



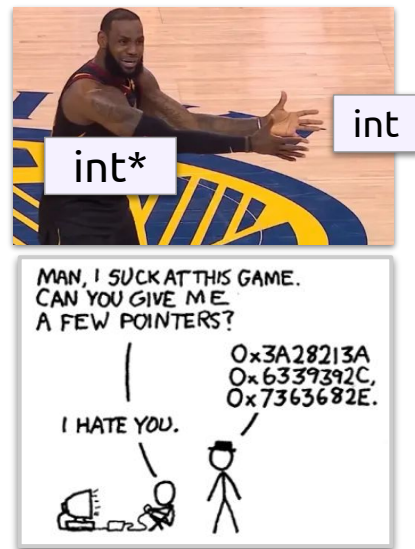
CS107 Lab 3 – Welcome Back!

👉 Pointers, Arrays and the Heap

Ben Yan, Winter 2026 🌨️

Slides Link: stanford.edu/~bbyan/cs107 :)

🎵 My Song of Week: Yuusha by YOASOBI



Stanford | ENGINEERING
Computer Science

Welcome Back to Lab!

*Hope that Assign2 went / is going well, and that
you're having fun learning string theory!*



Winter



Week 4



Spring



We have office hours each week in CoDa :) Feel free to stop by to work, get Voyager coffee, ask questions, or chat with Jerry!

Getting Setup for Lab 3



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

```
git clone /afs/ir/class/cs107/repos/lab3/shared lab3
```

- ❑ 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



Lecture Recap!



~10 minutes





C is Pass by Value

When passing parameters, **C passes a copy of whatever is passed.**

- ❑ To change values, and have those changes persist outside the function, **we must** 🙌 **pass a pointer to the value** (i.e. its address in memory).
- ❑ For instance, to change an **int**, we have to pass in an **int *** (pointer to an int)
- ❑ To change a **char*** (string), we have to pass in a **char**** (pointer to string).



C is Pass by Value

When passing parameters, **C passes a copy of whatever is passed.**

- ❑ To change values, and have those changes persist outside the function, **we must** 🙌 **pass a pointer to the value** (i.e. its address in memory).
- ❑ For instance, to change an **int**, we have to pass in an **int *** (pointer to an int)
- ❑ To change a **char*** (string), we have to pass in a **char**** (pointer to string).



Short Example

```
function mischief(int* ptrCopy) {  
    *ptrCopy = 701;  
}  
  
int main() {  
    int num = 107;  
    int* ptr = &num;  
    mischief(ptr);  
}
```



C is Pass by Value

When passing parameters, **C passes a copy of whatever is passed.**

- ❑ To change values, and have those changes persist outside the function, **we must** 🙌 **pass a pointer to the value** (i.e. its address in memory).
- ❑ For instance, to change an **int**, we have to pass in an **int *** (pointer to an int)
- ❑ To change a **char *** (string), we have to pass in a **char **** (pointer to string).



Short Example

```
function mischief(int* ptrCopy) {  
    *ptrCopy = 701;  
}  
  
int main() {  
    int num = 107;  
    int* ptr = &num;  
    mischief(ptr);  
}
```

num

107

0x10



C is Pass by Value

When passing parameters, **C passes a copy of whatever is passed.**

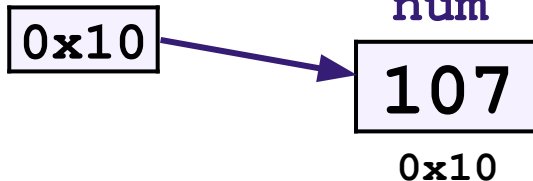
- ❑ To change values, and have those changes persist outside the function, **we must** 🙌 **pass a pointer to the value** (i.e. its address in memory).
- ❑ For instance, to change an **int**, we have to pass in an **int *** (pointer to an int)
- ❑ To change a **char*** (string), we have to pass in a **char**** (pointer to string).



Short Example

```
function mischief(int* ptrCopy){  
    *ptrCopy = 701;  
}  
  
int main(){  
    int num = 107;  
    int* ptr = &num;  
    mischief(ptr);  
}
```

ptr





C is Pass by Value

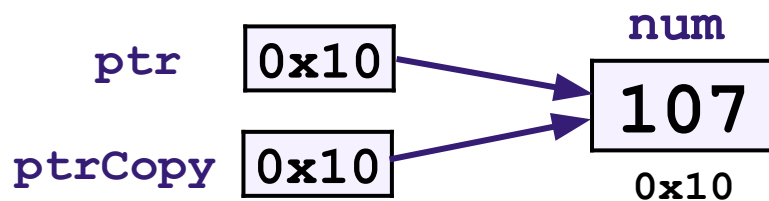
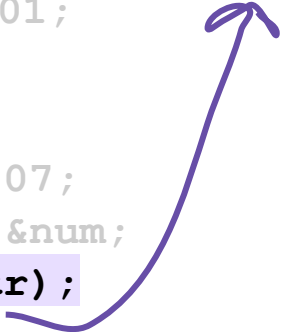
When passing parameters, **C passes a copy of whatever is passed.**

- ❑ To change values, and have those changes persist outside the function, **we must** 🙌 **pass a pointer to the value** (i.e. its address in memory).
- ❑ For instance, to change an **int**, we have to pass in an **int*** (pointer to an int)
- ❑ To change a **char*** (string), we have to pass in a **char**** (pointer to string).



Short Example

```
function mischief(int* ptrCopy){  
    *ptrCopy = 701;  
}  
  
int main(){  
    int num = 107;  
    int* ptr = &num;  
    mischief(ptr);  
}
```





C is Pass by Value

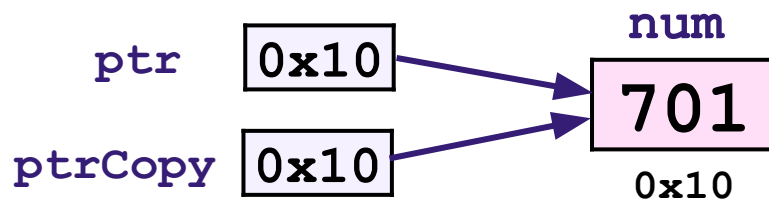
When passing parameters, **C passes a copy of whatever is passed.**

- ❑ To change values, and have those changes persist outside the function, **we must** 🙌 **pass a pointer to the value** (i.e. its address in memory).
- ❑ For instance, to change an **int**, we have to pass in an **int*** (pointer to an int)
- ❑ To change a **char*** (string), we have to pass in a **char**** (pointer to string).



Short Example

```
function mischief(int* ptrCopy){  
    *ptrCopy = 701;  
}  
  
int main(){  
    int num = 107;  
    int* ptr = &num;  
    mischief(ptr);  
}
```



Though the pointer is itself duplicated, **it's still pointing to the same value** in memory, and so **we can directly modify that value!**



Complete the sentence!

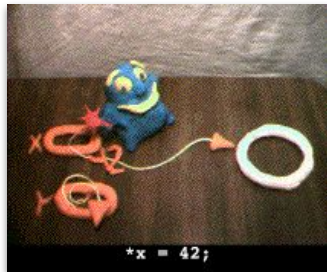


Pointers are used for ...





