



# CS107 Lab 2 – Welcome Back!



## C Strings, String Tokenization

Ben Yan, Winter 2026 🧊

Slides Link: [stanford.edu/~bbyan/cs107](https://stanford.edu/~bbyan/cs107) :)



Hope you've been having fun in Jerryland so far!

# Welcome Back to Lab!

*Hope that Assignment1 went / is going well,  
and that the bits & bytes are clicking together!*



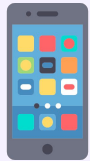
Winter



Week 3

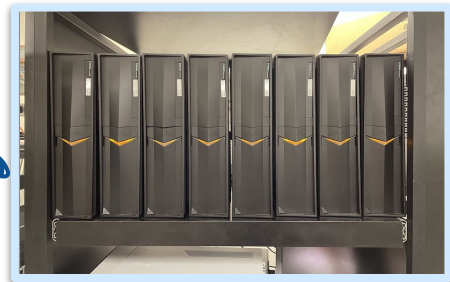


Spring



Also, if you haven't already, it's not a bad time to exchange contact info and find study partners or groups! [#powerOfFriendship](#)

# Getting Setup for Lab 2



- ❑ To get started, log into the **myth machines**, and clone the lab starter code using the command:

```
git clone /afs/ir/class/cs107/repos/lab2/shared lab2
```

- ❑ Next, also open the **online lab check-off** linked below (or in the lab 1 handout), so you can write as you go.



[Lab Handout](#)



[Lab Check-Off Form](#)

Don't worry at all if you fall behind on the write up! I'll aim to go over everything, though just see me after lab if anything is still left



What they used to teach in 107, back in the day

# Lecture Recap!



~10 minutes



**WHAT WAS  
HE COOKING?**



# A Brief History of C Strings

A C string is an **array of characters** (`char []`) with a **null-terminating character** `'\0'` at the end. **It is vital that a string be properly null-terminated.**

- ❏ For common operations (e.g., comparing/concatenating/searching strings), we can use string library functions in `<string.h>`

| 0x10 | 0x11 | 0x12 | 0x13 | 0x14 | 0x15 |
|------|------|------|------|------|------|
| h    | e    | l    | l    | 0    | \0   |

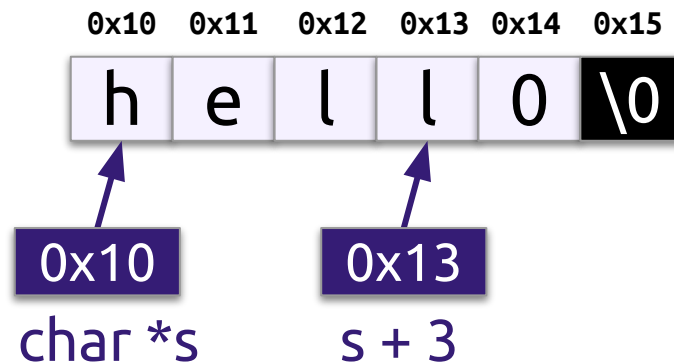


# A Brief History of C Strings

A C string is an **array of characters** (`char []`) with a **null-terminating character** `'\0'` at the end. **It is vital that a string be properly null-terminated.**

❑ For common operations (e.g., comparing/concatenating/searching strings), we can use string library functions in `<string.h>`

❑ C strings can be represented as **`char*`**: **pointers to characters**, and we can move those pointers to  advance through chars.



❑ **When passing strings as parameters, we always pass them as `char*` (a pointer to the first char).** Thus, chars changed inside a function will persist outside of it.

In summary, we usually find it more convenient and powerful to **use a `char*` / string pointer**  **to access, share, and manipulate the data** inside an array of characters.



# C Stings: Stack vs Data Segment

→ Characters of strings live either on the **stack (modifiable)** or in the **data segment (read-only)**, depending on how they're declared.



**Exercise! What's the difference?**

```
char myArr[6] = "Hello";
```

```
char* myPtr = "Hello";
```

Stack



Data Segment





# C Stings: Stack vs Data Segment

→ Characters of strings live either on the **stack (modifiable)** or in the **data segment (read-only)**, depending on how they're declared.



Exercise! What's the difference?

```
char myArr[6] = "Hello";
```

```
char* myPtr = "Hello";
```

Stack

H e l l o \0

Array of chars!

Data Segment

H e l l o \0

String literal



# C Stings: Stack vs Data Segment

- The characters of **myArr** live on the stack → **modifiable** ✓.
- The characters of **myPtr** live on read-only data segment → **unmodifiable** ✗.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char myArr[6] = "Hello";
    char *myPtr = "Hello";

    myArr[0] = 'J';
    printf("%s\n", myArr); // Prints "Jello"

    myPtr[0] = 'J'; // Segmentation fault !!
    printf("%s\n", myPtr);
    return 0;
}
```



Stack



J e l l o \0

Array of chars!

Data Segment



H e l l o \0

String literal

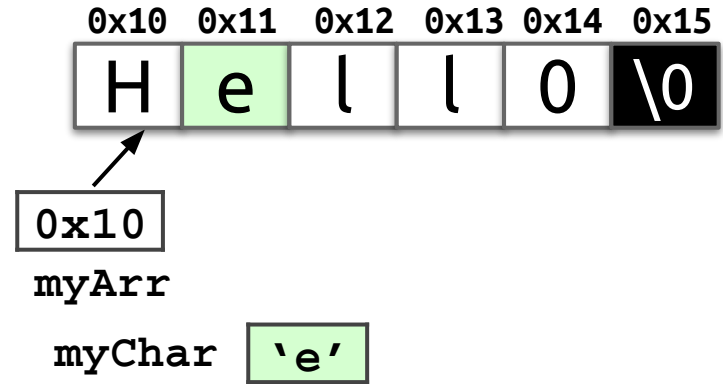


# Pointer Dereferencing (\*), Referencing (&)



**Exercise!** Rewrite the second line using pointer arithmetic

