Lecture 05
Lifetimes and Memory

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Spring 2023

Slides adapted from:
Jesse Tov, Vincent St-Amour

Northwestern

Administrivia

- Homework 1 due tonight
  - Homework 2 will also release tonight, with a one-week deadline
  - Lots of office hours today to help with questions
  - Slip days allow you to submit late without penalty
    - Use them when you need them

Today's Goals

- Continue examples of Strings, Arrays, and Pointers
  - Explain AddressSanitizer errors you'll get when working with them

- Discuss variable lifetimes: when is a variable no longer valid

- Understand memory and C memory layout
  - The basis for pointers and variable lifetimes

Getting the code for today

`cd ~/cs211/lec/` (or wherever you put stuff)

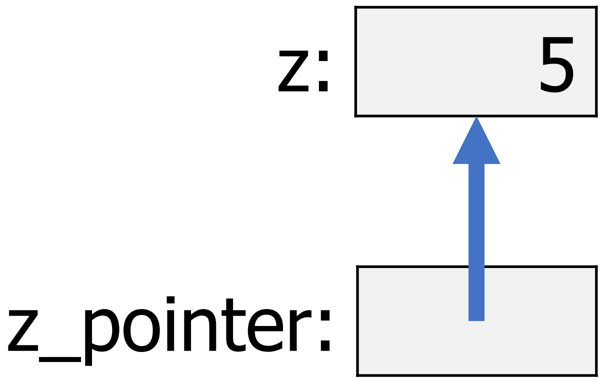
```
tar -xkvf ~cs211/lec/05_lifetimes_memory.tgz
cd 05_lifetimes_memory/
```

Outline

- **Pointers**
- Address Sanitizer
- Arguments to main()
- Variable Lifetimes
- Memory Layout

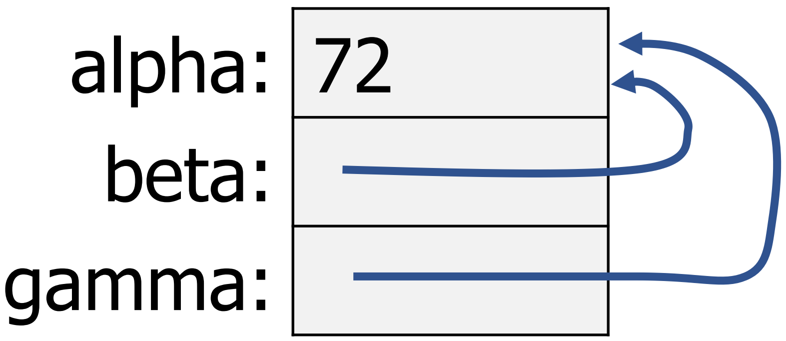
Pointers are another type of value

- Values could be a number, like 5 or 6.27
- Or they could be a “pointer” to an **object**
  - Points at the object, not the variable or value
  - It points at the “chunk of memory”
    - Technically, in C it holds the address of that memory



Pointer examples

```
double alpha = 72;
double* beta = &alpha;
double* gamma = beta;
```



- Pointers have a “value” that is some memory address
  - Contains the “location” of some object in memory
  - Conceptually an arrow pointing at that object

- Operator `&` gets the memory address of an object

Dereference a pointer to get the value it points at

```
void add_two(int* n) {
  *n = *n + 2;
}

int main(void) {
  int x = 15;
  add_two(&x);
  printf("%d\n", x);
  return 0;
}
```

- Operator `*` follows the pointer to interact with the value
  - Can be used to read or write
- End result: functions have the ability to directly modify variables through pointers

Possible pointer values

- Uninitialized

```
unsigned long* zeta;
```

- Pointing at an existing object

```
char* letter_ptr = &my_char;
```

- Null (explicitly pointing at nothing)

```
int* p = NULL;
bool* b = NULL;
double* d = NULL;
```

  - NULL works for any pointer type
  - NULL is NOT the same as uninitialized (🐝)
  - Dereferencing a null pointer is an error (segfault)

A note on writing meaningful code

- Technically, NULL pointers and null terminators are both implemented as a value zero (on any modern system)
  - `false` is implemented as zero as well
  - So, technically, you could use any to mean any

- But humans will be the ones reading your code
  - NULL '\0', 0, and `false` all have different meanings

  - NULL means pointers
  - '\0' means the end of strings
  - `false` means a Boolean value
  - 0 means a number

Use the one that is appropriate to the situation!