Pointers are used for lots of things in C!!



Pointers are used for ...



Pointing to a **location in memory**

Pointing to a contiguous series of locations in memory (i.e. **arrays**)

(Memorable) data types, e.g., **char *** as pointer representation for **strings**

Passing in **modifiable references** to an object (*pointers to variables, double ptrs*)

Addresses of **dynamically allocated memory** (pointers returned by **malloc**)



Today's Main Takeaway

<https://starwarsintrogenerator.com/scroller?u=22tb76zi>

*To have the changes to a value
persist outside of a function,
we must
pass a pointer to the value!*

Don't forget :)

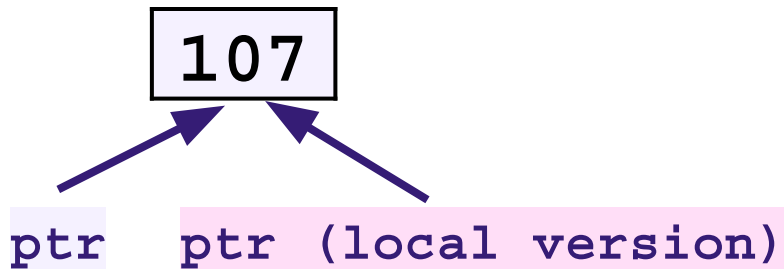


Pointer Freeing Exercise!

Your Sith Lord coworker suggests that you use this function below as a “**safer**” **free function**, to prevent accidentally using **ptr** from that point forward (by also making it **NULL**).

What could possibly go wrong?

```
void free_and_null(void *ptr){  
    free(ptr);  
    ptr = NULL;  
}
```



- Will this function correctly **free** the client's pointer?
- Will correctly set it to **NULL**?



Pointer Freeing Exercise!

Your Sith Lord coworker suggests that you use this function below as a “**safer**” **free function**, to prevent accidentally using **ptr** from that point forward (by also making it **NULL**).

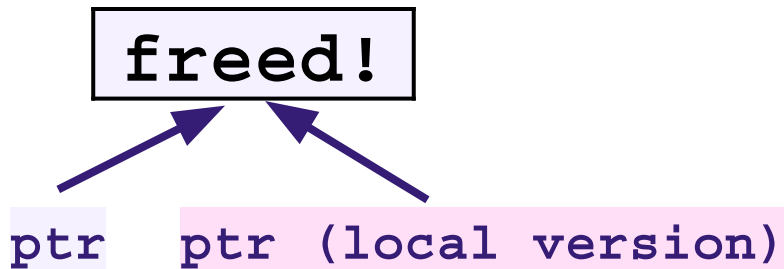
What could possibly go wrong?

```
void free_and_null(void *ptr){  
    free(ptr);  
    ptr = NULL;  
}
```

- Will this function correctly **free** the client’s pointer?
- Will correctly set it to **NULL**?



Yes, as our both pointers point to the same block of memory, so it’ll be freed.





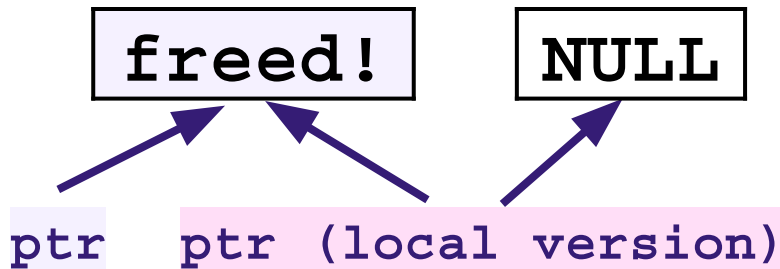
Pointer Freeing Exercise!

Your Sith Lord coworker suggests that you use this function below as a “**safer**” **free function**, to prevent accidentally using **ptr** from that point forward (by also making it **NULL**).

What could possibly go wrong?

```
void free_and_null(void *ptr){  
    free(ptr);  
    ptr = NULL;  
}
```

- Will this function correctly **free** the client's pointer?
- Will correctly set it to **NULL**?



No; it only sets **free_and_null**'s own copy of the **ptr** to **NULL**. You would have to pass a pointer to the client's pointer, to **NULL** it out.



Notes on Arrays and Pointers

- ❑ When passing **arrays as parameters**, we always pass them as a **pointer to the first element**

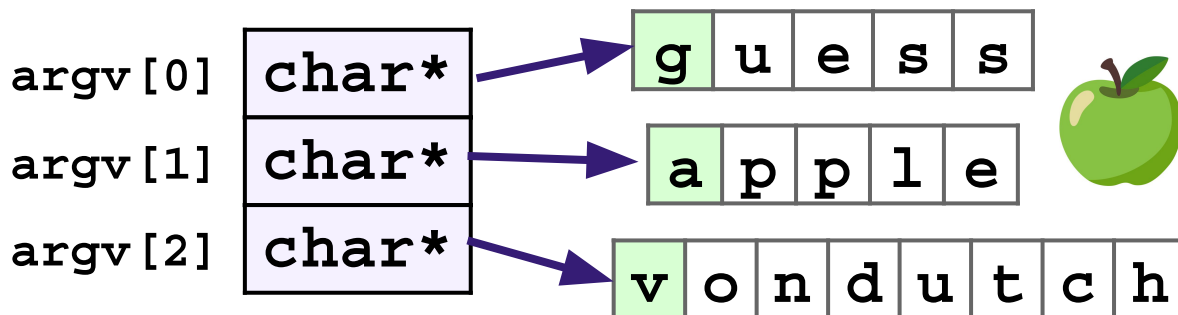
```
char album[5];
```

b	r	a	t	\n
---	---	---	---	----

```
play(album);
```

- ❑ **Array of pointers** = array where each element is a pointer to some data.
- ❑ For example, an **array of strings** stores **pointers to their first characters**, NOT the characters themselves. Those can be stored elsewhere in memory

