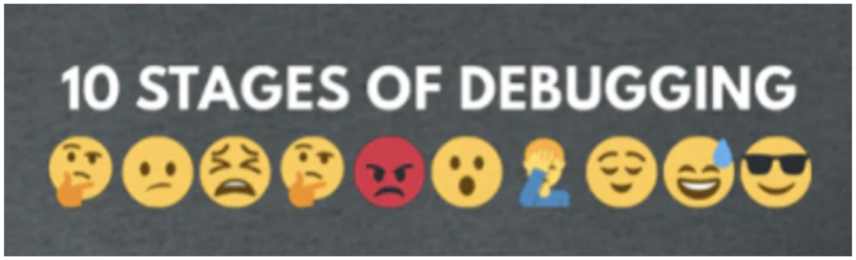
# CSE 333 22au
# Section 2



Debugging and Structs

UNIVERSITY *of* WASHINGTON

# Checking In & Logistics

- Exercise 3:
  - Due **Friday @ 10:00am (10/07)**
- Homework 1:
  - Due **Thursday @ 11:00pm (10/13)**
  - Start Early!

Any questions, comments, or concerns?

- Exercises going ok?
- Lectures making sense?

# 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

# 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

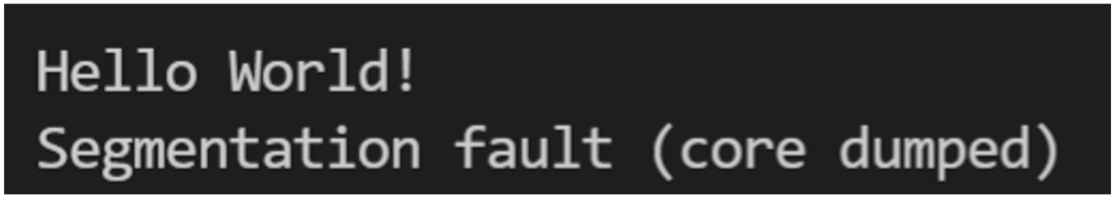
# Basic Functions in GDB

- 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/cse333/22au/resources/gdb-refcard.pdf

# Common Errors



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

- **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 `reverse.c`

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

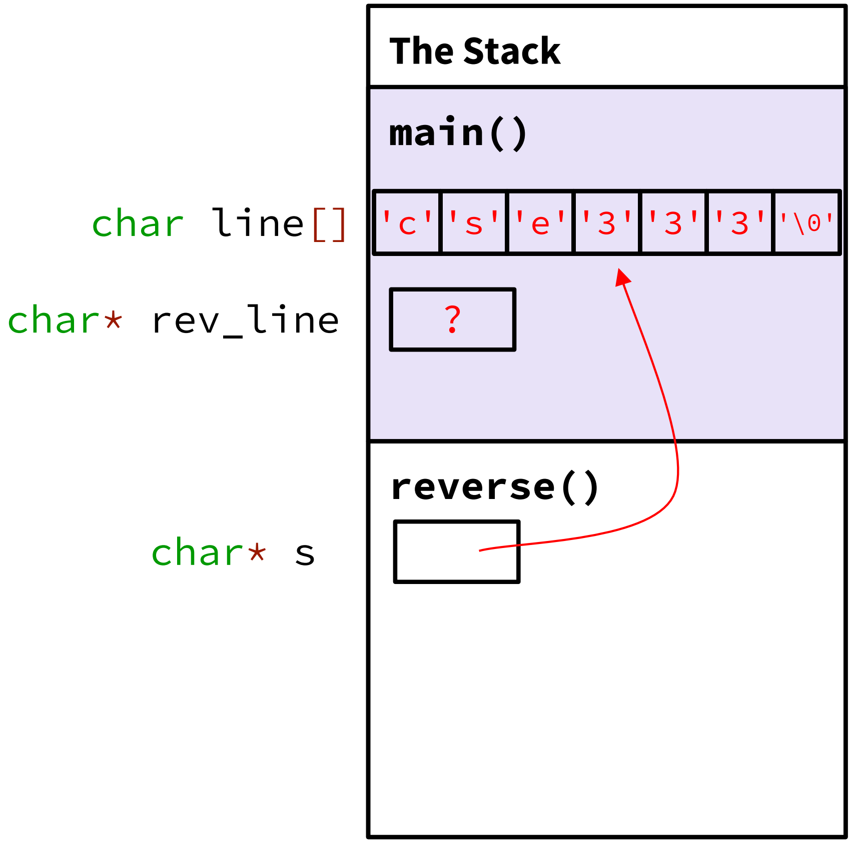
But it has a few problems… let’s take a look!