Arrays passed into functions are just pointers

array_print.c

- When you pass an array into a function, you don't pass a copy of the values
  - Instead you pass a pointer to the start of the array
  - Be sure to pass a length as well! (no way to determine that in C)

```
void print_array(int* values, int count) {
  . . .
}

int main(void) {
  int array[10] = {1, 2, 3, 4, 5, 5, 4, 3, 2, 1};
  print_array(array, 10);
  return 0;
}
```

Outline

- Pointers
- **Address Sanitizer**
- Arguments to main()
- Variable Lifetimes
- Memory Layout

DANGER! Nothing stops you from going past the end of an array

array_print.c

- C does not check whether your array accesses are valid
    - It just tries to grab the value in the memory you asked for

- Going past the end (or before the beginning) of an array is **UNDEFINED BEHAVIOR**
    - Could result in *anything* happening

- If you’re lucky, the code will crash
    - But you will not always get lucky
    - Be sure to always check if you’re going past the end of the array

Address Sanitizer

- Automatically compiled in as part of your homework code

- Checks various accesses to memory for validity
  - Produces long error messages that can be scary at first! But are really helpful!

  - Error locations: (more on these “locations” on Thursday)
    - Stack – local variable
    - Global – global variable (usually a string)
    - Heap – variable created with `malloc()`

  - Error types:
    - buffer-overflow – past the end of an array of memory
    - buffer-underflow – before the beginning of an array of memory (rare)
    - various others

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn’t fit on the slide)
```

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn’t fit on the slide)
```

Error is coming from AddressSanitizer

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn't fit on the slide)
```

Heap-buffer-overflow means past the end of an array created with `malloc()`

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn't fit on the slide)
```

The error happened in `expand_charseq()` in src/translate.c line 74

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn’t fit on the slide)
```

Full “stack trace” of functions that were called to get to where the error happened

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn't fit on the slide)
```

Where the array was created in the first place (`expand_charseq()` in translate.c line 62)

Live demos of AddressSanitizer

array_print.c
string_print.c

- With pointers, arrays, and strings

- Things to try
  - Intentionally go past end of array
  - Go before beginning of array
  - Use a pointer as an array
  - NULL pointer

  - Malformed string with printf

Where the error happened may not but where the bug is

- AddressSanitizer usually points to a line where the array is being accessed
- But the bug is often because an index is out of bounds
- Or because the pointer passed in was invalid to begin with
- This is a new class of problem you'll all have to deal with
  - Errors that occur because of bugs elsewhere

Other AddressSanitizer errors

string_print.c

- Dereferencing a NULL pointer

```
src/string_print.c:4:28: runtime error: load of null pointer of type 'const char'
AddressSanitizer:DEADLYSIGNAL
=================================================================
==2838978==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x000000400912 bp
0x000000000000 sp 0x7ffe1379cec0 T0)
==2838978==The signal is caused by a READ memory access.
==2838978==Hint: address points to the zero page.
SCARINESS: 10 (null-deref)
    #0 0x400911 in print_string_chars src/string_print.c:4
    #1 0x400a33 in main src/string_print.c:12
    #2 0x7fefdbf5a492 in __libc_start_main ../csu/libc-start.c:314
    #3 0x40082d in _start (/home/branden/cs211/f21/lec/04_arrays_strings/string_print+0x40082d)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV src/string_print.c:4 in print_string_chars
==2838978==ABORTING
```

Break + Say hi to your neighbors

- Things to share
  - Name

  - Major

  - One of the following
    - Favorite Candy
    - Favorite Pokemon
    - Favorite Emoji

Break + Say hi to your neighbors

- Things to share
  - Name -Branden
  - Major -Electrical and Computer Engineering, and Computer Science
  - One of the following
    - Favorite Candy - Twix
    - Favorite Pokemon - Eevee
    - Favorite Emoji - 

Outline

- Pointers
- Address Sanitizer
- **Arguments to main()**
- Variable Lifetimes
- Memory Layout

Passing arguments to main

- We’ve been using “`int main(void);`” as `main()`’s signature

- Actually, `main()` can receive arguments, which are what the user called the program with

```
% ./programname arg1 arg2 arg3
```