Array of **char***
e.g., **char* argv[3];**

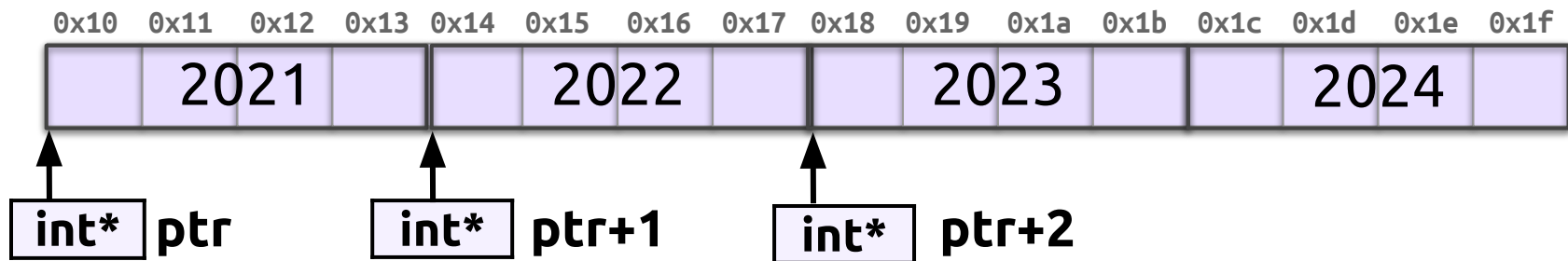




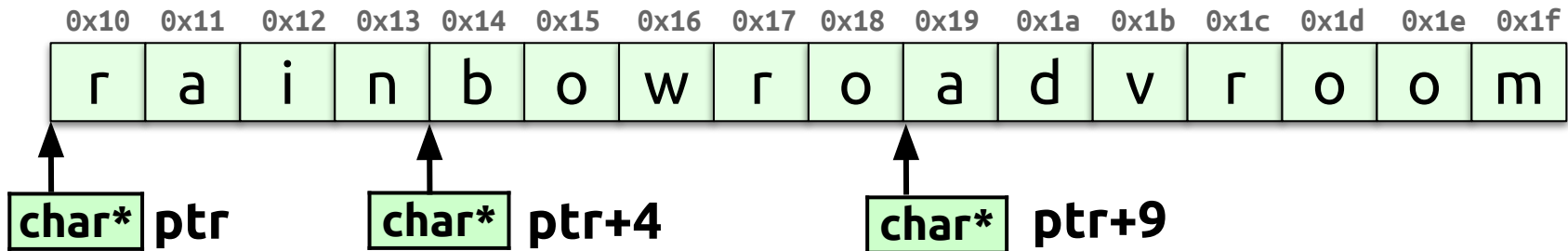
Pointers and Pointer Arithmetic

Pointer arithmetic works in terms of the **size of the data type being pointed to**

→ **+1** for an **int*** means advance one int, so **4 bytes**



→ **+1** for a **char*** means advance one char, so just **1 byte**



Happy to take any questions!

Are we vibing with course material so far?



We're definitely picking up some speed!



Helpful GDB Guide on Course Website

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

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

Debugging

and Nate Hardison, with modifications by Nick Troccoli and Lisa Yan

video.

have used a graphical debugger; these debuggers were to write your code, and allowed you to set breakpoints, variable values, among other features. In CS107, the rate program from your text editor, called `gdb` (the and-line debugger, meaning that you interact with it on ed commands. But it shares many similarities with dy used; it also allows you to set breakpoints, step able values. We recommend familiarizing yourself with ible.

- Running `gdb`
- Breakpoints
- Backtrace
- Controlling Execution
- Variables and Expressions
- Stack Frames
- Examining Memory
- Useful Tips
- Debugging Strategies
- `gdb` Commands Reference

<https://web.stanford.edu/class/archive/cs/cs107/cs107.1254/resources/gdb.html>



Q1: GDB Pointers and Arrays

In the newly cloned `lab3` directory, work with `code.c` and its program under GDB. Follow the instructions to learn more about pointers and arrays when they're passed in as parameters.

Constellate into groups; move into the next exercise if you finish early. Please lmk if you have any questions!



~ 15 min + 10 min all-lab discussion





GDB Tip: `layout src`

```
...ab/lab3 — ssh bbyan@myth.stanford.edu
(gdb) break main
Breakpoint 2 at 0x555555
(gdb) run
The program being debugg
Start it from the beginn
Starting program: /afs/.
[Thread debugging using
Using host libthread_db

Breakpoint 2, main (argc
52      int num = ar
(gdb) layout src
```

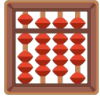


```
benjaminyan — bbyan@myth65: ~/CS107_TA/lab3 — ssh bbyan@myth.stanford.edu — 63x20
buggy.c
47      // this location was already freed!
48      free(another);
49      }
50
51      int main(int argc, char *argv[]) {
B> 52      int num = argc > 1 ? atoi(argv[1]) : 0;
53      char *str =  argc > 2 ? argv[2] : "Stanfor
54
55      if (num == 1) {
56          make_error_1(str);
57      } else if (num == 2) {
58          make_error_2();
59      } else if (num == 3) {

process 1100203 In: main      L52  PC: 0x555555555360
(gdb) info locals
num = 0
str = 0x7fffffe8b0 "\001"
(gdb)
```

As we're running GDB (i.e. stopped at a breakpoint), run **(gdb) layout src**

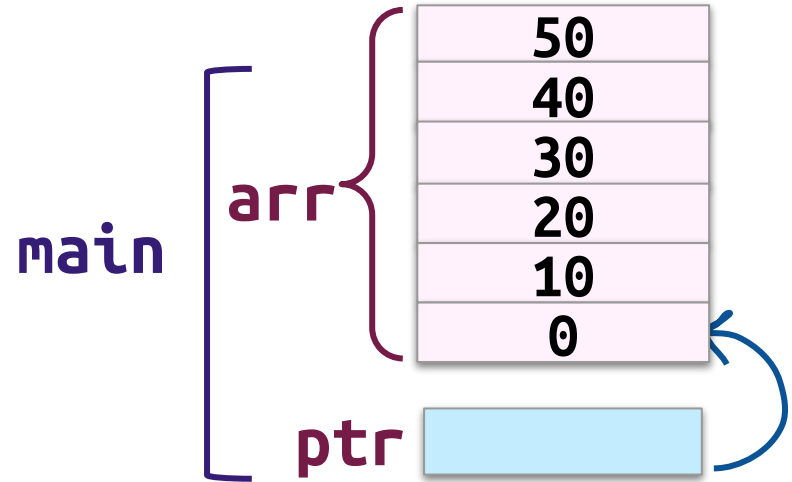
We have our **code** side-by-side with the debugger (with the **line we're about to run in GDB highlighted**) — neat right?

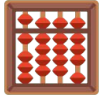


Q1: What does each expression value to?

```
int arr[1] = {0, 10, 20, 30, 40, 50};
```

```
(gdb) p *arr  
(gdb) p arr[1]  
(gdb) p &arr[1]  
(gdb) p *(arr+2)  
(gdb) p &arr[3] - &arr[1]  
  
(gdb) p sizeof(arr)  
(gdb) p arr = arr + 1
```





Q1: What does each expression value to?

```
int arr[1] = {0, 10, 20, 30, 40, 50};
```

(gdb) p *arr **0** (index-0 element of array)

(gdb) p arr[1] **10** (index-1 element)

(gdb) p &arr[1] **addr** of index-1 element

(gdb) p *(arr+2) **20** index-2 element

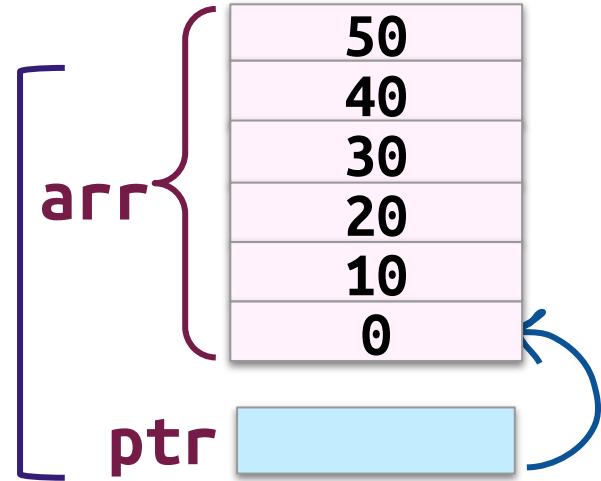
(gdb) p &arr[3] - &arr[1]

