code
    IncreaseCount(*comp_ptr);
    printf("2. %s, %d\n", comp_ptr->word,
           comp_ptr->count);
    ...
}
```

```
void IncreaseCount(WordCount wc) {
    wc.count += 1;
}
```



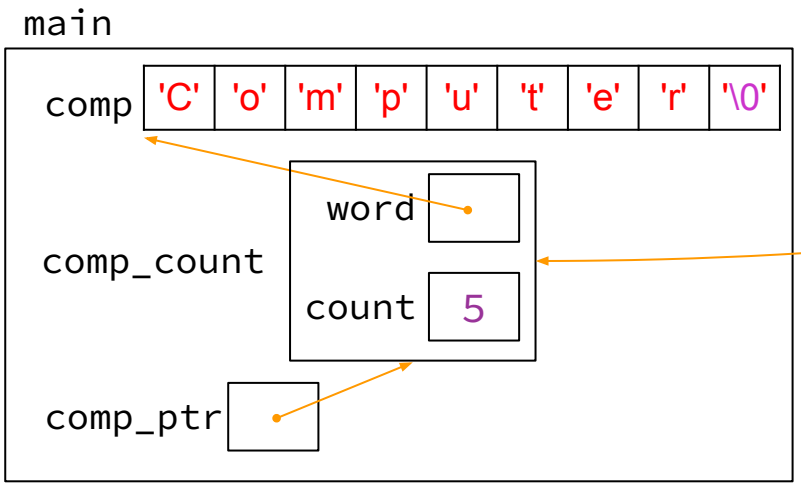


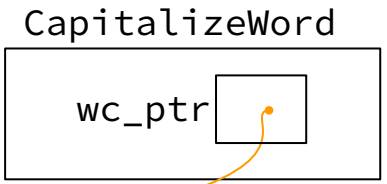
Console output

```
1. computer, 5
2. computer, 5
```

```
    // continued main code
    CapitalizeWord(comp_ptr);
    printf("3. %s, %d\n",
           comp_ptr->word,
           comp_ptr->count);
    ...
}
```