```
char myArr[6] = "Hello";  
char myChar = myArr[1]; // e
```



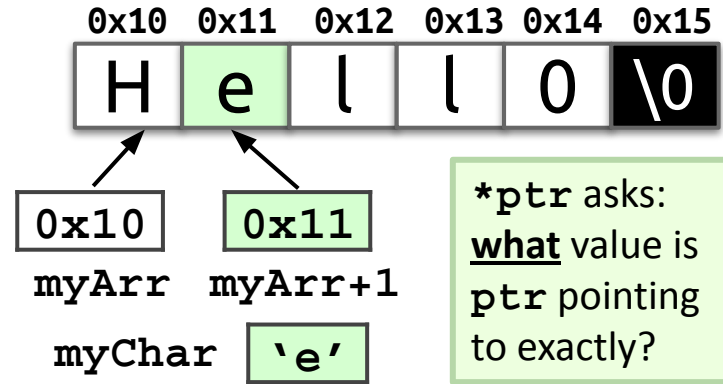


# Pointer Dereferencing (\*), Referencing (&)



**Exercise!** Rewrite the second line using pointer arithmetic

```
char myArr[6] = "Hello";  
char myChar = myArr[1]; // e  
char myChar = *(myArr + 1);
```





# Pointer Dereferencing (\*), Referencing (&)



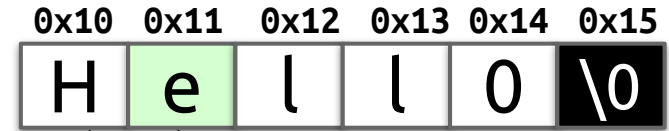
**Exercise!** Rewrite the second line using pointer arithmetic

```
char myArr[6] = "Hello";  
char myChar = myArr[1]; // e  
char myChar = *(myArr + 1);
```



**Exercise!** What's the difference?

```
char myArr[6] = "Hello";  
char* myPtr = myArr;  
char* myPtr2 = &(myArr[0]);
```



0x10

myArr

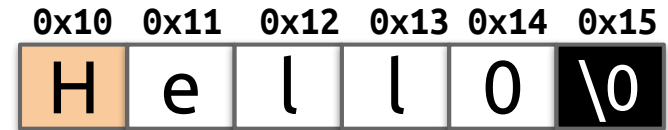
myChar

0x11

myArr+1

'e'

\*ptr asks:  
what value is  
ptr pointing  
to exactly?



0x10

myPtr

???

myPtr2

myArr[0]

'H'



# Pointer Dereferencing (\*), Referencing (&)



**Exercise!** Rewrite the second line using pointer arithmetic

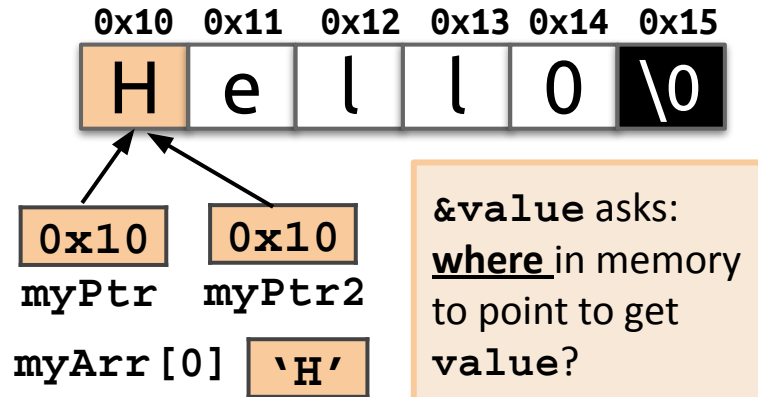
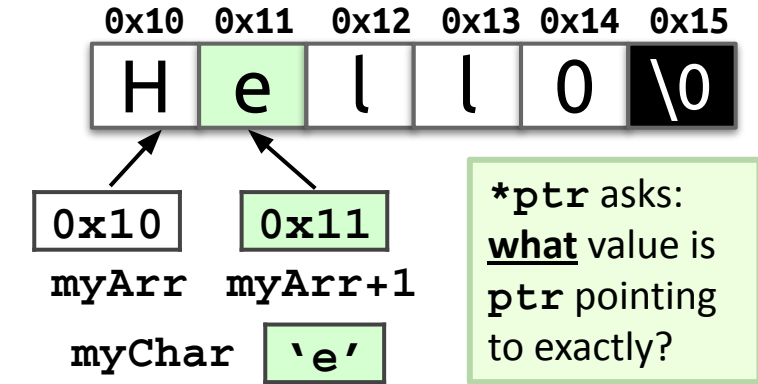
```
char myArr[6] = "Hello";  
char myChar = myArr[1]; // e  
char myChar = *(myArr + 1);
```



**Exercise!** What's the difference?