2 # of elems between arr[1] and arr[3] (not # of bytes, as our pointer arithmetic is in terms of ints)

(gdb) p sizeof(arr) **24** (6 ints * 4 bytes / int)

(gdb) p arr = arr + 1 **Invalid cast** (arr is array, not a pointer)

main

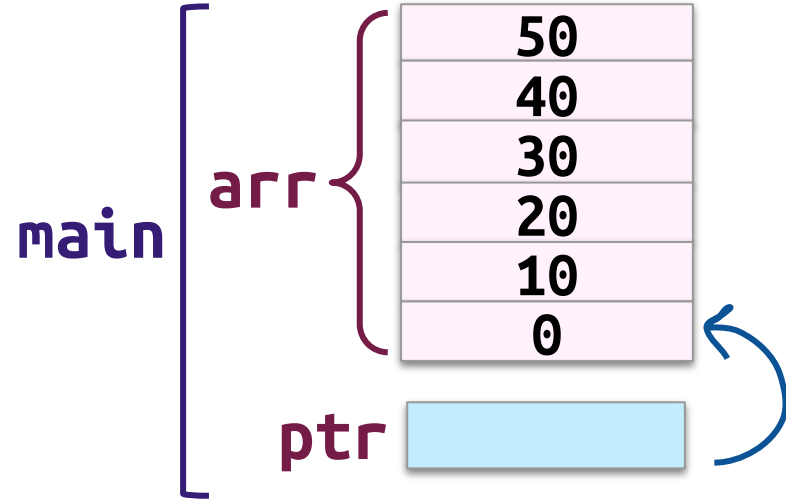




Check-Off Question

Why isn't `&arr` the same address as `&ptr`?
(even though `&arr[0]` and `&ptr[0]` are
the same address)

```
int arr[] = {0,10,20,30,40,50};  
int *ptr = arr;
```





Check-Off Question

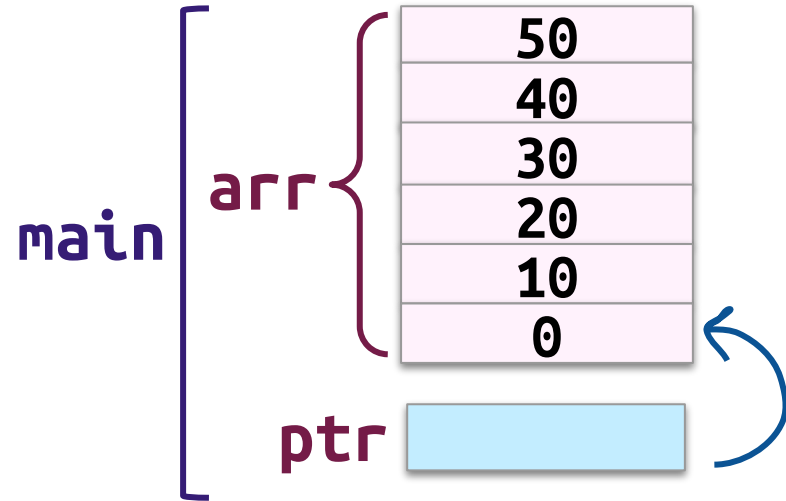
Why isn't `&arr` the same address as `&ptr`?
(even though `&arr[0]` and `&ptr[0]` are
the same address)

```
int arr[] = {0,10,20,30,40,50};  
int *ptr = arr;
```



An array is subtly different from a pointer.

When we ask where an array is located, we
typically mean **where the elements begin in
memory**. As such, `&arr` will be the **address of
the base element** of the array (i.e. `&arr[0]`).





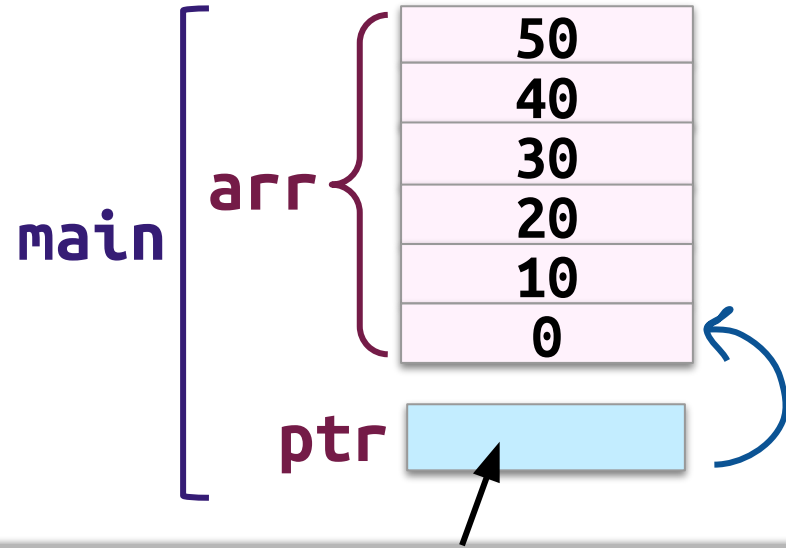
Check-Off Question

Why isn't `&arr` the same address as `&ptr`?
(even though `&arr[0]` and `&ptr[0]` are the same address)

```
int arr[] = {0,10,20,30,40,50};  
int *ptr = arr;
```

An array is subtly different from a pointer.

When we ask where an array is located, we typically mean **where the elements begin in memory**. As such, `&arr` will be the **address of the base element** of the array (i.e. `&arr[0]`).



However, `&ptr` refers to **where the pointer itself is being stored in memory** i.e. the address of the **blue box** above.

This is **not what array address it's pointing to** (the **pink box with 0**).



Array-To-Pointer Decay

When passed into functions, arrays decay into pointers to the starting element 🙌

```
int arr[] = {0,10,20,  
             30,40,50};
```

```
int *ptr = arr;  
binky(arr, ptr);
```

```
void binky(int a[], int *b) {  
    b[2] = 107;  
    a[2] *= 7;  
}
```

main

arr

50

40

30

20

10

0

ptr

binky

a

b

arr when passed into **binky** behaves like a pointer (**int ***), so inside **binky**, **a** (**arr**) and **b** (**ptr**) point to the same value



Q2: Valgrind: Heap Errors

In the `lab3` directory, peruse over (yet another lol) `buggy.c`, and also a `leak.c`, which may or may not have memory leaks, who knows `~\_(\ツ)\_/~`.

What exactly is going on? What does the memory magician Valgrind think?

Constellate into groups; move into the next exercise if you finish early. Please lmk if you have any questions!



