

# CSE 333 23sp

## Section 2

Debugging and Structs



# Checking In & Logistics

- Exercise 3:
  - Due **Monday @ 11:00am (4/10)**
- HW1 Partner Form
  - Due **Thursday @ 11:59pm (4/6)**
  - Form is pinned on Ed
- Homework 1:
  - Due **Thursday @ 11:59pm (4/13)**
  - Start Early!

Any questions, comments, or concerns?

- Exercises going ok?
- Lectures making sense?

# 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 simplestring_st {  
    char* word;  
    int   length;  
};  
struct simplestring_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

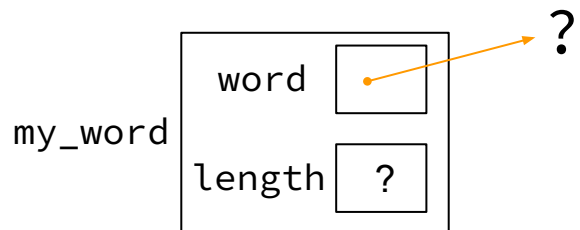
```
struct simplestring_st {  
    char* word;  
    int length;  
};  
typedef struct simplestring_st SimpleString;  
SimpleString my_word;
```

```
typedef struct simplestring_st {  
    char* word;  
    int length;  
} SimpleString;  
SimpleString 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 simplestring_st {  
    char* word;  
    int length;  
} SimpleString;  
SimpleString my_word;
```

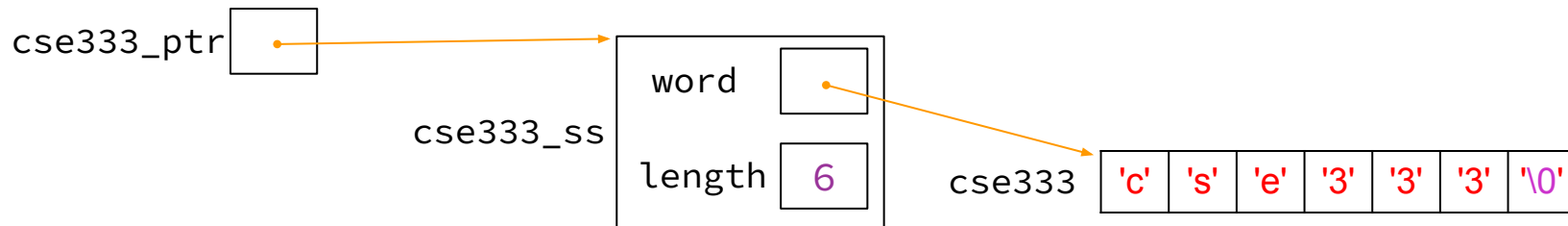


# Structs and Pointers

- “.” to access field from `struct` instance
- “->” to access field from `struct` pointer

```
typedef struct simplestring_st {  
    char* word;  
    int length;  
} SimpleString;
```

```
char cse333[] = "cse333";  
SimpleString cse333_ss;  
SimpleString* cse333_ptr = &cse333_ss;  
  
cse333_ss.word = cse333;  
cse333_ptr->length = strlen(cse333);
```



# Passing Structs as Parameters

- 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



# 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
- Why can't I just use print statements? They got me through 14x?
  - Bigger badder bugs beseech better debuggers!



# Debugging Strategies

Many debugging strategies exist but here's a simple 5 step process!

1. **Observation:** Something is wrong with your program!
2. **Hypothesis:** What do you think is going wrong?
3. **Experiment:** Use debuggers and other tools to verify the problem
4. **Analyze:** Identify and implement a fix to the problem.
5. Repeat steps 1-4 until *bug free*!