Real signature for main

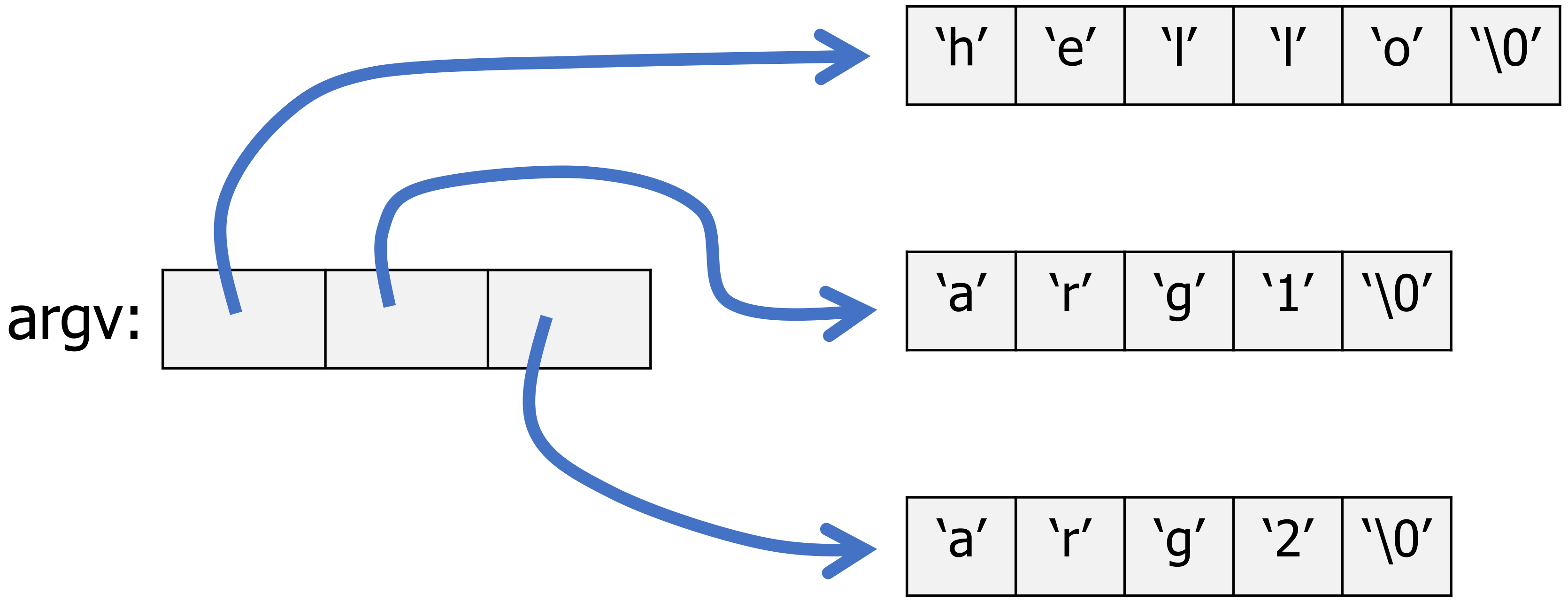
- The real signature for `main()` is:

```
int main(int argc, char* argv[]);
```

- `argc` – the number of strings in `argv` (length of `argv`)
- `argv` – an array of strings (array of char*)
  - The first string is the name of the program itself
  - The remaining strings are the arguments to the function
- By using `main(void)`, we've just been ignoring these
  - Which is fine, because they aren't always useful

Pointer to a pointer

Run program as: `./hello arg1 arg2`



`char* argv[]` – array of pointers

Working with argv

argv_print.c

- Let’s print out all the arguments to the function

```
int main(int argc, char* argv[]) {
  for (int i=0; i<argc; i++) {
    printf(“Argument %d: \”%s\”\n”, i, argv[i]);
  }

  return 0;
}
```

Outline

- Pointers
- Address Sanitizer
- Arguments to main()
- **Variable Lifetimes**
- Memory Layout

When is a pointer "valid"?

1. If it is initialized

2. If the variable it is referencing still has a valid lifetime
   - Variables "live" until the end of the scope they were created in
   - Scopes are defined by { }

   - Example:

```
void some_function(void) {
    int a = 5;
}
```

← `a` goes "out of scope" here
The variable stops being "alive"

Examples of variable lifetimes

```
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

a: 5

Examples of variable lifetimes

```
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

a: 5

Examples of variable lifetimes

```
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

a: 5

Examples of variable lifetimes

```
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

~~a~~: 

- Variable `a` is no longer "alive" at this point
  - It "poofs" out of existence
  - The variable is no longer valid

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```

n: 17

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```

n:	17
a:	5

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

n:	17
a:	5

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```

n:	17
a:	5
b:	16

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

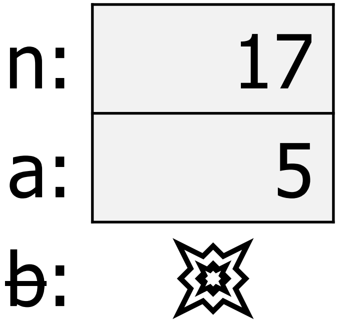
n:	17
a:	5
b:	16

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

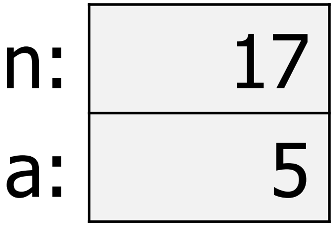


Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```



Referring to variable `b` at this point would be a compilation error

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

~~n~~:

~~a~~:

Variable lifetimes are what makes loops work

- Variables created inside of loops only exist until the end of that iteration of the loop
  - i.e. they only exist until the next end curly brace }

```
while (n < 5) {
    int i = 1;
    n += i;
}
```

A new variable `i` is created each time the loop repeats

Dangling pointers reference invalid objects

```
int* get_pointer_to_value(void) {
  int n = 5;
  return &n;
}

int main(void) {
  int* x = get_pointer_to_value();
  printf("%d\n", *x);
  return 0;
}
```

Dangling pointers reference invalid objects

dangling_pointer.c

```
int* get_pointer_to_value(void) {
  int n = 5;
  return &n;
}

int main(void) {
  int* x = get_pointer_to_value();
  printf("%d\n", *x);
  return 0;
}
```

`n` goes out of scope at the end of this function

So what does the pointer point to???

Dangling pointers are especially dangerous

- Accessing a dangling pointer is **UNDEFINED BEHAVIOR**
  - Anything could happen!
- If you are lucky: segmentation fault (a.k.a. SIGSEGV)
  - The OS kills your program because it accesses invalid memory
- If you are unlucky: *anything at all*
  - Including returning the correct result the first time you run it and an incorrect result the second time
- AddressSanitizer checks for this and will gift you a crash

String literals are an exception to scoping rules

string_lifetime.c

- String literals **always** exist
  - This is why they cannot be modified. They might be reused later

```
const char* get_pointer_to_string(void) {
  return “oh, hello!”; // this is okay for string literals
}

int main(void) {
  const char* string = get_pointer_to_string();
  printf(“%s on broadway\n”, string);
  return 0;
}
```

Break + Question

```
int* get_array_pointer(int* array, int length) {
    if (length > 2) {
        return &(array[2]);
    }
    return array;
}

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int* x = get_array_pointer(array, 5);
    printf("%d\n", *x);
    return 0;
}
```

Is it valid to return a pointer here?

Will this access fault?

Break + Question

```
int* get_array_pointer(int* array, int length) {
    if (length > 2) {
        return &(array[2]);
    }
    return array;
}

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int* x = get_array_pointer(array, 5);
    printf("%d\n", *x);
    return 0;
}
```

Is it valid to return a pointer here? **Yes**

This code works because the lifetime of the array is longer than the lifetime of the `get_array_pointer()` function.

Will this access fault? **No**

Outline

- Pointers
- Address Sanitizer
- Arguments to main()
- Variable Lifetimes
- **Memory Layout**

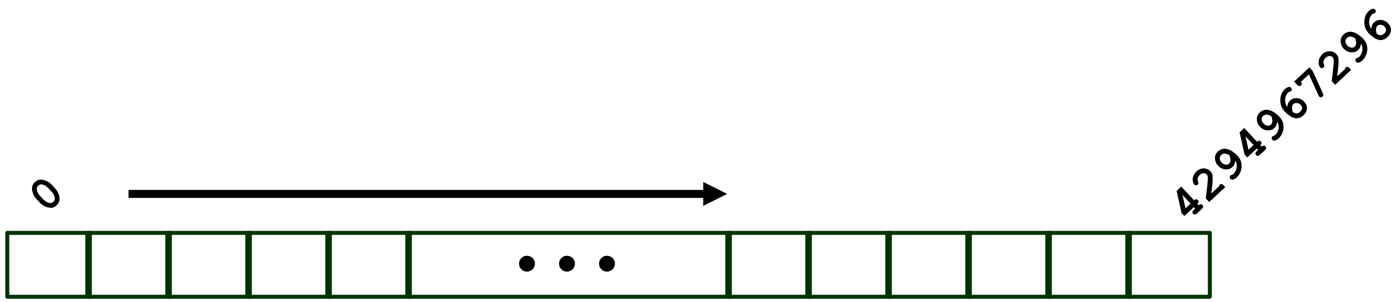
Memory

- Computers have memory
  - RAM sticks
  - Also some dedicated memory inside of the processor

- The operating system of the computer hands out chunks of memory to running processes
  - Like our compiled C programs

  - While they are running, they have a certain amount of memory reserved for their use
    - You can see this in Task Manager on Windows (or Top on Linux)