~ 20 min + 10 min all-lab discussion



Error 1: Writing memory past allocated space

```
Copying string 'Stanford' (9 bytes total) to destination heap-allocated
8 bytes.
==3612603== Invalid write of size 1
==3612603== at 0x483F0BE: strcpy (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3612603== by 0x109283: make_error_1 (buggy.c:25)
==3612603== by 0x1093E3: main (buggy.c:64)
==3612603== Address 0x4a5f488 is 0 bytes after a block of size 8
alloc'd
==3612603== at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3612603== by 0x1091EA: make_error_1 (buggy.c:17)
==3612603== by 0x1093E3: main (buggy.c:64)
==3612603==
==3612603==
==3612603== HEAP SUMMARY:
==3612603== in use at exit: 0 bytes in 0 blocks
==3612603== total heap usage: 3 allocs, 3 frees, 1,040 bytes
allocated
==3612603==
==3612603== All heap blocks were freed -- no leaks are possible
==3612603==
==3612603== For lists of detected and suppressed errors, rerun with: -s
==3612603== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
0)
```

9 bytes (including \0!)

S	t	a	n	f	o	r	d	\0
---	---	---	---	---	---	---	---	----



Only 8 bytes of allocated space!

It's a memory issue to write
past this! (i.e. invalid write)



Error 2: Reading Memory Already Freed

```
==898626== Invalid read of size 1
==898626==    at 0x483EF46: strlen (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==898626==    by 0x48E5E94: __vfprintf_internal (vfprintf-
internal.c:1688)
==898626==    by 0x48CEEBE: printf (printf.c:33)
==898626==    by 0x1092CC: make_error_2 (buggy.c:34)
==898626==    by 0x1093B5: main (buggy.c:61)
==898626== Address 0x4a5f480 is 0 bytes inside a block of size 14
free'd
==898626==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==898626==    by 0x1092B4: make_error_2 (buggy.c:31)
==898626==    by 0x1093B5: main (buggy.c:61)
==898626== Block was alloc'd at
==898626==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==898626==    by 0x490C50E: strdup (strdup.c:42)
==898626==    by 0x10927E: make_error_2 (buggy.c:29)
==898626==    by 0x1093B5: main (buggy.c:61)
==898626==
```

Note that even though the error occurred inside `printf`, it's not the fault of **`printf`**'s implementation!

It's the fault of what was passed in (e.g., an already freed pointer)

Check-Off Q3: Similarly, for Valgrind errors showing a trace to `strcpy`, `strdup`, `strlen`, **it's more likely that we passed in invalid arguments / parameters** (i.e. a non-null terminated string).

```
33 printf("Freed memory contains
'%s'\n", ptr); //but ptr was freed!
```



Leaks: Memory Not Freed Properly



```
==2796932== Command: ./leaks 1
==2796932==
==2796932==
==2796932== HEAP SUMMARY:
==2796932==      in use at exit: 8 bytes in 1 blocks
==2796932==    total heap usage: 1 allocs, 0 frees, 8 bytes allocated
==2796932==
==2796932== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2796932==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==2796932==    by 0x10918A: make_leak_1 (leaks.c:14)
==2796932==    by 0x109217: main (leaks.c:32)
==2796932==
==2796932== LEAK SUMMARY:
==2796932==    definitely lost: 8 bytes in 1 blocks
==2796932==    indirectly lost: 0 bytes in 0 blocks
==2796932==    possibly lost: 0 bytes in 0 blocks
==2796932==    still reachable: 0 bytes in 0 blocks
==2796932==    suppressed: 0 bytes in 0 blocks
==2796932==
==2796932== For lists of detected and suppressed errors, rerun with: -s
==2796932== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```
void make_leak_1() {
    void *memory = malloc(8);
    ...
    // memory was never freed!
}
```

Check-Off Q3 (More):

Furthermore, for leaks, even though it might appear that something went wrong with `malloc()`, it's more likely that we forgot to free memory allocated to us by `malloc`!



Q3: Debugging!

In the newly cloned **lab3** directory, peruse over **piglatin** (not a real language lol), and try to get it to **crash / segmentation fault!**

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



~ 20 min + 10 min all-lab discussion



tree → eetray

pointer → ointerpay

yellow → elloway

before 1st vowel

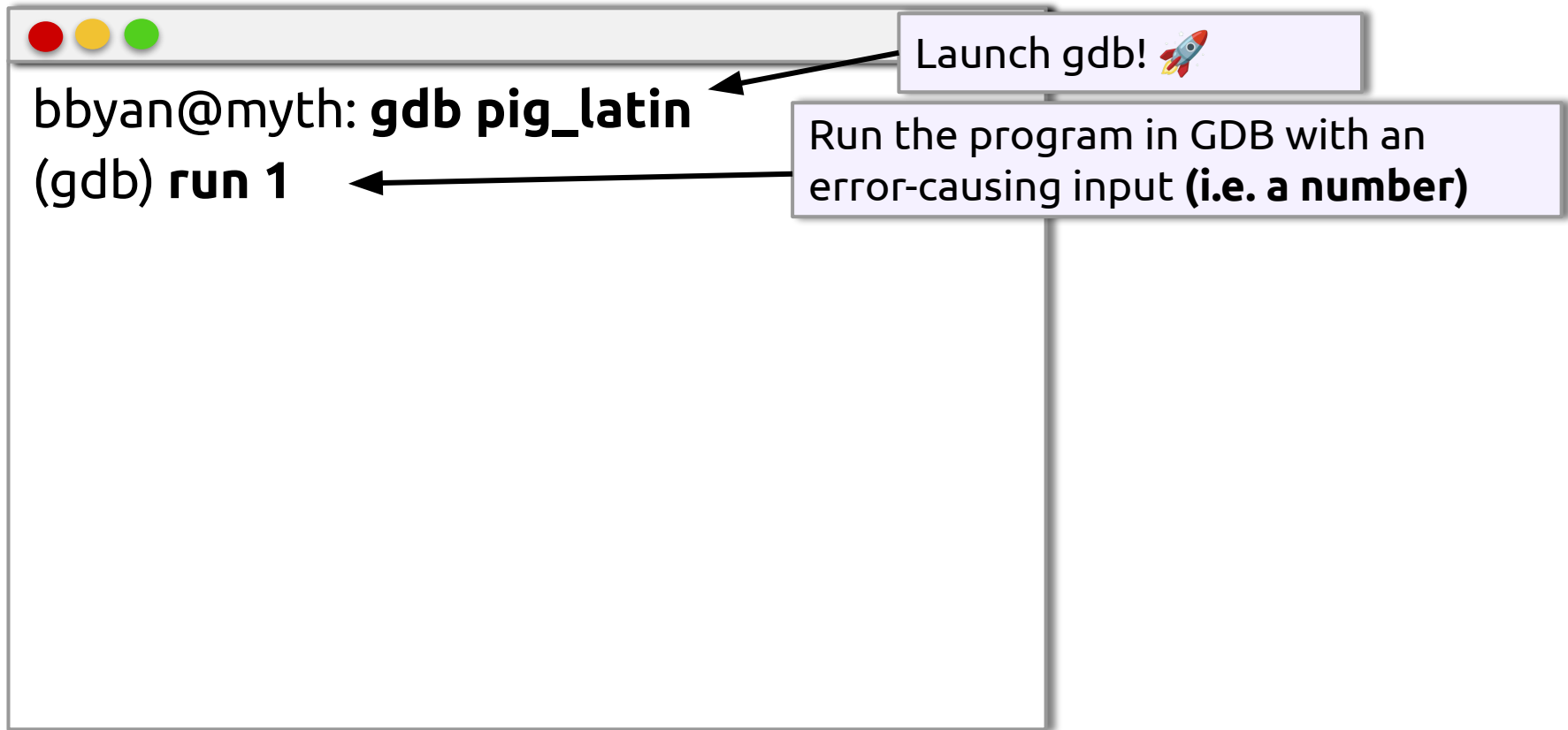
1st vowel & after

Check-Off Q: Who's Segfault? A GDB Whodunit



Launch gdb!  And don't forget to make first!