# Exercise 1

# Complete the Memory Diagram

```
  int main() {
➡ char line[MAX_STR];
➡ char* rev_line;

➡ printf("Please enter a string: ");
➡ fgets(line, MAX_STR, stdin);
➡ rev_line = reverse(line);
   .
   .
   .
```



The Stack

main()

char line[]

'c' 's' 'e' '3' '3' '3' '\0'

char* rev_line

?

reverse()

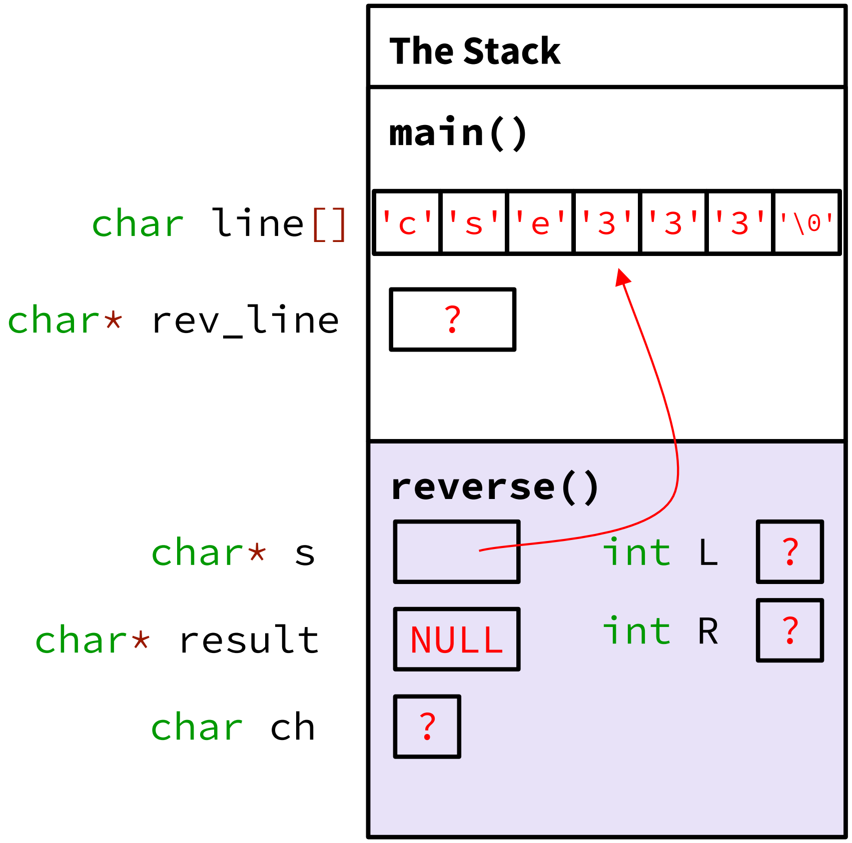
char* s

*unreached code omitted for space

# Complete the Memory Diagram

```
char* reverse(char* s) {
    char* result = NULL;
    int L, R;
    char ch;

    strcpy(result, s);
        .
        .
        .
```



The Stack

main()

char line[]

'c' 's' 'e' '3' '3' '3' '\0'

char* rev_line

?

reverse()

char* s

char* result

NULL

char ch

?