```
char myArr[6] = "Hello";  
char* myPtr = myArr;  
char* myPtr2 = &(myArr[0]);
```

**Answer: None!** They both store a pointer to first element of myArr!





# Relevant String Library Functions for Lab

```
int strcmp(str1, str2)
```

Returns **0** if str1 and str2 are identical  
Returns **negative** if str1 comes first in the dictionary, **positive** if str2 comes first

|      |       |      | returns |    |
|------|-------|------|---------|----|
| str1 | black | str2 | pink    | <0 |
| str1 | pink  | str2 | pink    | 0  |

```
size_t strlen(str)
```

Returns length of string, **not including the null-terminating \0**

|     |                  | returns |
|-----|------------------|---------|
| str | n y a n c a t \0 | 7       |

```
char *strncpy(dest, src, n)
```

**Copies n bytes** from source into destination (1 char = 1 byte), but doesn't guarantee result will be null-terminated unless the **\0** is copied over

| src | n | y | a | n | c | a | t | \0 | n |
|-----|---|---|---|---|---|---|---|----|---|
|     |   |   |   |   |   |   |   |    | 4 |

| returns | dest | n | y | a | n |  |  |
|---------|------|---|---|---|---|--|--|
| char*   |      |   |   |   |   |  |  |



# More String Functions for Lab!

`char *strstr(str1, str2)`

Returns pointer to **first occurrence** of `str2` in `str1`, or `NULL` if not found.

`str1`   `a b e n t w o b e n s \0`  
Returns `char*`   `str2`   `b e n \0`

`size_t strspn(str1, str2)`

Returns **length** of initial portion of `str1` that only contains characters from `str2`

`str1`   `= + = f l o m o = l a g \0`  
Returns `3`   `str2`   `= + \0`

`size_t strcspn(str1, str2)`

Returns **length** of initial portion of `str1` that **doesn't** contain characters from `str2`

`str1`   `y a g a = + = = g a t e s \0`  
Returns `4`   `str2`   `= + \0`

# Happy to take questions!

We vibing with course material so far?





# Check-Off Question!



How would we write a method `isSuffix(char *t, char *s)` that returns `true` if string `s` is a **suffix** of `t`, and `false` otherwise?

t n y a n c a t

s c a t

suffix?

t b e n y a n y a n

s n y a n

suffix?

t n y a n c a t

s n y a n

suffix?

t n y a n c a t

s c a r

suffix?

t c a t

s n y a n c a t

suffix?



# Check-Off Question!



How would we write a method `isSuffix(char *t, char *s)` that returns `true` if string `s` is a **suffix** of `t`, and `false` otherwise?

t n y a n c a t

s c a t

suffix?



t b e n y a n y a n

s n y a n

suffix?



t n y a n c a t

s n y a n

suffix?



t n y a n c a t

s c a r

suffix?



t c a t

s n y a n c a t

suffix?





# isSuffix(t, s) Thought Process

How would we write a method `isSuffix(char *t, char *s)` that returns `true` if string `s` is a **suffix** of `t`, and `false` otherwise?

t n y a n c a t

s c a t

suffix?



t n y a n c a t

s c a r

suffix?



? For this string problem, questions I might ask myself:



# isSuffix(t, s) Thought Process

How would we write a method `isSuffix(char *t, char *s)` that returns `true` if string `s` is a **suffix** of `t`, and `false` otherwise?

t n y a n c a t

s c a t

suffix?



t n y a n c a t

s c a r

suffix?



? For this string problem, questions I might ask myself:

→ What string functions could be useful here?

`strcmp(str1, str2)`, since I want to eventually compare strings / substrings



# isSuffix(t, s) Thought Process

How would we write a method `isSuffix(char *t, char *s)` that returns `true` if string `s` is a **suffix** of `t`, and `false` otherwise?

t n y a n c a t

s c a t

suffix?



t n y a n c a t

s c a r

suffix?



? For this string problem, questions I might ask myself:

→ What string functions could be useful here?

`strcmp(str1, str2)`, since I want to eventually compare strings / substrings

→ What parts of each string do I want to compare?

The **entirety of suffix candidate s**, and the **back part / last len(s) characters** of string `t`.



# isSuffix(t, s) Thought Process

How would we write a method `isSuffix(char *t, char *s)` that returns `true` if string `s` is a **suffix** of `t`, and `false` otherwise?

t n y a n c a t

s c a t

suffix?



t n y a n c a t

s c a r

suffix?



? For this string problem, questions I might ask myself:

→ What string functions could be useful here?

`strcmp(str1, str2)`, since I want to eventually compare strings / substrings

→ What parts of each string do I want to compare?

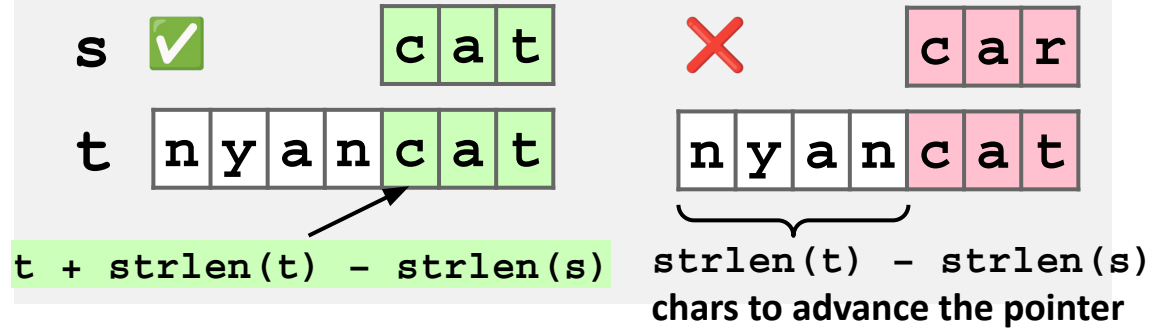
The **entirety of suffix candidate s**, and the **back part / last len(s) characters of string t**.

→ How do I get a `char*` / string that's just the back part of string `t`?

Easiest way: **pointer arithmetic!** Increment our `char* t` to point at a later character!

# isSuffix(t, s) Implementation

check if the last characters of **t** matches the suffix (note we're checking **strlen(s)** chars)