# Key debugging skills to master

1. Stop at “interesting” places
  - Debug after a crash or segfault
  - Use breakpoints to stop during execution
2. Look around when stopped
  - Print values of variables
  - Look at source code
  - Look up/down call chain
3. Resume execution
  - Incrementally, step at a time
  - Until next breakpoint
  - Until finished

# 333 Debugging Options

- `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, memory leaks on the heap)
  - If your code uses `malloc`, should use `--leak-check=full` option

# Basic Functions in GDB

- Setting breakpoints:
  - `break <filename>:<line#>`
- Advancing
  - `step` – into functions
  - `next` – over functions
  - `continue` – to next break
- Reading Values
  - `print` – evaluate expression once
  - `display` – keep evaluating expression
- Examining memory
  - `x` – dereference provided address
  - `bt` – backtracing

- Reference Card:

[https://courses.cs.washington.edu/courses/cse333/23sp/debug/gdb\\_refcard.pdf](https://courses.cs.washington.edu/courses/cse333/23sp/debug/gdb_refcard.pdf)

```
Hello World!  
Segmentation fault (core dumped)
```

# Common Errors

- **Misusing Functions:** Read documentation (online, through man pages, or the .h files for your homework) for function parameters and function purpose
  - Oftentimes, this leads to unexpected results!
- **Segmentation Fault:** Dereferencing an uninitialized pointer, NULL, a previously-freed pointer, or many other things.
  - GDB automatically halts execution when SIGSEGV is received, useful for debugging
- **Memory “Errors”:** Many possible errors, commonly use of uninitialized memory or “memory leaks” (data allocated on heap that does not get free’d).
  - Use `valgrind` to help catch memory errors!

# Trying to Run `wordcount.c`

We have a program `wordcount.c` that accepts a string from the user and reverses it!

- `wget` <https://courses.cs.washington.edu/courses/cse333/23sp/sections/02/code/wordcount.c>
- `wget` <https://courses.cs.washington.edu/courses/cse333/23sp/sections/02/code/Makefile>
- run `make` to compile the code
- run `make val` to run valgrind

But it has a few problems... let's take a look!



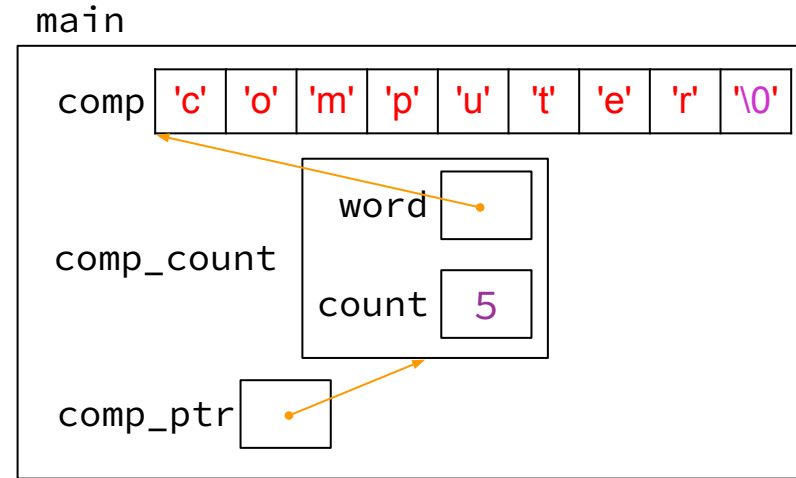
# Exercise 1

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

# Complete the Memory Diagram

```
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);
    ...
}
```



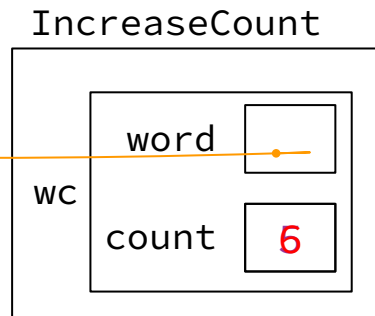
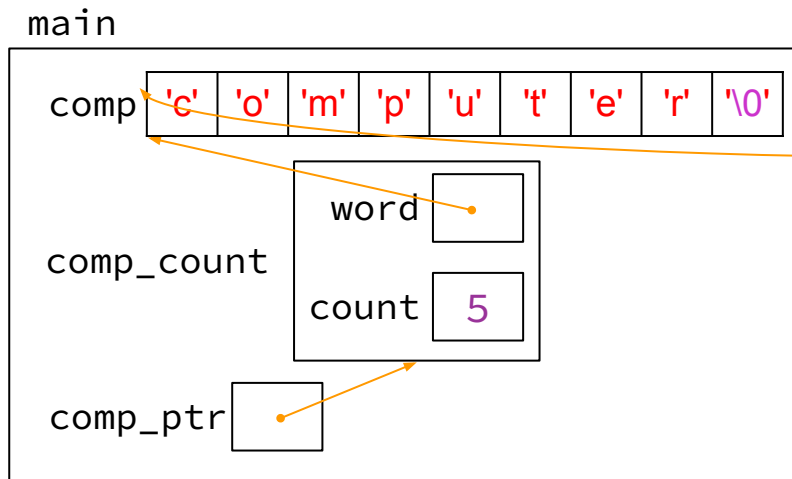
Console output

```
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);
```

```
...
```

```
}
```