int L

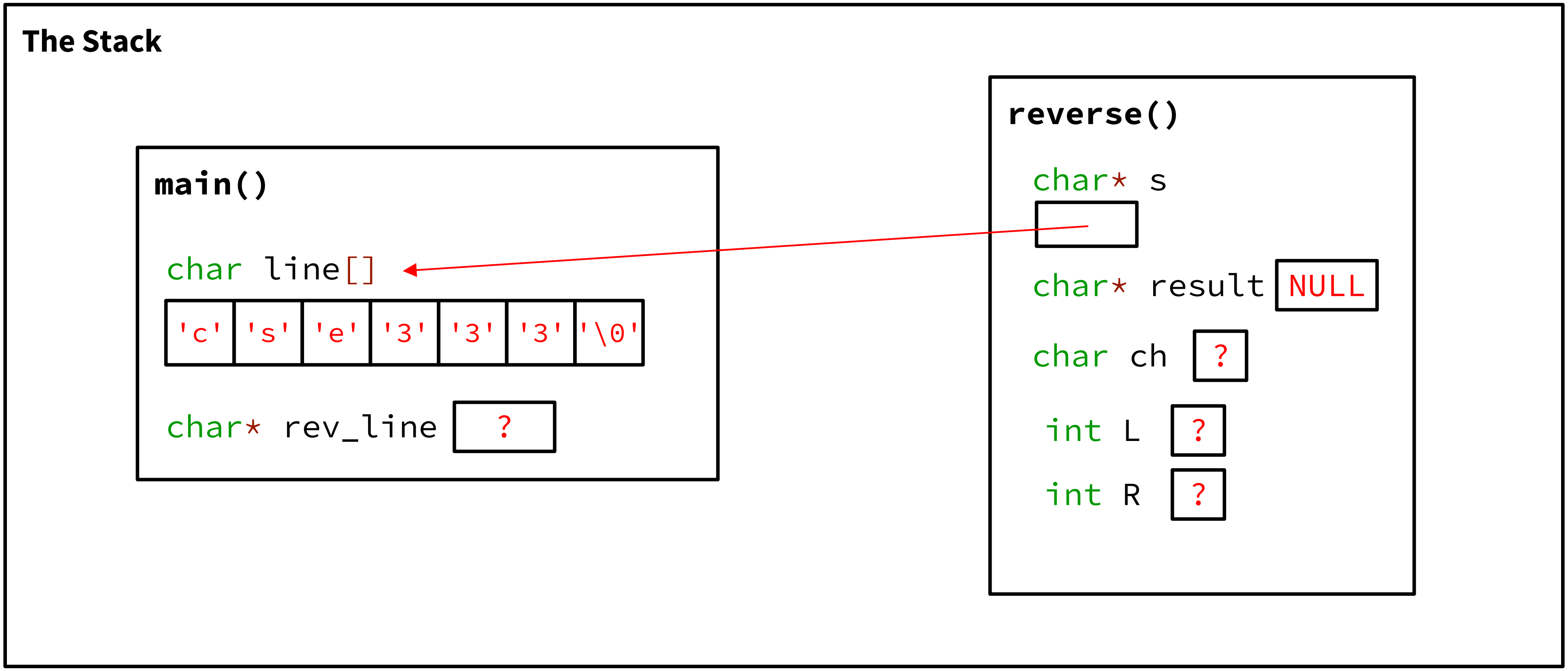
?

int R

?

*unreached code omitted for space

# Completed Memory Diagram



**The Stack**

**main()**

char line[]

| | | | | | | |
|---|---|---|---|---|---|---|
| 'c' | 's' | 'e' | '3' | '3' | '3' | '\0' |

char* rev_line ?

**reverse()**

char* s

char* result NULL

char ch ?

int L ?

int R ?