```
// checks if s matches the last strlen(s) characters of t
return strcmp(s, t + strlen(t) - strlen(s)) == 0;
```

# isSuffix(t, s) — Intercept Edge Case!

✗ not possible is  
suffix is longer

s n y a n c a t  
t c a t

else: check if the last characters of t matches the  
suffix (note we're checking `strlen(s)` chars)

s ✓ c a t  
t n y a n c a t

`t + strlen(t) - strlen(s)`

✗ c a r  
n y a n c a t

`strlen(t) - strlen(s)`  
chars to advance the pointer

```
if (strlen(s) > strlen(t)) {  
    return false; // impossible if suffix s is longer  
}  
// checks if s matches the last strlen(s) characters of t  
return strcmp(s, t + strlen(t) - strlen(s)) == 0;
```



## isSuffix(t, s) Solution

How would we write a method `isSuffix(char *t, char *s)` that returns `true` if string `s` is a **suffix** of `t`, and `false` otherwise?

t n y a n c a t

s c a t

suffix?



t n y a n c a t

s c a r

suffix?



```
if (strlen(s) > strlen(t)){  
    return false; // impossible if suffix s is longer  
}  
// checks if s matches the last strlen(s) characters of t  
return strcmp(s, t + strlen(t) - strlen(s)) == 0;
```



# Valgrind Magic Guide

CS107   Calendar   Handouts ▾   Lectures ▾   Assignments ▾   Labs ▾   Exams ▾   Gradebook   Getting Help

V

- Syllabus
- SCPD Information
- Honor Code
- Course Enrollment FAQ
- Course Reader
- Getting Started on Myth
- Unix Guide
- Emacs Guide
- The GDB Debugger
- Valgrind Guide**
- C Standard Library Guide
- Debugging Guide
- x86-64 Guide
- Callgrind Guide

check

Julie Zelenski and Chris Gregg, with modifications by Nick Troccoli

video.

detects memory leaks and memory errors. Some of the mismanagement of memory: allocating the wrong size, accessing memory after it was freed, overrunning a buffer, are tricky, as they can provide little debugging

erved problem back to underlying root cause can be help!

Memory Leaks

ues: memory *errors* and memory *leaks*. When a program dynamically allocates memory and forgets to later free it, it creates a leak. A memory leak generally won't cause a program to misbehave, crash, or give wrong answers, and is not an urgent situation. A memory error, on the other hand, is a red alert. Reading uninitialized memory, writing past the end of a piece of memory, accessing freed memory, and other memory errors can have significant consequences. Memory errors should never be treated casually or ignored. Although this guide describes about how to use Valgrind to find both, keep in mind that errors are by far the primary concern, and memory leaks can generally be resolved later.

Running A Program Under Valgrind

[/web.stanford.edu/class/archive/cs/cs107/cs107.1264/resources/valgrind.html](http://web.stanford.edu/class/archive/cs/cs107/cs107.1264/resources/valgrind.html)

Memory Errors Vs. Memory Leaks

Running A Program Under Valgrind

Finding Memory Errors

Finding Memory Leaks

Frequently Asked Questions

<https://web.stanford.edu/class/archive/cs/cs107/cs107.1264/resources/valgrind.html>



# String Functions Reference

| Function                                                                                                    | Description                                                                                                                                                                                                                                                     |
|-------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>strlen(<i>str</i>)</code>                                                                             | returns the # of chars in a C string (before null-terminating character).                                                                                                                                                                                       |
| <code>strcmp(<i>str1</i>, <i>str2</i>)</code> ,<br><code>strncmp(<i>str1</i>, <i>str2</i>, <i>n</i>)</code> | compares two strings; returns 0 if identical, <0 if <i>str1</i> comes before <i>str2</i> in alphabet, >0 if <i>str1</i> comes after <i>str2</i> in alphabet. <b>strncmp</b> stops comparing after at most <i>n</i> characters.                                  |
| <code>strchr(<i>str</i>, <i>ch</i>)</code><br><code>strrchr(<i>str</i>, <i>ch</i>)</code>                   | character search: returns a pointer to the first occurrence of <i>ch</i> in <i>str</i> , or <b>NULL</b> if <i>ch</i> was not found in <i>str</i> . <code>strrchr</code> find the last occurrence.                                                               |