Check-Off Q: Who's Segfault? A GDB Whodunit



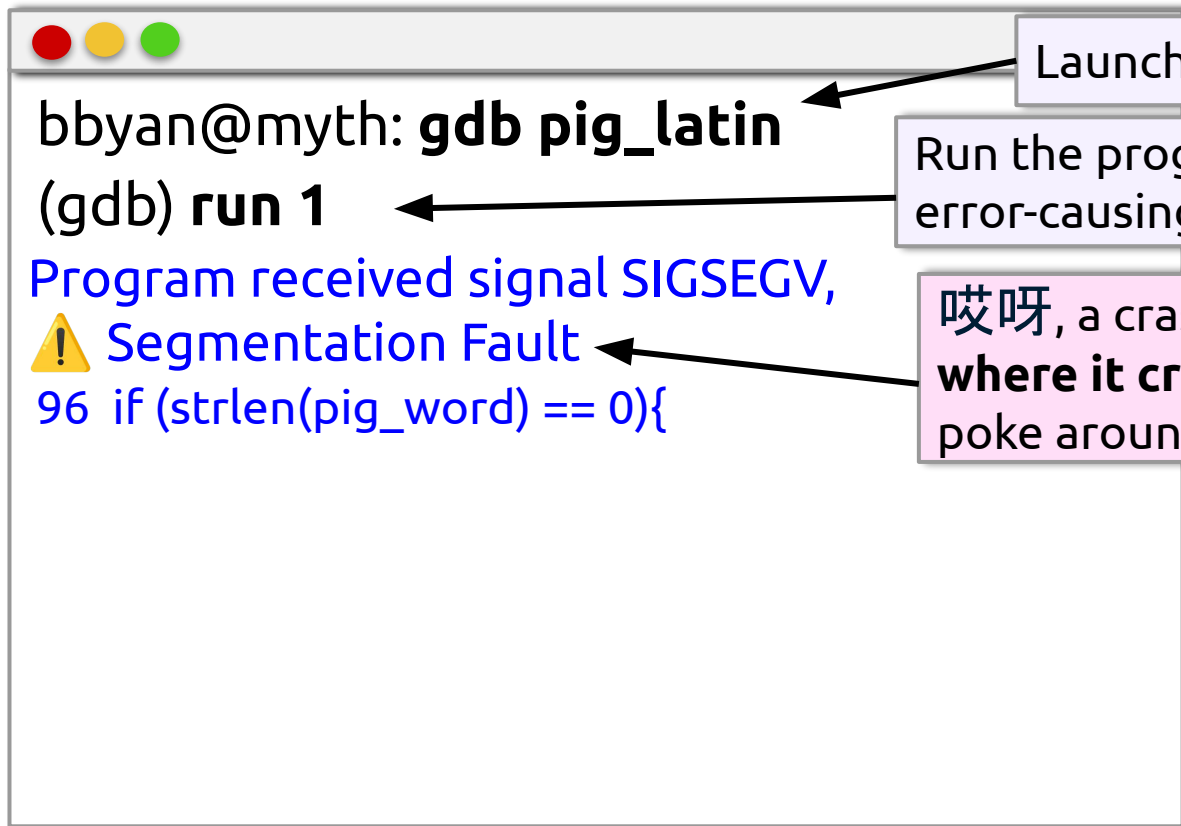
A terminal window with a title bar containing three colored buttons (red, yellow, green). The terminal displays the following text:

```
bbyan@myth: gdb pig_latin  
(gdb) run 1
```

Two callout boxes with arrows pointing to the terminal text:

- A box pointing to `gdb` contains the text: "Launch gdb! 
- A box pointing to `run 1` contains the text: "Run the program in GDB with an error-causing input (**i.e. a number**)"

Check-Off Q: Who's Segfault? A GDB Whodunit



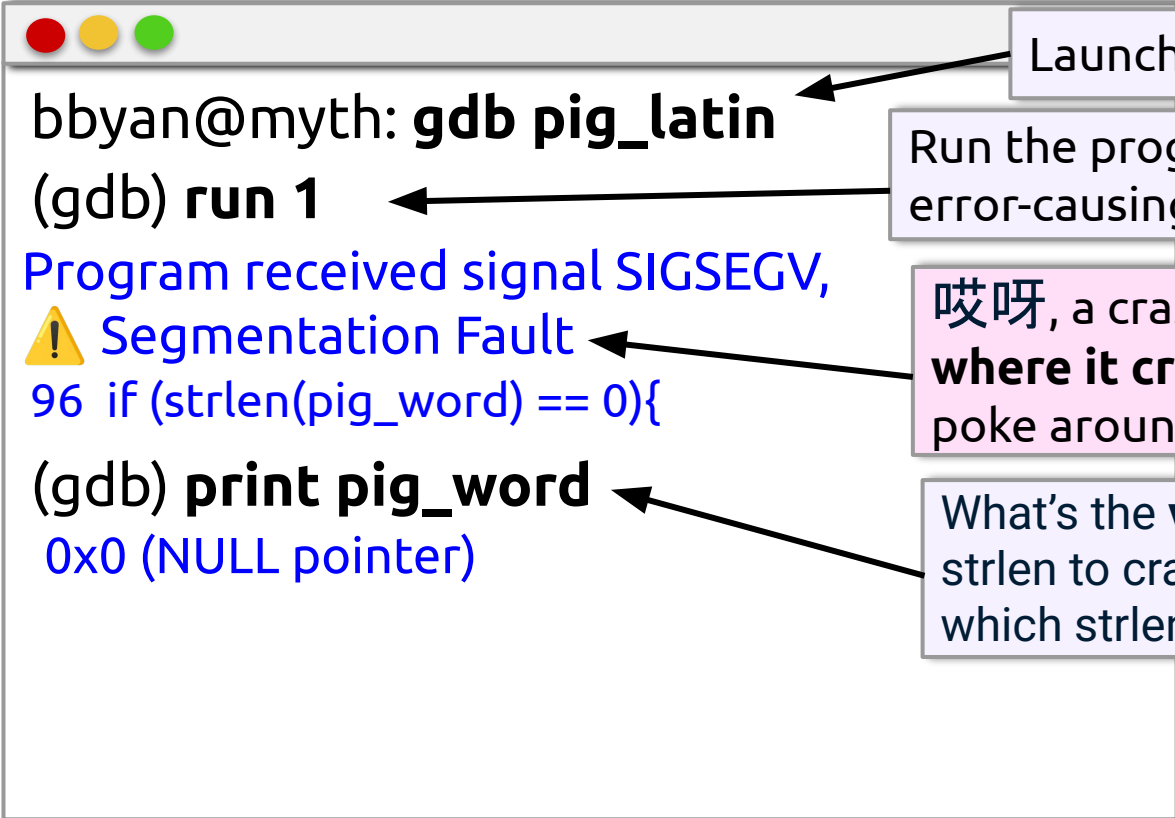
Launch gdb! 