# Exercise 2 & 3

# Fix 1: Segfault

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

  ```
  result = NULL;
  strncpy(result, word, strsize);
  ```

- New version:

  ```
  result = (char*) malloc(strsize);
  strncpy(result, word, strsize);
  ```

# Fix 2: Doesn’t reverse string

- Tool help: run in gdb, break on `reverse()`, step through code, `print /s word` at end of function (prints as string)

- Old version:

```
char ch;
int L = 0, R = strlen(result);
```

- New version:

```
char ch;
int L = 0, R = strlen(result) - 1;
```

# Fix 3: Memory leaks

- Tool help: run under `valgrind`, identify un-freed allocation line numbers

- Old version:
  ```
  char* reverse(char* s) { ...
  return result; }
  ```
- New version:
  ```
  char* reverse(char* s) { ...
  return result; }
  At end of main:   free(rev_line);
  ```

# Style Fixes

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

```
result = (char*) malloc(strsize);
if (result == NULL) {
  // sample error checking. Read the spec on the requirements
  // for handling malloc!
  exit(EXIT_FAILURE);
}
```

- Remember to do this for the sake of code style! Malloc errors are rare, but we still check for failure to keep our code consistent

# 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

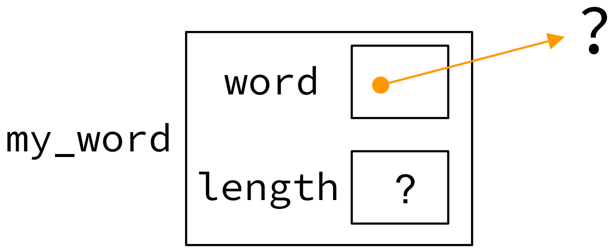
```c
struct simplestring_st {
  char* word;
  int   length;
};
typedef struct simplestring_st SimpleString;
SimpleString my_word;
```

```c
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;
```



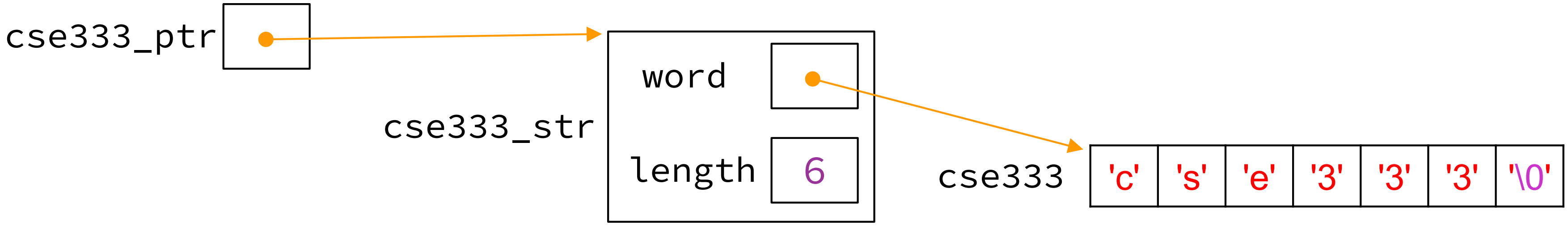
# Structs and Pointers

- "`.`" 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_count.word = cse333_ss;
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