| <code>strstr(<i>haystack</i>, <i>needle</i>)</code>                                                         | string search: returns a pointer to the start of the first occurrence of <i>needle</i> in <i>haystack</i> , or <b>NULL</b> if <i>needle</i> was not found in <i>haystack</i> .                                                                                  |
| <code>strcpy(<i>dst</i>, <i>src</i>)</code> ,<br><code>strncpy(<i>dst</i>, <i>src</i>, <i>n</i>)</code>     | copies characters in <i>src</i> to <i>dst</i> , including null-terminating character. Assumes enough space in <i>dst</i> . Strings must not overlap. <b>strncpy</b> stops after at most <i>n</i> chars, and <u>does not</u> add null-terminating char.          |
| <code>strcat(<i>dst</i>, <i>src</i>)</code> ,<br><code>strncat(<i>dst</i>, <i>src</i>, <i>n</i>)</code>     | concatenate <i>src</i> onto the end of <i>dst</i> . <b>strncat</b> stops concatenating after at most <i>n</i> characters. <u>Always</u> adds a null-terminating character.                                                                                      |
| <code>strspn(<i>str</i>, <i>accept</i>)</code> ,<br><code>strcspn(<i>str</i>, <i>reject</i>)</code>         | <b>strspn</b> returns the length of the initial part of <i>str</i> which contains <u>only</u> characters in <i>accept</i> . <b>strcspn</b> returns the length of the initial part of <i>str</i> which does <u>not</u> contain any characters in <i>reject</i> . |



# Q1: Valgrind!

In the `lab2` directory, read over `buggy.c`, which has two memory errors, as outlined in the **lab handout**. What exactly is going on? What does the memory magician Valgrind think?

Constellate into groups, move to next exercise if you finish early. Please lmk if you have questions!



~ 20 min + 10 min all-lab discussion



**Tips for Reading Valgrind Output:** Look for what error messages are saying certainly (uninitialized value, invalid read, etc.) – but also **any references to functions / lines of code we've written**, e.g., `helper_function_1` (`buggy.c: 32`), which are good places to look!

stop glamorizing “the Grind” and start glamorizing whatever this is





# Review of Error 1

```
==10750== Use of uninitialised value of size 8
==10750==    at 0x4C30F62: strlen (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==10750==    by 0x4006D3: makeError1 (buggy.c:17)
==10750==    by 0x4007F6: main (buggy.c:48)
==10750==
==10750== Invalid read of size 1
==10750==    at 0x4C30F62: strlen (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==10750==    by 0x4006D3: makeError1 (buggy.c:17)
==10750==    by 0x4007F6: main (buggy.c:40)
==10750== Address 0x100000000 is not stack'd, malloc'd or (recently)
free'd
==10750==
==10750==
==10750== Process terminating with default action of signal 11
(SIGSEGV)
==10750== Access not within mapped region at address 0x100000000
==10750==    at 0x4C30F62: strlen (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==10750==    by 0x4006D3: makeError1 (buggy.c:17)
==10750==    by 0x4007F6: main (buggy.c:40)
==10750== If you believe this happened as a result of a stack
==10750== overflow in your program's main thread (unlikely but
```

These all refer to the same bug (on line 17 of buggy.c), where we call strlen:

```
17  printf("strlen(uninitializedPtr) =
%zu\n", strlen(uninitializedPtr));
```

This string comes from the previous line:

```
16  char* uninitializedPtr;
```

**What's wrong here?**

**Calling strlen() on an uninitialized pointer causes a segmentation fault.**

The program is reading memory at an invalid / garbage location ! !



## Review of Error 2

Unlike Error 1, Error 2 is more asymptomatic i.e. it doesn't cause a segmentation fault or even the program to crash

```
bbyan@myth57:~/CS107_TA/lab2$ ./buggy 1
Segmentation fault
bbyan@myth57:~/CS107_TA/lab2$ ./buggy 2
strlen(non-terminated) = 3
```

Here's where valgrind comes in handy! With **valgrind ./buggy 2**,

```
==11338== Conditional jump or move depends on uninitialised value(s)
==11338==    at 0x4C30F78: strlen (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==11338==    by 0x400722: makeError2 (buggy.c:25)
==11338==    by 0x400808: main (buggy.c:42)
```



What **string handling error** is likely cause of this Valgrind report?

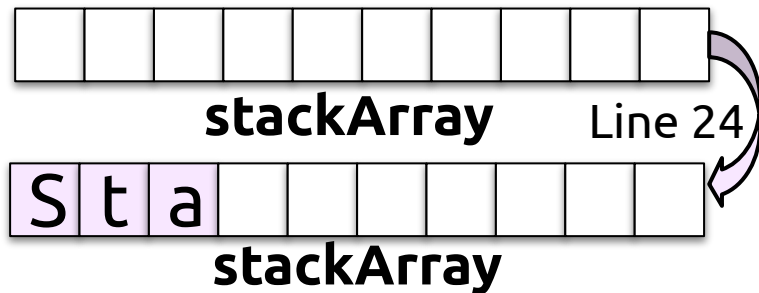


## Review of Error 2 (Check-Off Q)