Run the program in GDB with an error-causing input (**i.e. a number**)

哎呀, a crash! But **GDB leaves us where it crashed**, allowing us to poke around & see what happened.

```
bbyan@myth: gdb pig_latin
(gdb) run 1
Program received signal SIGSEGV,
! Segmentation Fault
96 if (strlen(pig_word) == 0){
```

Check-Off Q: Who's Segfault? A GDB Whodunit



Launch gdb! 

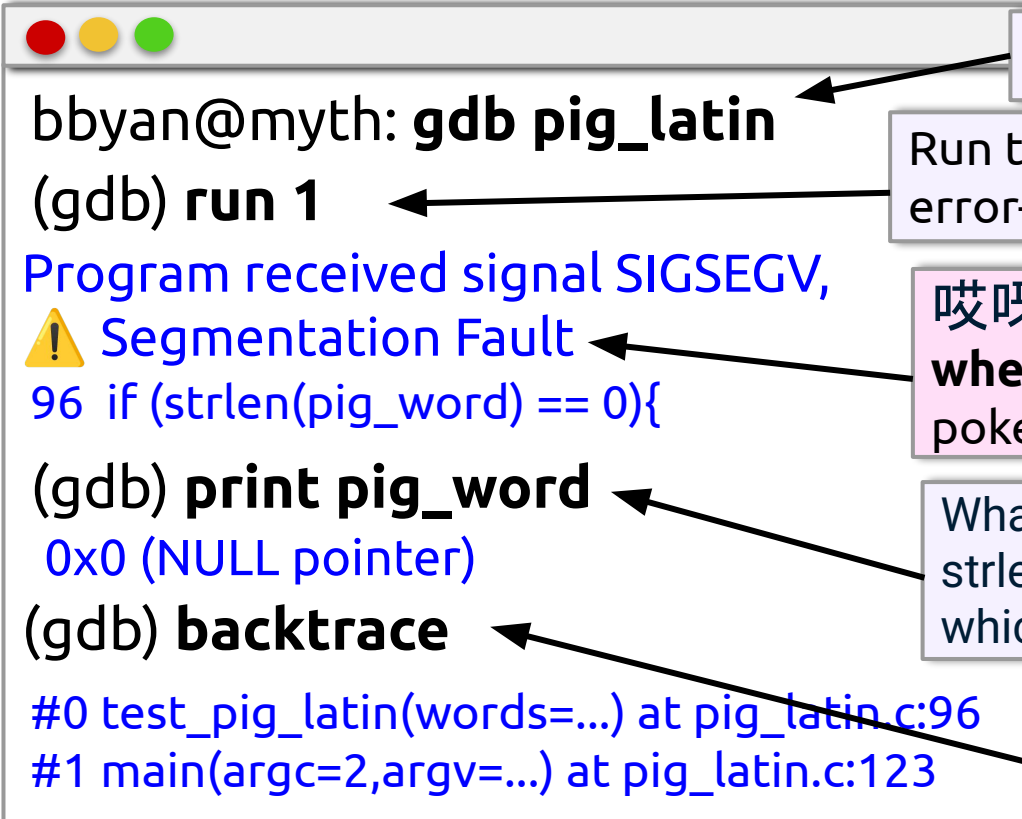
Run the program in GDB with an error-causing input (**i.e. a number**)

哎呀, a crash! But **GDB leaves us where it crashed**, allowing us to poke around & see what happened.

What's the **value of pig_word** causing strlen to crash? It's a **NULL pointer**, which strlen can't read!

```
bbyan@myth: gdb pig_latin
(gdb) run 1
Program received signal SIGSEGV,
! Segmentation Fault
96 if (strlen(pig_word) == 0){
(gdb) print pig_word
0x0 (NULL pointer)
```

Check-Off Q: Who's Segfault? A GDB Whodunit



The image shows a GDB terminal window with several annotations. The terminal text is as follows:

```
bbyan@myth: gdb pig_latin
(gdb) run 1
Program received signal SIGSEGV,
! Segmentation Fault
96 if (strlen(pig_word) == 0){
(gdb) print pig_word
0x0 (NULL pointer)
(gdb) backtrace
#0 test_pig_latin(words=...) at pig_latin.c:96
#1 main(argc=2,argv=...) at pig_latin.c:123
```

Annotations (callouts) with arrows pointing to the terminal:

- Launch gdb!**  (points to the terminal title bar)
- Run the program in GDB with an error-causing input (**i.e. a number**) (points to `run 1`)
- 哎呀, a crash! But **GDB leaves us where it crashed**, allowing us to poke around & see what happened. (points to the `Segmentation Fault` message)
- What's the **value of pig_word** causing `strlen` to crash? It's a **NULL pointer**, which `strlen` can't read! (points to `0x0 (NULL pointer)`)
- Also, where are we?** Looks like we're in **test_pig_latin** trying to convert our input (**1**) to pig latin. (points to the `backtrace` output)

Check-Off Q: Who's Segfault? A GDB Whodunit

Ok, how did `pig_word` become **NULL**? Let's look around more

Check-Off Q: Who's Segfault? A GDB Whodunit

Ok, how did `pig_word` become **NULL**? Let's look around more

bbyan@myth: **list**

```
95 char *pig_word = pig_latin(words[i]);  
96 if (strlen(pig_word) == 0){  
97     printf("unable to translate ...");
```

Check-Off Q: Who's Segfault? A GDB Whodunit

Ok, how did `pig_word` become **NULL**? Let's look around more

bbyan@myth: **list**

```
95 char *pig_word = pig_latin(words[i]);  
96 if (strlen(pig_word) == 0){  
97     printf("unable to translate ...");
```

It got returned by a helper `pig_latin()` – but why? Maybe to signal *unable to translate* ❌?

Check-Off Q: Who's Segfault? A GDB Whodunit



```
bbyan@myth: list
95 char *pig_word = pig_latin(words[i]);
96 if (strlen(pig_word) == 0){
97     printf("unable to translate ...");
bbyan@myth: break pig_latin
```

Ok, how did **pig_word** become **NULL**? Let's look around more

It got returned by a helper **pig_latin()** – but why? Maybe to signal *unable to translate* ❌?

Let's rewind and investigate **pig_latin()**!
We place a breakpoint there and rerun.

Check-Off Q: Who's Segfault? A GDB Whodunit

bbyan@myth: **list**

```
95 char *pig_word = pig_latin(words[i]);  
96 if (strlen(pig_word) == 0){  
97     printf("unable to translate ...");
```

bbyan@myth: **break pig_latin**

bbyan@myth: **run 1**

● Breakpoint 1, pig_latin(in="1"), at pig_latin.c: 61

```
// if word ... we can't translate ⚠  
61 if (strcspn(in, ALPHABET) > 0) {
```

Ok, how did **pig_word** become **NULL**? Let's look around more

It got returned by a helper **pig_latin()** – but why? Maybe to signal *unable to translate* ❌?