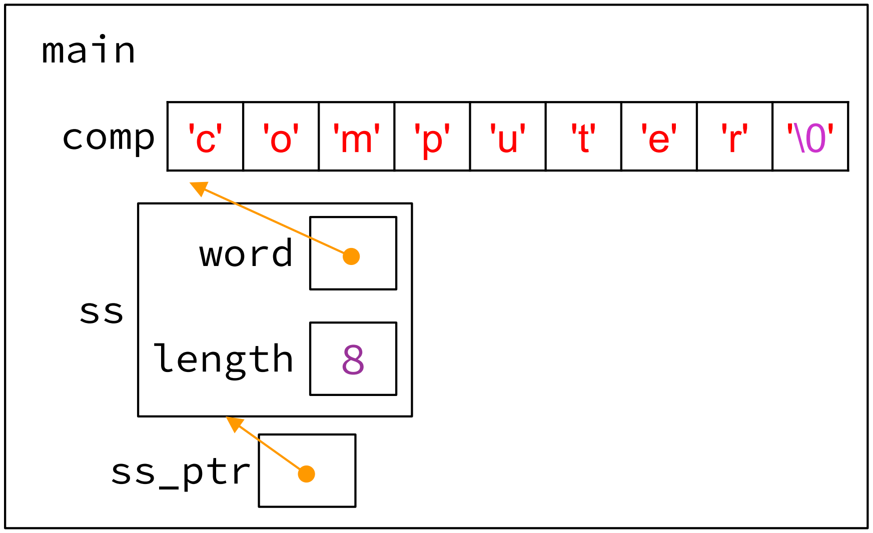
# Exercise 4

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);
  ...
}
```

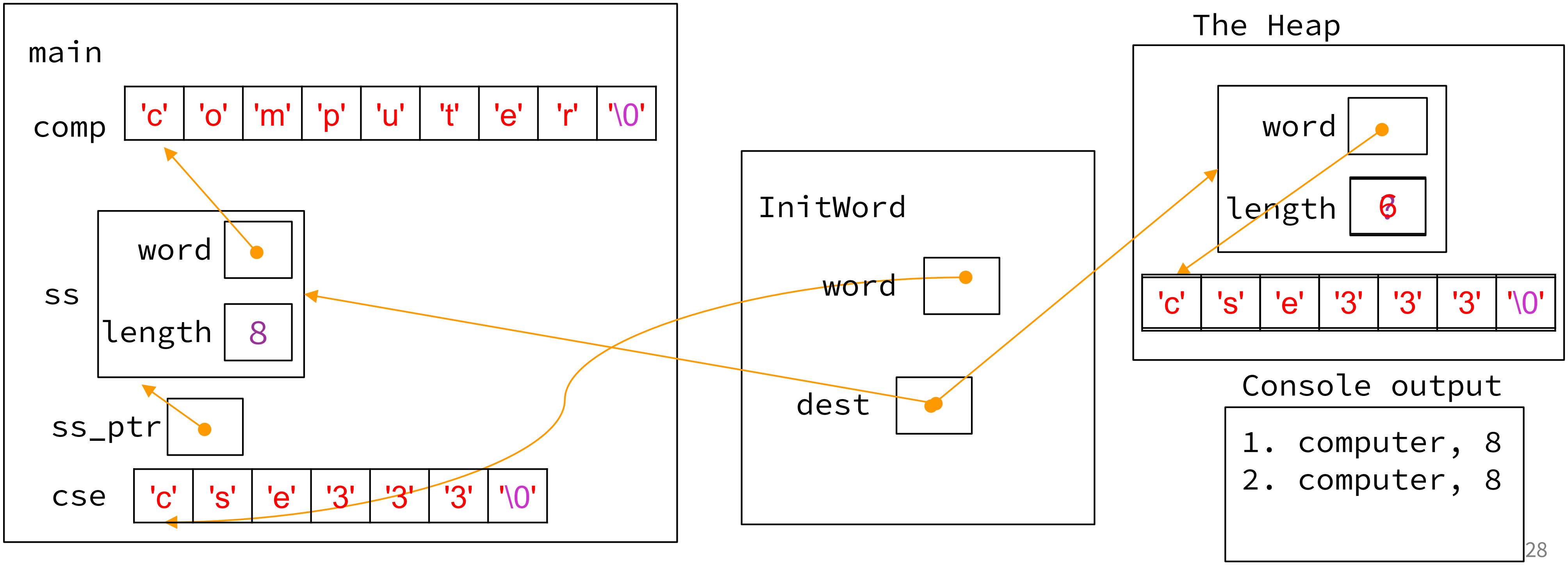


Console output