```
void CapitalizeWord(WordCount* wc_ptr) {
  wc_ptr->word[0] &= ~0x20;
}
```



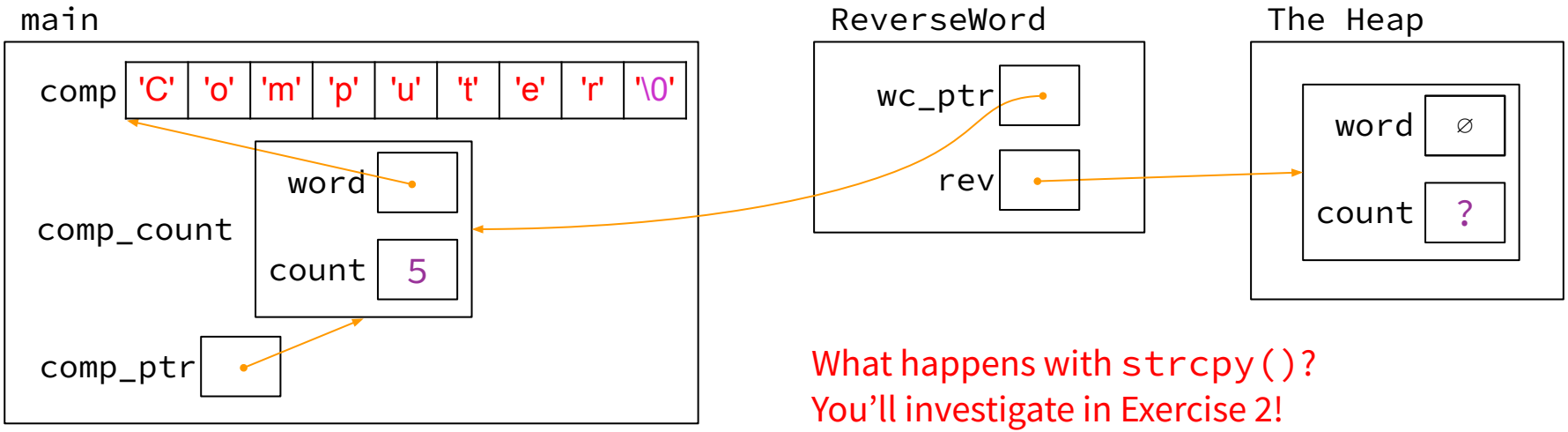


Console output

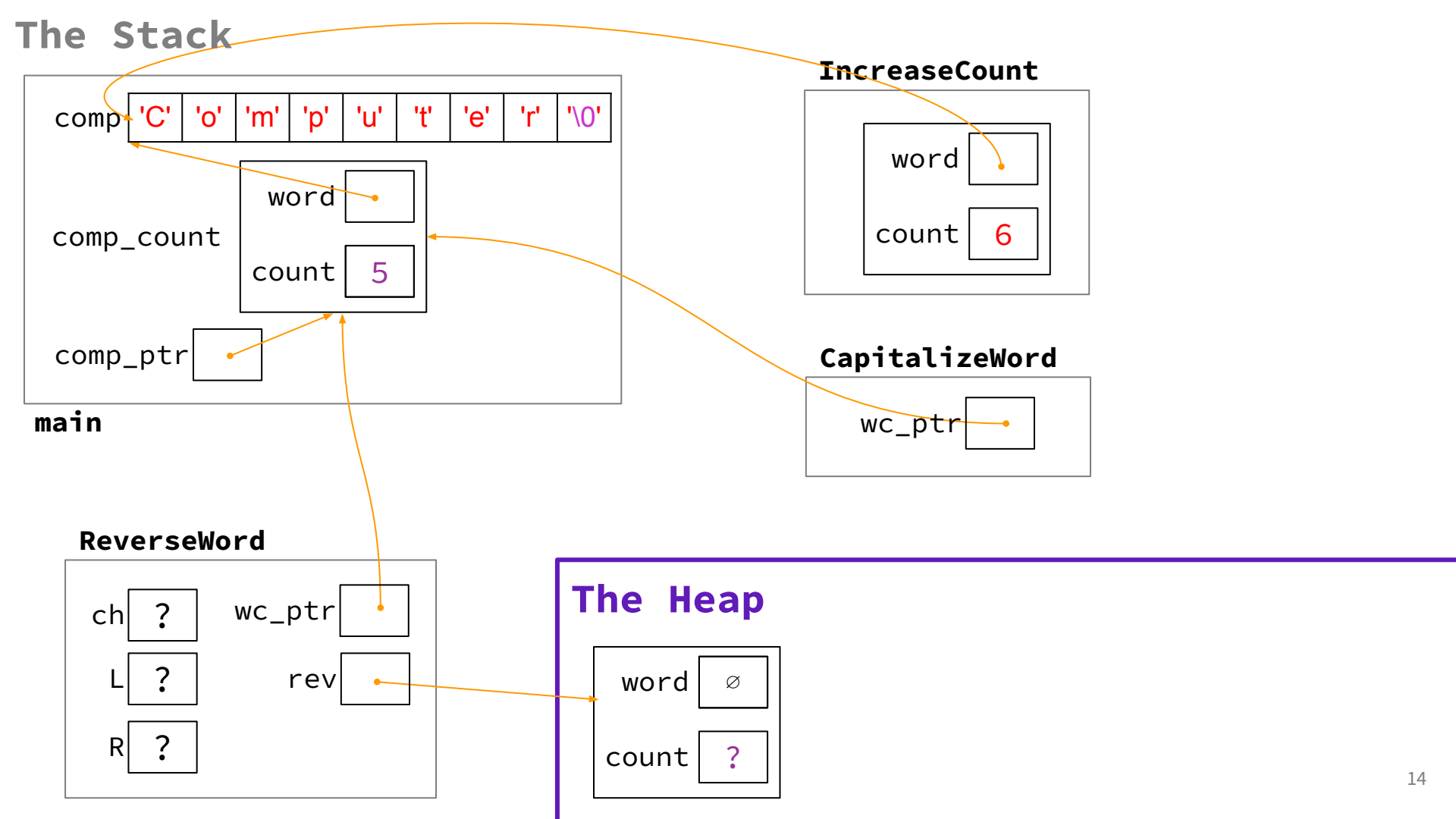
```
1. computer, 5
2. computer, 5
3. Computer, 5
```

```
    // continued main code
    *comp_ptr = ReverseWord(comp_ptr);
    printf("4. %s, %d\n",
           comp_ptr->word,
           comp_ptr->count);
    return EXIT_SUCCESS;
}
```