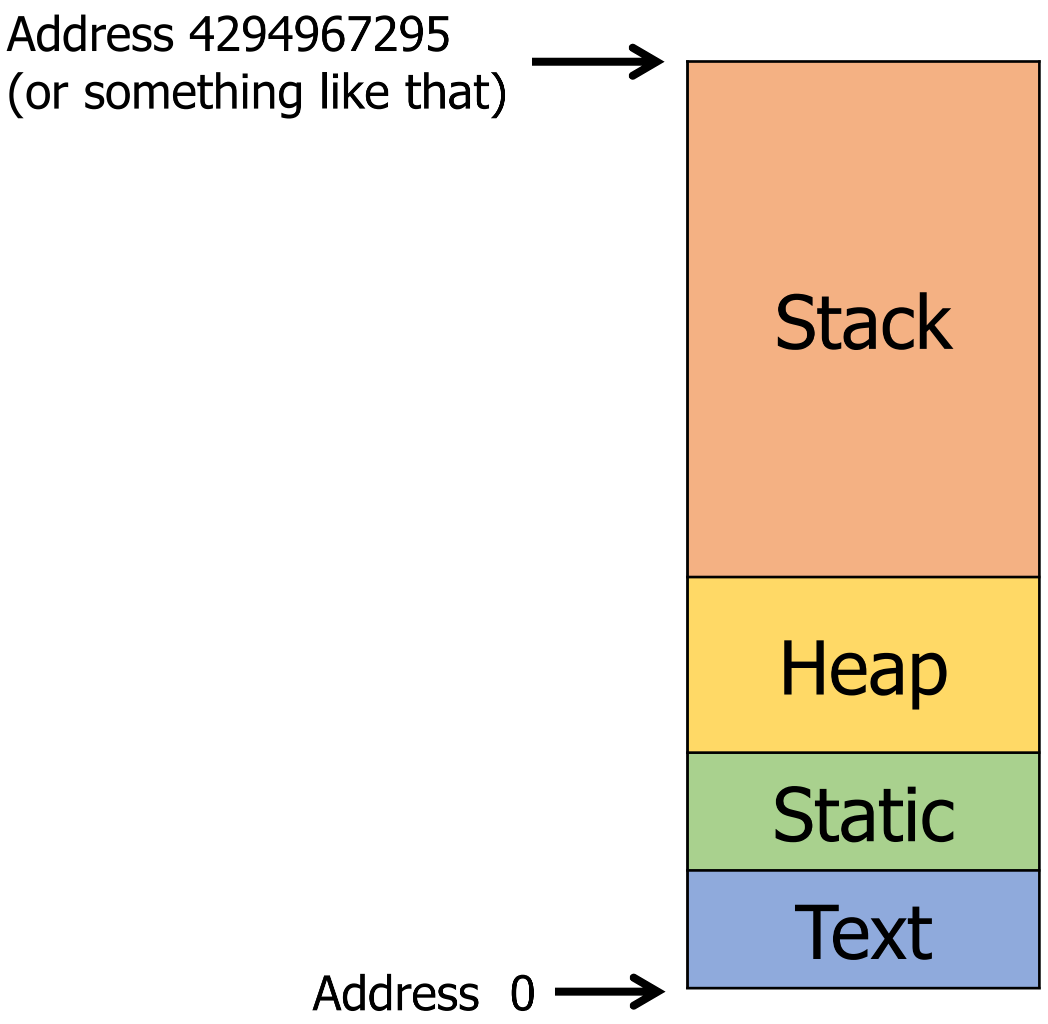
What is memory conceptually?



- A nearly infinite series of slots that can be used to hold data
  - Units of memory are known as bytes
  - So 4 GB of RAM is memory with 4294967296 bytes
    - Typical variables take 1-8 bytes
- Each slot in the memory has an index: a memory address
  - Pointers are the memory address of a variable