```
==11338== Conditional jump or move depends on uninitialised value(s)
==11338==    at 0x4C30F78: strlen (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==11338==    by 0x400722: makeError2 (buggy.c:25)
==11338==    by 0x400808: main (buggy.c:42)
```

The bug occurs **around line 25** of buggy.c – let's take a close look!

```
23 char stackArray[10];
24 strncpy(stackArray, "Stanford", 3);
25 printf("strlen(non-terminated) = %zu\n",
        strlen(stackArray));
```



What's wrong here?

**No null-terminating character!**

- ❑ Calling **strlen** on a string missing a null-terminator.
- ❑ The program keeps reading past the last character and into uninitialized memory ! in search of `\0`.
- ❑ The outputted lengths may not be consistent as a result.





# Valgrind Exercise Takeaways

- 1 Reading Valgrind output takes some time and practice, like reading sheet music
- 2 Memory errors may be asymptomatic / not immediately apparent or visible, but that doesn't excuse it! :(
- 3 Run Valgrind before submitting assignments 



## Q2: `strtok()` code study

**Goal:** Understand the implementation of the `strtok` library function, so that **eventually you can engineer an improved version in Assign2**.

Constellate into groups, move to next exercise if you finish early. Please lmk if you have questions!

**Advice:** Choose an example string, and diagram/draw out what `strtok()` is doing on paper! Track where each string is pointing / how it's updated



~ 30 min + 10 min all-lab discussion



`strtok`  
from lab



`scan_token`  
for Assign2

## `char *musl_strtok(char *s, const char *delimiters)`

- Splits **string** `s` into substrings or “tokens” that are separated by delimiters
- Returns **one token per call** to `strtok`
- Maintains state across calls -- **calls are not independent!**

```
str: "web.stanford.edu/class/archive/cs/cs107"  
delimiters: "/"
```

The tokens would be:

“web.stanford.edu”

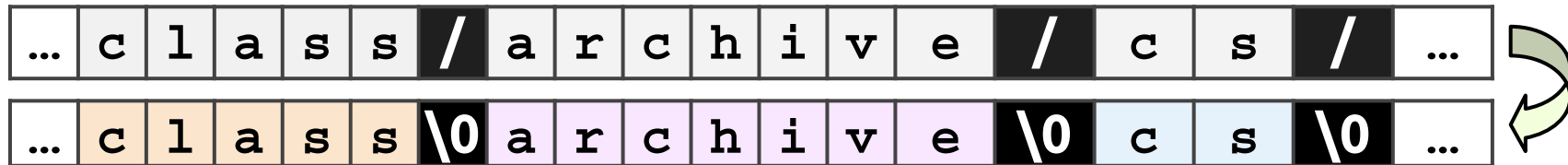
“class”

“archive”

“cs”

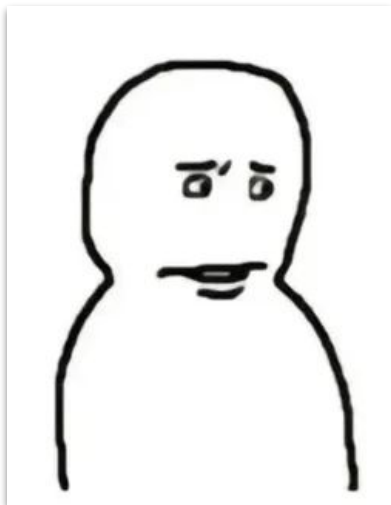
“cs107”

To do this, instead of creating new strings, `strtok` chops the existing string by overriding **delimiters** with **null-terminators** when needed, e.g., after everything,





## strtok() code



Let's break it  
down puzzle  
piece by piece! 🧩

```
1 char *musl_strtok(char *s, const char *delimiters) {
2     static char *p = NULL;
3
4     if (s == NULL) {
5         s = p;
6         if (s == NULL) {
7             return NULL;
8         }
9     }
10
11     s += strspn(s, delimiters);
12     if (*s == '\0') {
13         p = NULL;
14         return p;
15     }
16
17     p = s + strcspn(s, delimiters);
18     if (p[0] != '\0') {
19         p[0] = '\0';
20         p++;
21     } else {
22         p = NULL;
23     }
24
25     return s;
26 }
```

What is static p tracking,  
and how is s updated?

How/where is s being  
moved by **strcspn**?

How is p being moved  
by **strcspn**? Where/why  
is the string modified?

## Check-Off Q: What if we pass in a string constant?

What happens if we uncomment line 26 in strtok\_example.c?

```
26 token = strtok("red-green-blue", delimiters);
```

# Check-Off Q: What if we pass in a string constant?

What happens if we uncomment line 26 in strtok\_example.c?