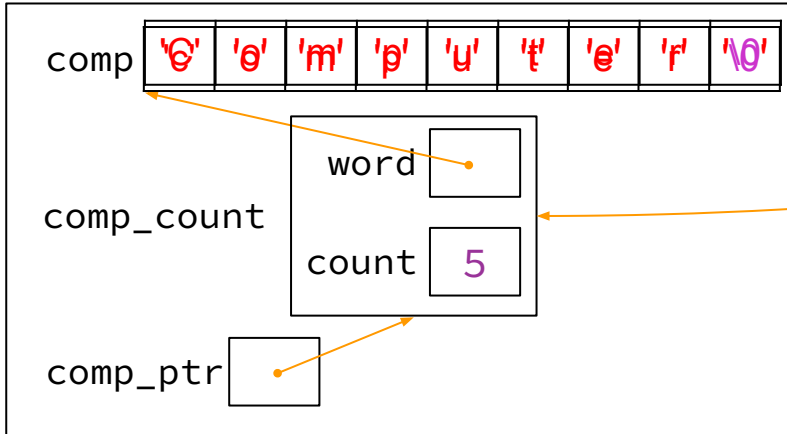
By changing the second bit in the upper nibble (6th most significant bit) of the character you can transform a character into uppercase or lowercase. If it's a 1 then it's lowercase, 0 is uppercase.

Going to lowercase: `lowch = ch | 0x20;`

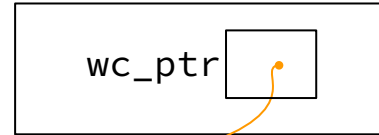
Going to uppercase: `upch = ch & ~0x20;`

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

main



CapitalizeWord



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) {
```

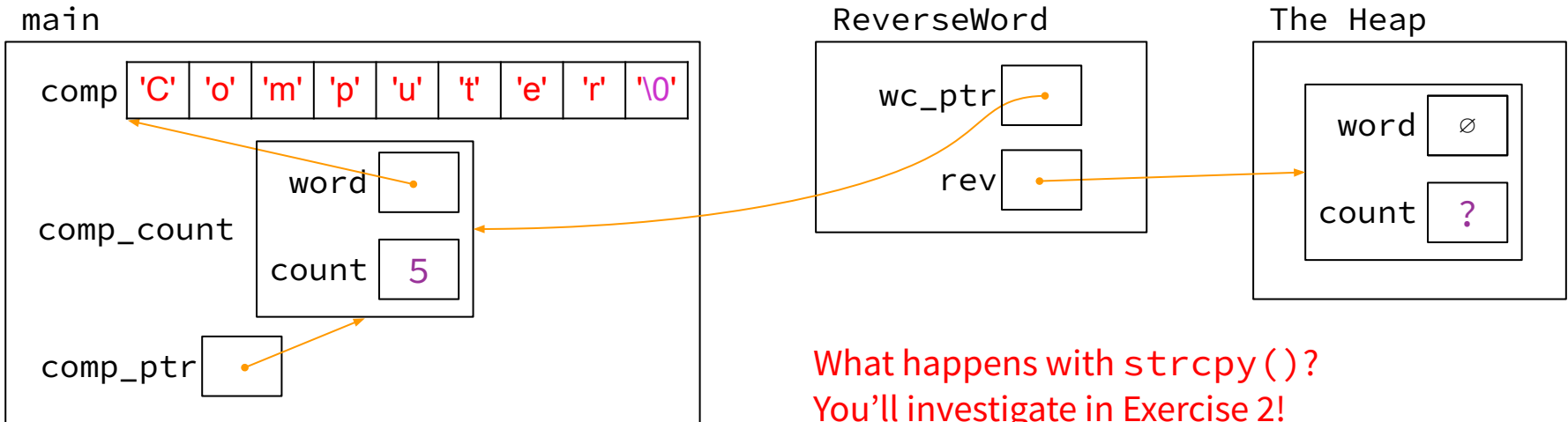
```
→ WordCount* rev = (WordCount*)
```