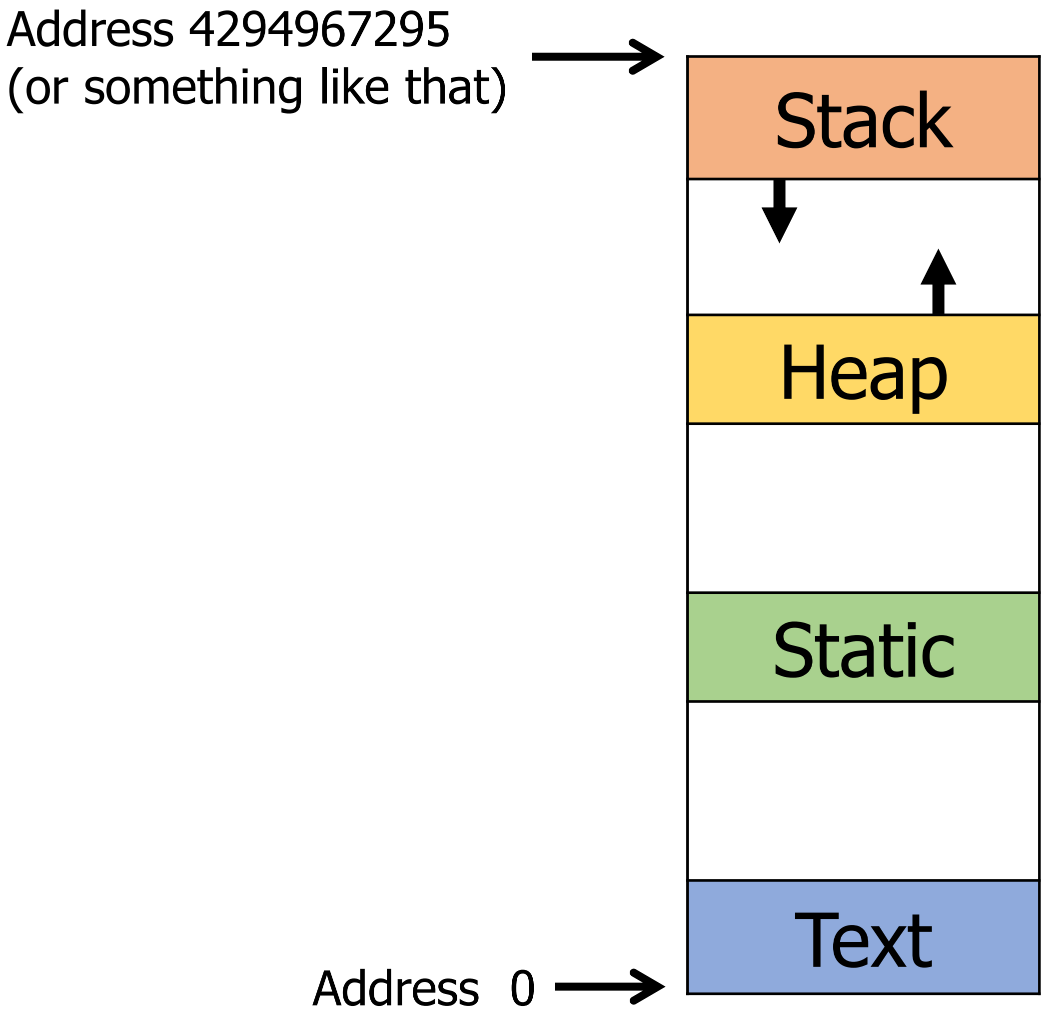
C memory layout

- Stack Section
  - Local variables
  - Function arguments
- Heap Section
  - Memory granted through `malloc()`
- Static Section (a.k.a. Data Section)
  - Global variables
  - Static function variables
  - Subsection with read-only data
    - Like string literals
- Text Section (a.k.a Code Section)
  - Program code



C memory layout

- Conceptually, the sections are laid out next to each other
- Realistically, there are huge gaps between them
  - Because most programs don't use all that much memory
- The stack/heap sections can grow in size if necessary