```
26 token = strtok("red-green-blue", delimiters);
```

```
// lines 17-20 of strtok
p = s + strcspn(s, delimiters);
if (p[0] != '\0') {
    p[0] = '\0';
    P++;
}
```

We get a **segmentation fault**  
(protection violation) !!

Because the code attempts to edit  
**p[0]** of a **read-only string literal p**.  
That's not good :(

Hi can I enter  
and modify  
the string?

strtok



|   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|
| r | e | d | - | g | r | e | e | n | - | b | l | u | e | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|

**Read-Only Data Segment**



# Static Variables: Kind of Like Vampires

- ❑ **Lifetime is eternal the entire program execution** (not just single run of the function)
- ❑ Static variables are **only initialized once** – and **if defined in a function, scope is still limited to that function** (think outside scope as sunlight, so vampire needs to stay inside)

```
static char *p = NULL;   
if (s == NULL) {  
    s = p;  
    if (s == NULL) {  
        return NULL;  
    }  
}
```



# Static Variables: Kind of Like Vampires

- ❑ **Lifetime is eternal** the entire program execution (not just single run of the function)
- ❑ Static variables are **only initialized once** – and **if defined in a function, scope is still limited to that function** (think outside scope as sunlight, so vampire needs to stay inside)

```
static char *p = NULL;   
if (s == NULL) {  
    s = p;  
    if (s == NULL) {  
        return NULL;  
    }  
}
```

The `s == NULL` checks if there's an existing tokenization to continue

→ If true, we set `s` to `p`, which keeps track of **where we left off in the original string**.

→ If false, start new tokenization with `s`  
Static pointer `p` is only initialized once to `NULL`.



**s = p**

Metaphor: Tokens as  location strings, and  delimiters as roads to get to them



## 🧩 Check-Off: How is `strspn` used when tokenizing?

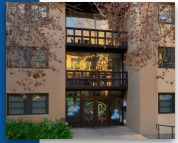
```
s += strspn(s, delimiters);  
if (*s == '\\0') {  
    p = NULL;  
    return p;  
}
```

delimiters

= ==

= = = l l a g a == c o d a \\0

↑  
s



## 🧩 Check-Off: How is **strspn** used when tokenizing?

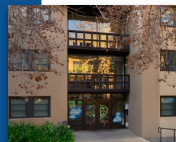
```
s += strspn(s, delimiters);  
if (*s == '\\0') {  
    p = NULL;  
    return p;  
}
```

**strspn** is used to advance past any delimiters at the start of the string.

- If we've reached the end (`\\0`), we exit the function
- Otherwise, we've **reached a new token** to extract → next slide!

**3 chars** to move past delimiters at start

**strspn(s, delimiters)** **3** **delimiters** **=** **==**





# Check-off Q: How is **strcspn** used?

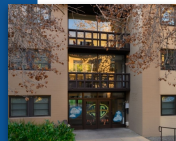
```
p = s + strcspn(s, delimiters);  
if (p[0] != '\\0') {  
    p[0] = '\\0';  
    p++;  
} else {  
    p = NULL;  
}  
return s;
```

delimiters = 

|   |   |
|---|---|
| = | ≡ |
|---|---|

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |     |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|-----|
| = | = | = | l | l | a | g | a | ≡ | ≡ | c | o | d | a | \\0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|-----|

↑  
**s**





# Check-off Q: How is **strcspn** used?

```
p = s + strcspn(s, delimiters);  
if (p[0] != '\0') {  
    p[0] = '\0';  
    p++;  
} else {  
    p = NULL;  
}  
return s;
```

**strcspn** finds the next delimiter/endpoint of the current token, and moves pointer **p** there.

→ If **p[0] = \0**, we've reached the end of the whole string, and we're done!

→ If **p[0] != \0**, we set a **null-terminator** there (to truncate **s** to just that token) and **p++**; to continue moving.

**5 chars** to reach the next  
delimiter (token endpoint)

**strcspn(s, delimiters)**

**5**

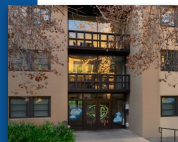
**delimiters**

**=** **≡**

**= = = 1 1 a g a ≡ ≡ c o d a \0**

**s**

**p=s+5**



# Check-off Q: How is **strcspn** used?

```
p = s + strcspn(s, delimiters);  
if (p[0] != '\0') {  
    p[0] = '\0';  
    p++;  
} else {  
    p = NULL;  
}  
return s;
```

**strcspn** finds the next delimiter/endpoint of the current token, and moves pointer **p** there.

→ If **p[0] = \0**, we've reached the end of the whole string, and we're done!

→ If **p[0] != \0**, we set a **null-terminator** there (to truncate **s** to just that token) and **p++**; to continue moving.

**5 chars** to reach the next delimiter (token endpoint)

**strcspn(s, delimiters)** **5** **delimiters** **=** **≡**

 **= = = 1 1 a g a \0 ≡ c o d a \0** 

↑  
**s**

↑  
**p = s + 5**

↑  
**p++;**

Note **s** is now the str/token “llaga”, which we return!



## Next call to strtok: repeat steps!

Find start of next token