```
WordCount ReverseWord(WordCount* wc_ptr) {
    WordCount* rev = (WordCount*)
                     malloc(sizeof(WordCount));
    rev->word = NULL;
    strcpy(rev->word, wc_ptr->word);
    ...
```



What happens with `strcpy()`?
You'll investigate in Exercise 2!


The Stack
comp
'C'
'o'
'm'
'p'
'u'
't'
'e'
'r'
'\0'
word
comp_count
count
5
comp_ptr
main
IncreaseCount
word
count
6
CapitalizeWord
wc_ptr
ReverseWord
ch
?
L
?
R
?
wc_ptr
rev
The Heap
word
∅
count
?

Debugging Tools

Debugging

- ✨ **Debugging is a skill that you will need throughout your career!** ✨
- The 333 projects are big with lots of potential for bugs
  - Learning to use the debugging tools will make your life a lot easier
  - Course staff will help you learn the tools in office hours, too
- Debugging tool output can be scary at first, but extremely useful once you know how to parse it

333 Debugging Tools

- `gdb` (GNU Debugger) is a general-purpose debugging tool
  - Stops at breakpoints and program crashes
  - Lots of helpful features for tracing code, checking current expression values, and examining memory
- `valgrind` specifically check for memory errors
  - Great for catching non-crashing odd behavior (*e.g.*, using uninitialized values)
  - If your code uses `malloc`, should use `--leak-check=full` option

Tracing Code in gdb

- Setting breakpoints
  - `break <function_name>`
  - `break <filename:line#>`
- Advancing
  - `step` – into functions
  - `next` – over functions
  - `finish` – out of current function
  - `continue` – to next break
- Printing values
  - `print` – evaluate expression once
  - `display` – keep evaluating expression
- Examining memory
  - `x` – dereference provided address
- More on course website!

Common Errors: Misuse of Functions

- If you are unsure of what a function does or what its parameters or return value are for, *read the function block comment*
  - Typically on function declarations in header files
  - Write great function block comments yourself!!!
- If you are unsure of what a C library function does, use `man` to find more information
  - Example: `man strcpy`
  - Note: also supports Unix commands, but doesn't hold info for C++

Common Errors: Segmentation Fault

- Common causes of segmentation fault:
  - Dereferencing an uninitialized pointer, NULL, or a previously-freed pointer
  - Accessing past the end of an array
  - … many more!
- `gdb` is very helpful for identifying the source of a segfault
  - Will automatically stop execution when SIGSEGV signal received (`run`)
  - `backtrace` to examine current stack frames
  - `print` to identify the bad value that caused it

Common Errors: Memory “Errors”

- Common examples of memory issues:
  - Use of uninitialized memory
  - Reading/writing memory after it has been freed – Dangling pointers
  - 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
- `valgrind` will tell you the line of code that caused or originated the memory issue
  - Only analyzes that specific execution, so may need to re-run to get more code coverage

Exercise 2

Fix 1: Doesn't increment

- Tool help: stepping through code with gdb
- Old version:

```
void IncreaseCount(WordCount wc) {
  wc.count += 1;
}
```

- New version:

```
void IncreaseCount(WordCount* wc_ptr) {
  wc_ptr->count += 1;
}
```

Fix 2: Segfault

- Tool help: `run` in gdb to find segfault, `man` for `strcpy`
- Old version:

```
rev->word = NULL;
strcpy(rev->word, wc_ptr->word);
```

- New version:

```
rev->word = (char*) malloc((strlen(wc_ptr->word) + 1)
                           * sizeof(char) );
strcpy(rev->word, wc_ptr->word);
```

Fix 3: Doesn't reverse string

- Tool help: break on `ReverseWord`, step through code, `print /s rev->word` at end of function (prints as string)
- Old version:

```
char ch;
int L = 0, R = strlen(rev->word);
```

- New version:

```
char ch;
int L = 0, R = strlen(rev->word) - 1;
```

Fix 4: Reading uninitialized memory

- Tool help: run under valgrind, identify error line number
- Old version:

  Did not set count!
- New version:

```
rev->count = 0;
```

Fix 5: Memory leaks

- Tool help: run under valgrind, identify unfreed allocation line numbers
- Old version:

```
WordCount ReverseWord(WordCount* wc_ptr) { ...
return *rev; }
```

- New version:

```
WordCount* ReverseWord(WordCount* wc_ptr) { ...
return rev; }
At end of main:    free(comp_ptr->word);
                   free(comp_ptr);
```

Exercise 3

Style Fixes

- Tool help: None? Lecture slides! Google C++ Style Guide!
- `malloc` error checking:

```
if (rev == NULL) { return NULL; }
if (rev->word == NULL) { return NULL; }
```

- struct passing:

```
WordCount* ReverseWord(WordCount wc);
```