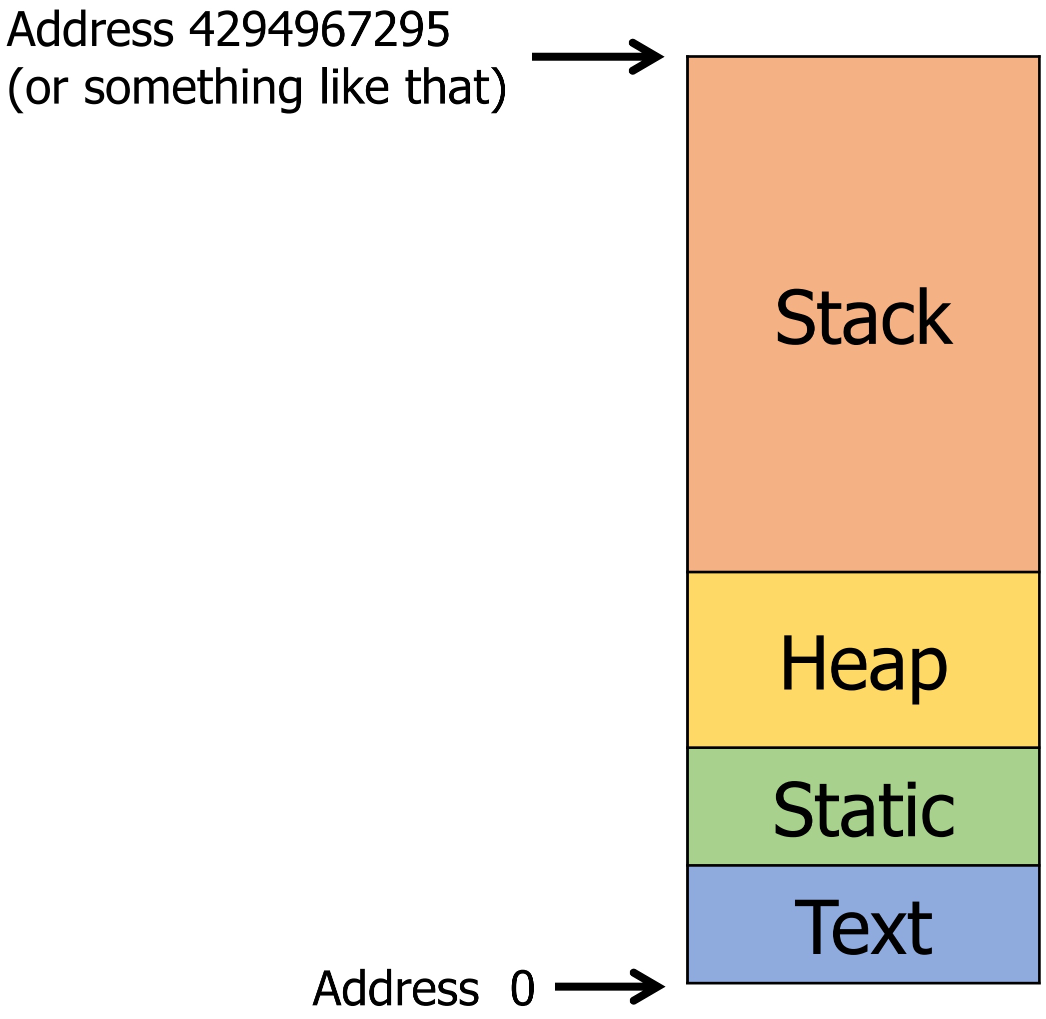


C memory layout

```
int a;
void foo(short b) {
     static int c = 3;

     char* d;
     d = (char*) malloc(4);

     printf("Hello CS211\n");
}
```