Let's rewind and investigate **pig_latin()**!
We place a breakpoint there and rerun.

Check-Off Q: Who's Segfault? A GDB Whodunit

Ok, how did `pig_word` become **NULL**? Let's look around more

```
bbyan@myth: list
95 char *pig_word = pig_latin(words[i]);
96 if (strlen(pig_word) == 0){
97     printf("unable to translate ...");
```

It got returned by a helper `pig_latin()` – but why? Maybe to signal *unable to translate* ❌?

```
bbyan@myth: break pig_latin
```

Let's rewind and investigate `pig_latin()`!
We place a breakpoint there and rerun.

```
bbyan@myth: run 1
```

● Breakpoint 1, `pig_latin(in="1")`, at `pig_latin.c: 61`

```
// if word ... we can't translate ⚠️
61 if (strcspn(in, ALPHABET) > 0) {
```

It indeed returns **NULL** for non-alphabetic input – so `test_pig_latin()`, the function we return to, **should check for NULL ptr.**

```
➡ bbyan@myth: next
62     return NULL;
```

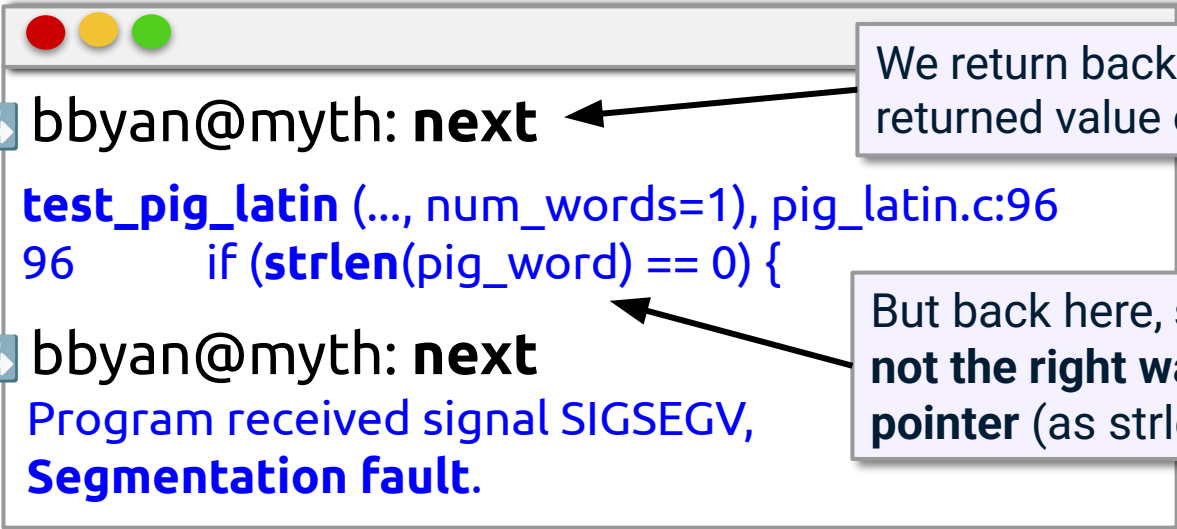
Check-Off Q: Who's Segfault? A GDB Whodunit

 bbyan@myth: **next**

We return back to **test_pig_latin**, with a returned value of **pig_word = NULL**;

```
test_pig_latin (... , num_words=1), pig_latin.c:96
96      if (strlen(pig_word) == 0) {
```

Check-Off Q: Who's Segfault? A GDB Whodunit



```
bbyan@myth: next
test_pig_latin (... , num_words=1), pig_latin.c:96
96      if (strlen(pig_word) == 0) {
bbyan@myth: next
Program received signal SIGSEGV,
Segmentation fault.
```

We return back to `test_pig_latin`, with a returned value of `pig_word = NULL`;

But back here, `strlen(pig_word) == 0` is **not the right way to check for a NULL pointer** (as `strlen` crashes ⚠).



So what is the right way to check if a pointer is NULL?

Check-Off Q: Who's Segfault? A GDB Whodunit

↳ bbyan@myth: **next**

```
test_pig_latin (... , num_words=1), pig_latin.c:96  
96      if (strlen(pig_word) == 0) {
```

We return back to **test_pig_latin**, with a returned value of **pig_word = NULL**;

But back here, **strlen(pig_word) == 0** is **not the right way to check for a NULL pointer** (as **strlen** crashes ⚠).

So instead, on line 96, we can do
if (pig_word == NULL){ ... }
which is our fix! ✓





Helpful Debugging Guide

The screenshot shows the CS107 website interface. At the top, a dark red navigation bar contains the text "CS107" and a series of links: "Calendar", "Handouts", "Lectures", "Assignments", "Labs", "Gradebook", and "Getting Help". The "Handouts" link is active, and a white dropdown menu is displayed over the main content area. This menu lists various resources: "Syllabus", "SCPD Information", "Honor Code", "Course Enrollment FAQ", "Course Reader", "Getting Started on Myth", "Unix Guide", "Emacs Guide", "Vim Guide", "The GDB Debugger", "Valgrind Guide", "C Standard Library Guide", "Debugging Guide" (which is highlighted with a red rectangular border), "x86-64 Guide", and "Callgrind Guide". In the background, the main content area features a large heading "Debugging" in red, followed by a paragraph about developing skills as a computer scientist. Below this, there is a section titled "Debugging" with a sub-heading "The classic..." and a paragraph discussing debugging techniques and tools like GDB and Valgrind.

<https://web.stanford.edu/class/archive/cs/cs107/cs107.1254/resources/debugging.html>



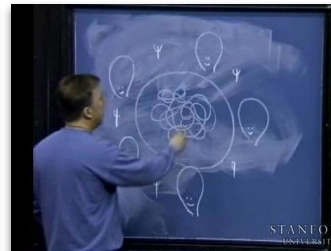
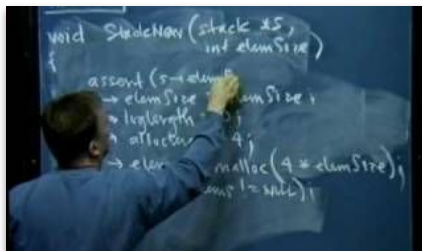
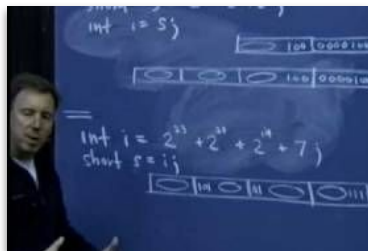
Once again, today's main takeaway :)

*To have the changes to a value
persist outside of a function,
we must
pass a pointer to the value!
Don't forget :)*

Text: To have the changes to a value persist outside of a function,
we must pass a pointer to the value! Don't forget :)



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, email or over Slack
- I always appreciate and value any feedback regarding labs, materials, teaching, office hours, and so on!



“Always two there are, no more, no less. A pointer and a pointee.”
– Yoda said this I’m pretty sure