```
    malloc(sizeof(WordCount));
```

```
→ rev->word = NULL;
```

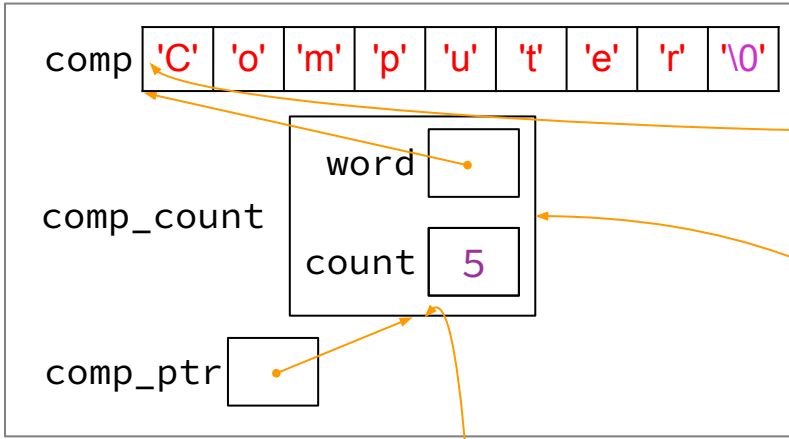
```
→ strcpy(rev->word, wc_ptr->word);
```

```
...
```



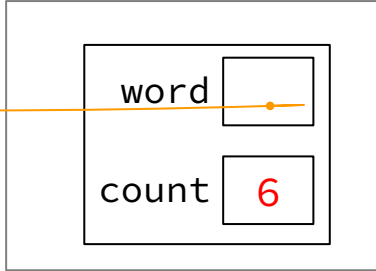
What happens with `strcpy()`?  
You'll investigate in Exercise 2!

# The Stack

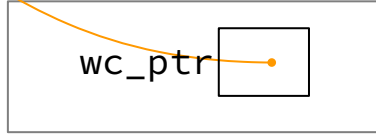


main

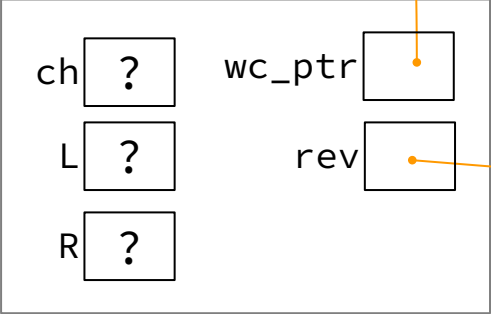
## IncreaseCount



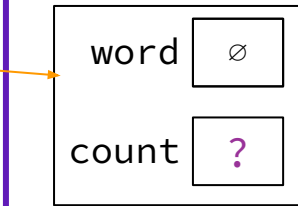
## CapitalizeWord



## ReverseWord



## The Heap



# 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);
```

# Some Debugging Tips

- Count and track where you call `malloc()` to make it easier for you to call `free` later
- Run `valgrind` frequently, for example after implementing a particular method
- Running `gdb + backtrace` after a `segfault` is a good place to start debugging
-