```
1. computer, 8
```

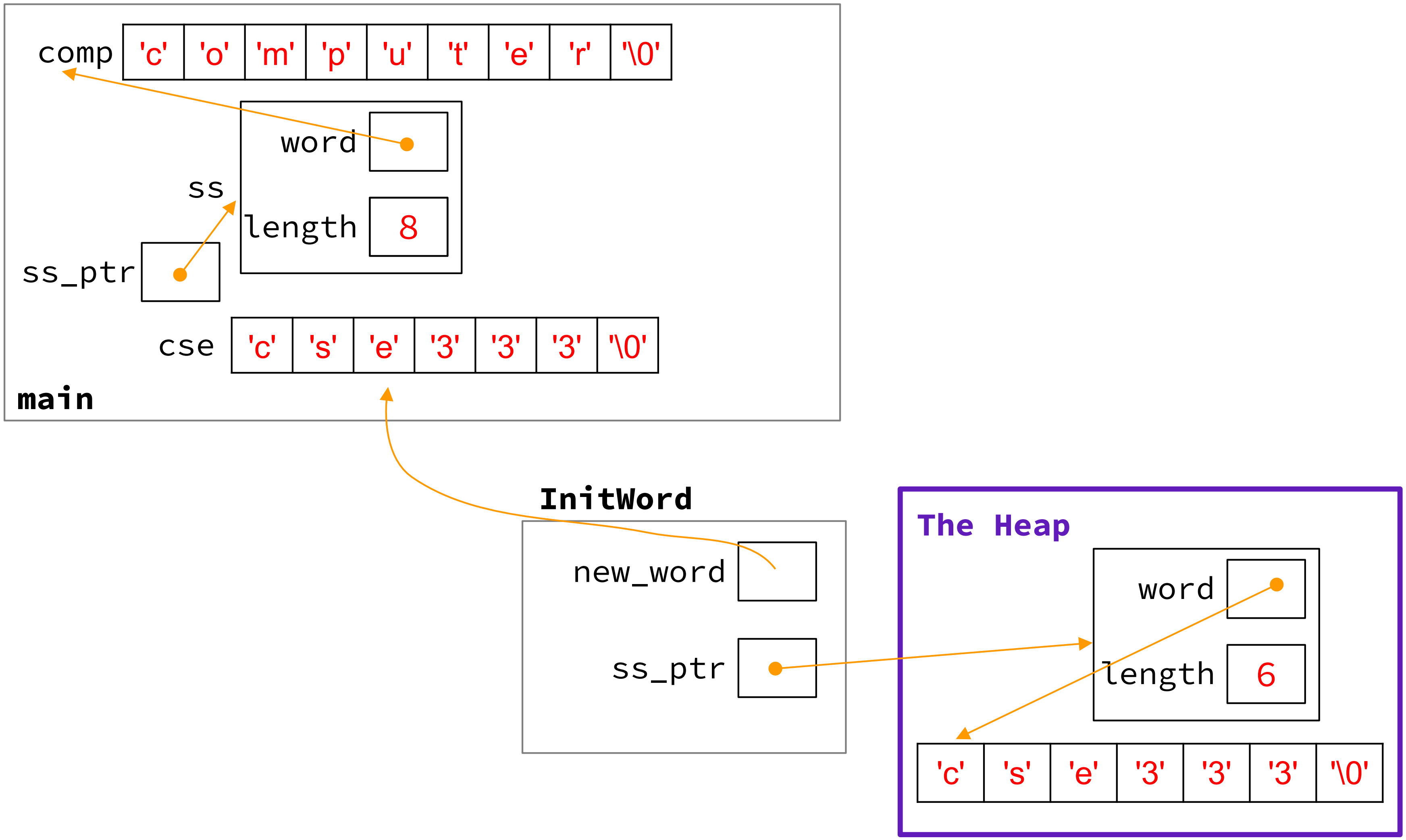
```c
  // continued main code
  char cse[] = "cse333";
  InitWord(cse, ss_ptr);
  printf("2. %s, %d\n", ss_ptr->word,
         ss_ptr->length);
  ...
}
```

```c
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);
 }
```



```
1. computer, 8
2. computer, 8
```

## The Stack



comp

| 'c' | 'o' | 'm' | 'p' | 'u' | 't' | 'e' | 'r' | '\0' |
|---|---|---|---|---|---|---|---|---|

ss

word

length 8

ss_ptr

cse

| 'c' | 's' | 'e' | '3' | '3' | '3' | '\0' |
|---|---|---|---|---|---|---|

**main**

**InitWord**

new_word

ss_ptr

**The Heap**

word

length 6

| 'c' | 's' | 'e' | '3' | '3' | '3' | '\0' |
|---|---|---|---|---|---|---|

# 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);
}
```