

CSE 333 24sp

Section 2

Debugging and Structs



Checking In & Logistics

- Exercise 5:
 - Due **Monday @ 10:00am (7/07)**
- Homework 1:
 - Due **Thursday next week @ 11:00pm (7/10)**
 - Start Early!
- No lecture tomorrow: Happy 4th 🎆

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;
```

The tag: simplestring_st

contents of the string

length of the string

Typedef

- The C Programming language provides the keyword `typedef`, which defines an alias (alternate name) for an existing data type

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

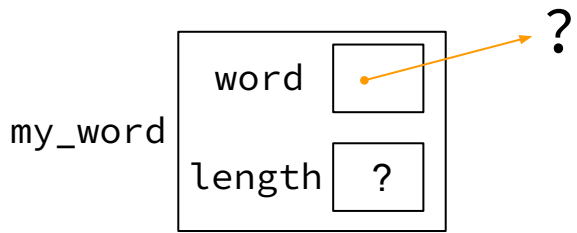
```
typedef struct simplestring_st {  
    char* word;  
    int   length;  
} SimpleString;  
SimpleString my_word;
```

shorter! more compact!

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;
```

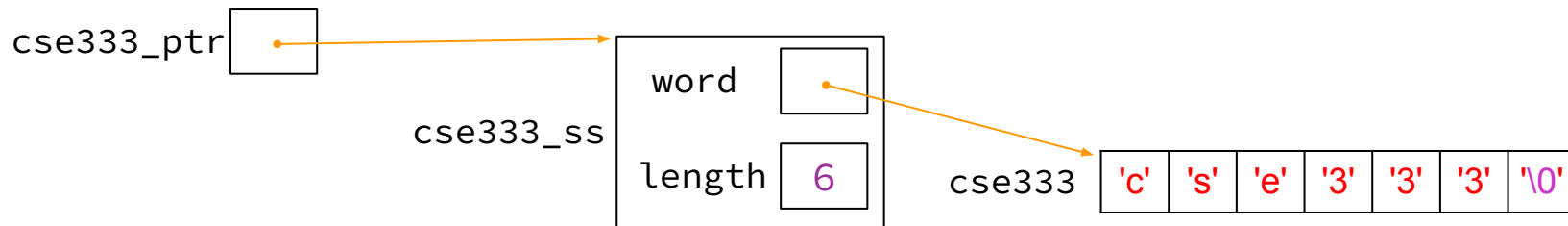


Structs and Pointers

- “.” 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

Debugging Strategy

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

1. **Observe symptoms:** Something is wrong with your program!
2. **Local Cause Hypothesis:** What do you think is going wrong?
3. **Test:** 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` to 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/cse333/25su/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

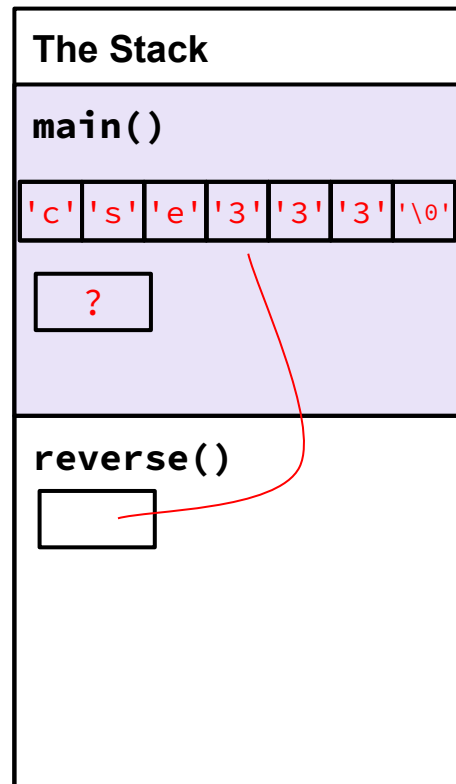
Complete the Memory Diagram

```
int main() {  
→ char line[MAX_STR];  
→ char* rev_line;  
  
→ printf("Please enter a string: ");  
→ fgets(line, MAX_STR, stdin);  
→ rev_line = reverse(line);  
.  
.  
.
```

char line[]

char* rev_line

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);  
  .  
  .  
  .  
}
```

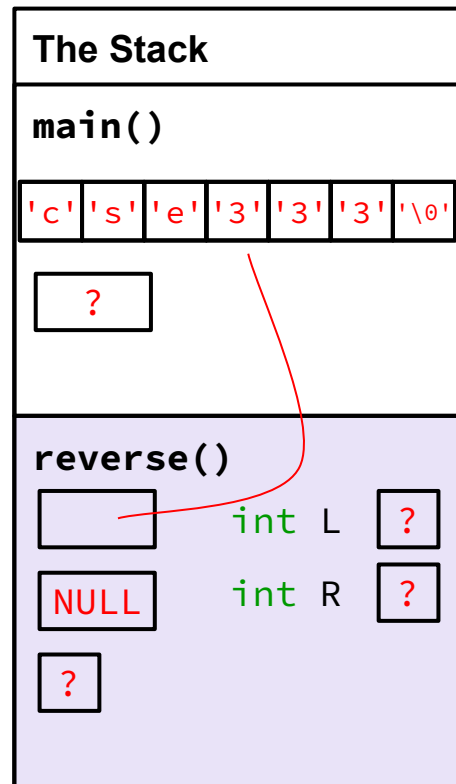
char line[]

char* rev_line

char* s

char* result

char ch



*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 occurrence

- Old version:

```
result = NULL;  
strcpy(result, s);
```

- New version:

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