# Exercise 4

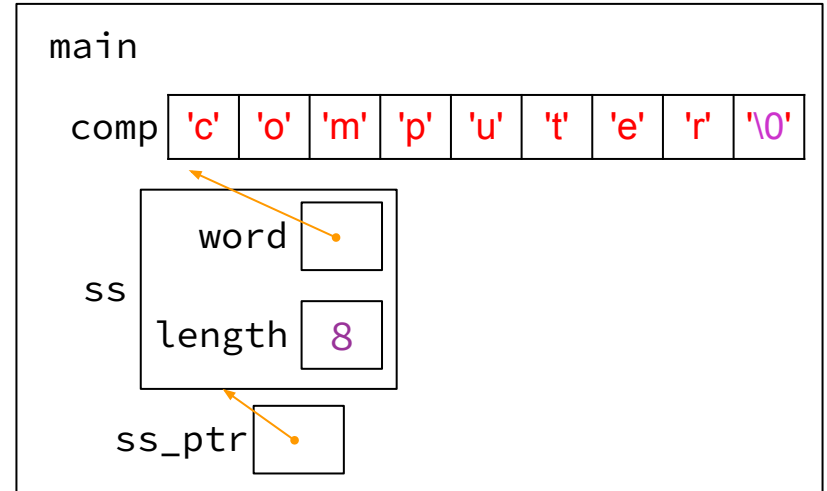


Note: boxes with a function name above are local variables on the stack

# Complete the Memory Diagram

```
int main(int argc, char* argv[]) {
    char comp[] = "computer";
    SimpleString ss = {comp, strlen(comp)};
    SimpleString* ss_ptr = &ss;

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



Console output

```
1. computer, 8
```

```

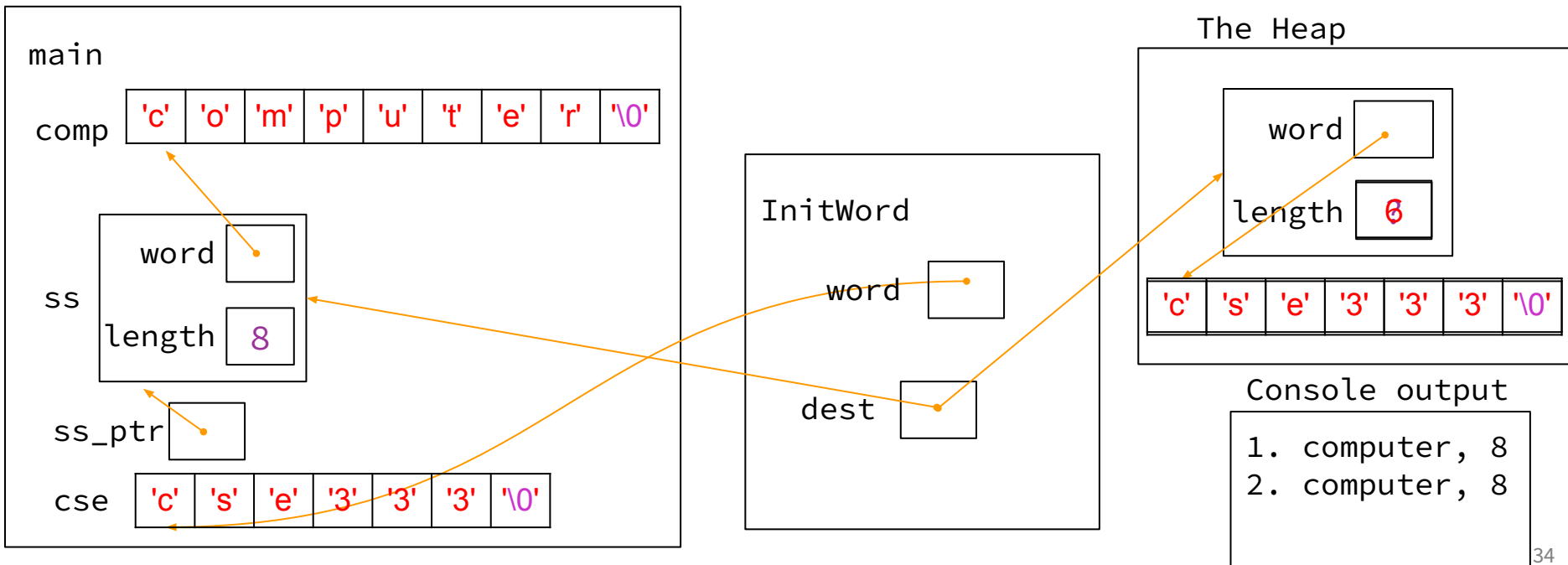
// continued main code
char cse[] = "cse333";
InitWord(cse, ss_ptr);
printf("2. %s, %d\n", ss_ptr->word,
      ss_ptr->length);
...
}

```

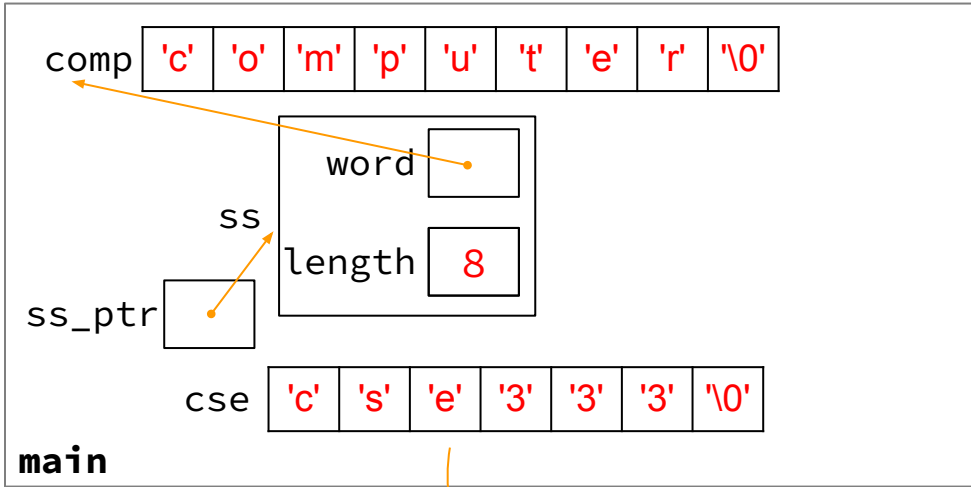
```