```
s += strspn(s, delimiters);
```

delimiters



Find where token ends

```
p = s + strcspn(s, delimiters);
```



# Next call to strtok: repeat steps!

Find start of next token

```
s += strspn(s, delimiters);
```

delimiters

|   |   |
|---|---|
| = | ≡ |
|---|---|

|   |   |   |   |   |    |
|---|---|---|---|---|----|
| ≡ | c | o | d | a | \0 |
|---|---|---|---|---|----|



s



s +

|   |
|---|
| 1 |
|---|

1 chars to move past delimiters at start



strspn(s, delimiters)

|   |
|---|
| 1 |
|---|

Find where token ends

```
p = s + strcspn(s, delimiters);
```



# Next call to strtok: repeat steps!

Find start of next token

```
s += strspn(s, delimiters);
```

delimiters

= ≡

|   |   |   |   |   |    |
|---|---|---|---|---|----|
| ≡ | c | o | d | a | \0 |
|---|---|---|---|---|----|



s

s+1

1

1 chars to move past delimiters at start



strspn(s, delimiters) 1

Find where token ends

```
p = s + strcspn(s, delimiters);
```

|   |   |   |   |   |    |
|---|---|---|---|---|----|
| ≡ | c | o | d | a | \0 |
|---|---|---|---|---|----|



s

p=s+

4

4 chars to reach the next  
delimiter or \0 (token endpoint)



strcspn(s, delimiters) 4

Note s is now the final str/  
token “coda”, which we return!  
And we are **all done** tokenizing  
the original string yay! 🎉🎉



# strtok() Exercise Takeaways

1

I've found **107 to be as much an arts and crafts course as a programming one**: always have a paper / tablet at your side to sketch diagrams, examples, or illustrate your thoughts!

2

**Start with tangible examples**, e.g., choose and work with a small string like  "red-green-blue", and see if you can find patterns!

3

Feel free to come up with metaphors (even as bad as mine lol), personification, **storytelling if it helps with code understanding!**

4

All roads lead to ~~Rome~~ CoDa — legit see everyone here haha



# Parting Thoughts: NULL vs. \0 vs. ""

- **NULL: is a pointer** (specifically a pointer with numeric value 0)
  - **p = NULL** changes a pointer
- **\0: is a character** (a character with numeric value 0)
  - **\*p = '\0'** changes a character
- **"": is the empty string** (a string consisting of just a null terminator)
  - **p = ""** points to a string constant with a single null terminator

## Disambiguation (disambiguation)

From Wikipedia, the free encyclopedia

For information on disambiguation of topic names in Wikipedia, see [Wikipedia:Disambiguation](#).

Disambiguation usually refers to **word-sense disambiguation**, the process of identifying which meaning of a word is used in context.

**Disambiguation** may also refer to:

- Sentence boundary disambiguation, the problem in natural language processing of deciding where sentences begin and end
- Syntactic disambiguation, the problem of resolving syntactic ambiguity
- Semantic disambiguation, the problem of resolving semantic ambiguity
- Memory disambiguation, a set of microprocessor execution techniques

Look up *disambiguation*, *disambiguate*, *disambiguations*, or *disambiguator* in Wiktionary, the free dictionary.

## Music [\[ edit \]](#)

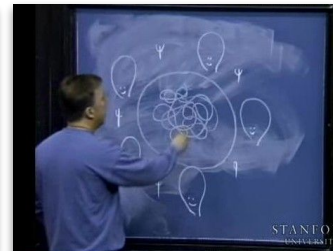
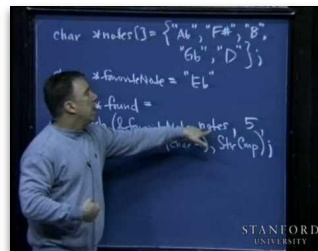
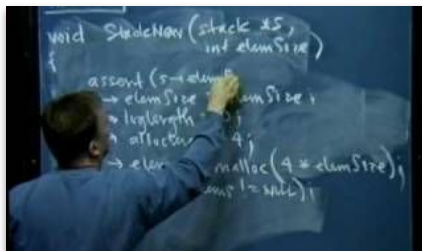
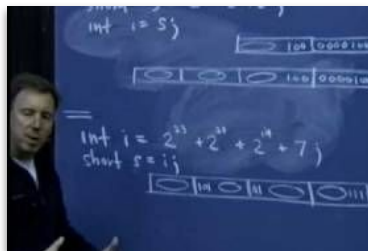
- *Ø (Disambiguation)*, a 2010 album by Underoath
- *Disambiguation* (Pandelis Karayorgis album), a 2002 album by Pandelis Karayorgis and Mat Maneri

## See also [\[ edit \]](#)

- *Ambiguity*, an attribute of any concept, idea, statement or claim whose meaning, intention or interpretation cannot be definitively resolved



Anonymous feedback form, appreciate any lab/teaching feedback! ❤️



# Thank you and have a beautiful week!

- Please reach out to me if you have any questions, snack recommendations, or topics you want covered!
-  Contact: [bbyan@stanford.edu](mailto:bbyan@stanford.edu), email or over Slack
- I always appreciate and value any feedback regarding labs, materials, teaching, office hours, and so on!



The Jerryland ~~rollercoaster~~ calm Ferris wheel continues