void InitWord(char* word, SimpleString* dest) {
    dest = (SimpleString*)
        malloc(sizeof(SimpleString));
    dest->length = strlen(word);
    dest->word = (char*) malloc(sizeof(char) *
        (dest->length + 1));
    strncpy(dest->word, word, dest->length + 1);
}

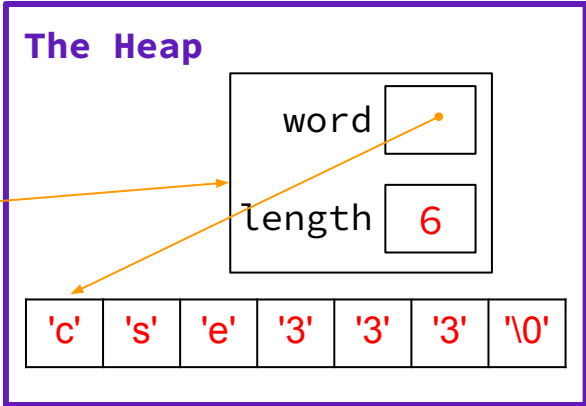
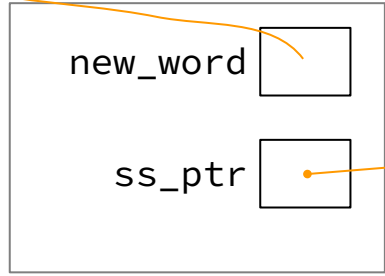
```



# The Stack



## InitWord



# Exercise 5 (Bonus)

# Exercise 5

- `InitWord` doesn't initialize a `SimpleString` properly... how can we fix that?
- If we can't edit the original pointer... modify a pointer to the pointer in main!

```
void InitWord(char* word, SimpleString** dest) {  
    *dest = (SimpleString*) malloc(sizeof(SimpleString));  
  
    (*dest)->length = strlen(word);  
    (*dest)->word = (char*) malloc(sizeof(char) * ((*dest)->length + 1));  
  
    strncpy((*dest)->word, word, (*dest)->length + 1);  
}
```