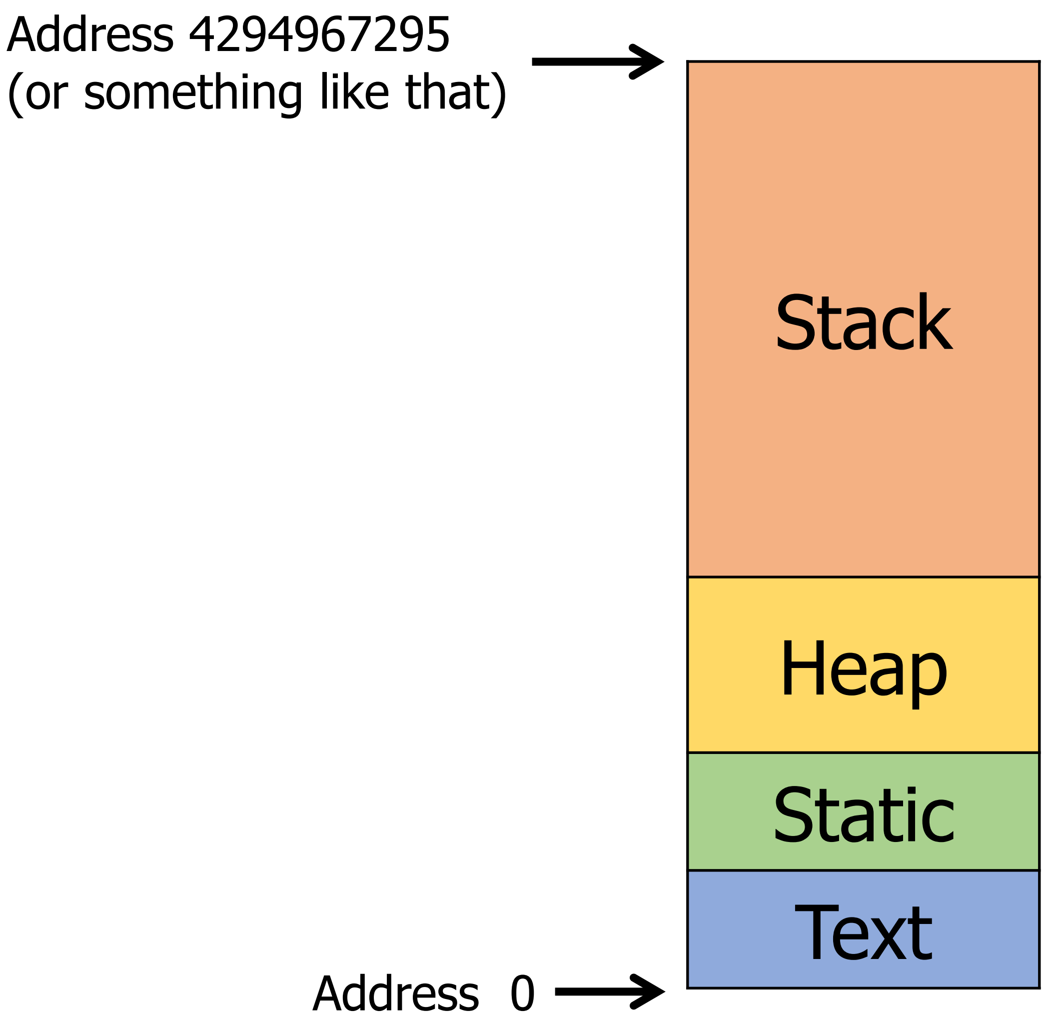


C memory layout

```
int a;
void foo(short b) {
      static int c = 3;

      char* d;
      d = (char*) malloc(4);

      printf("Hello CS211\n");
}
```

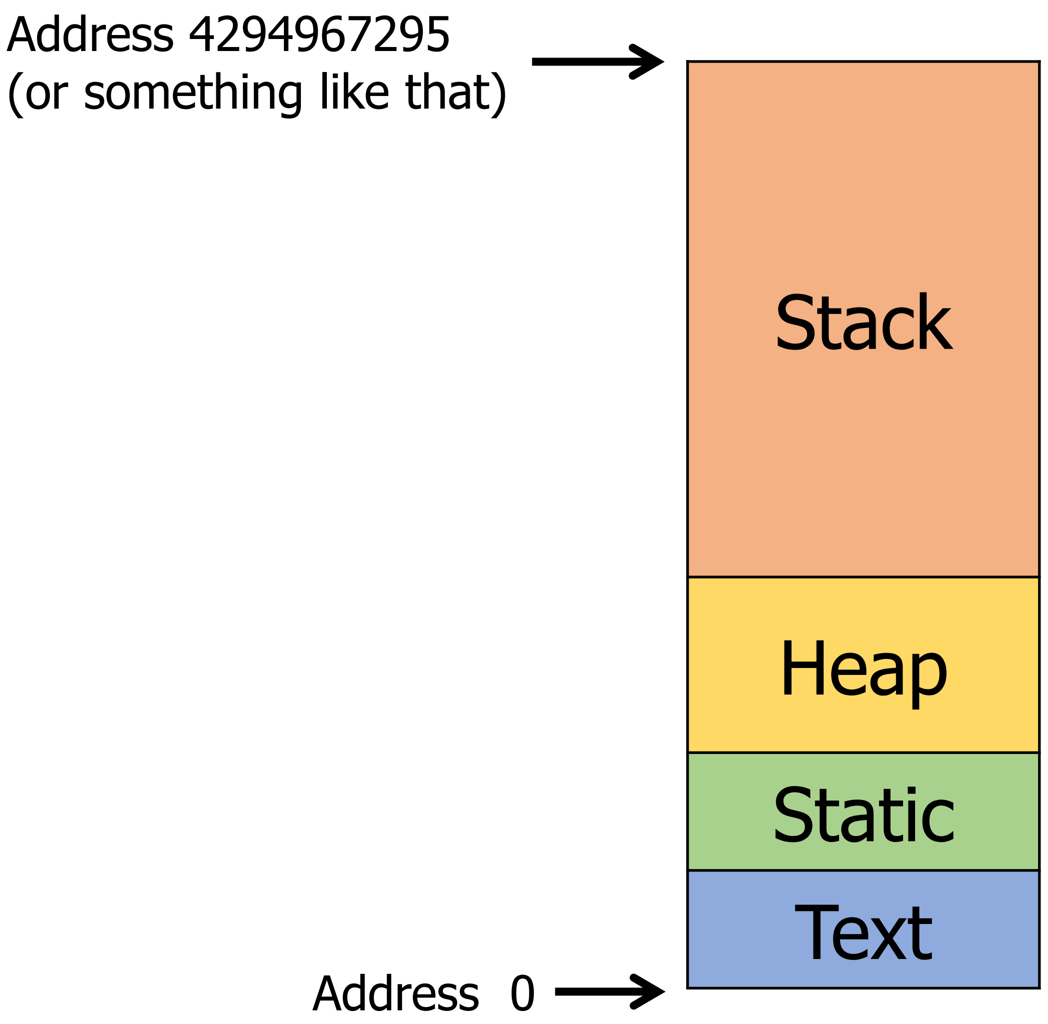


C memory layout

```
int a;
void foo(short b) {
     static int c = 3;

     char* d;
     d = (char*) malloc(4);

     printf("Hello CS211\n");
}
```

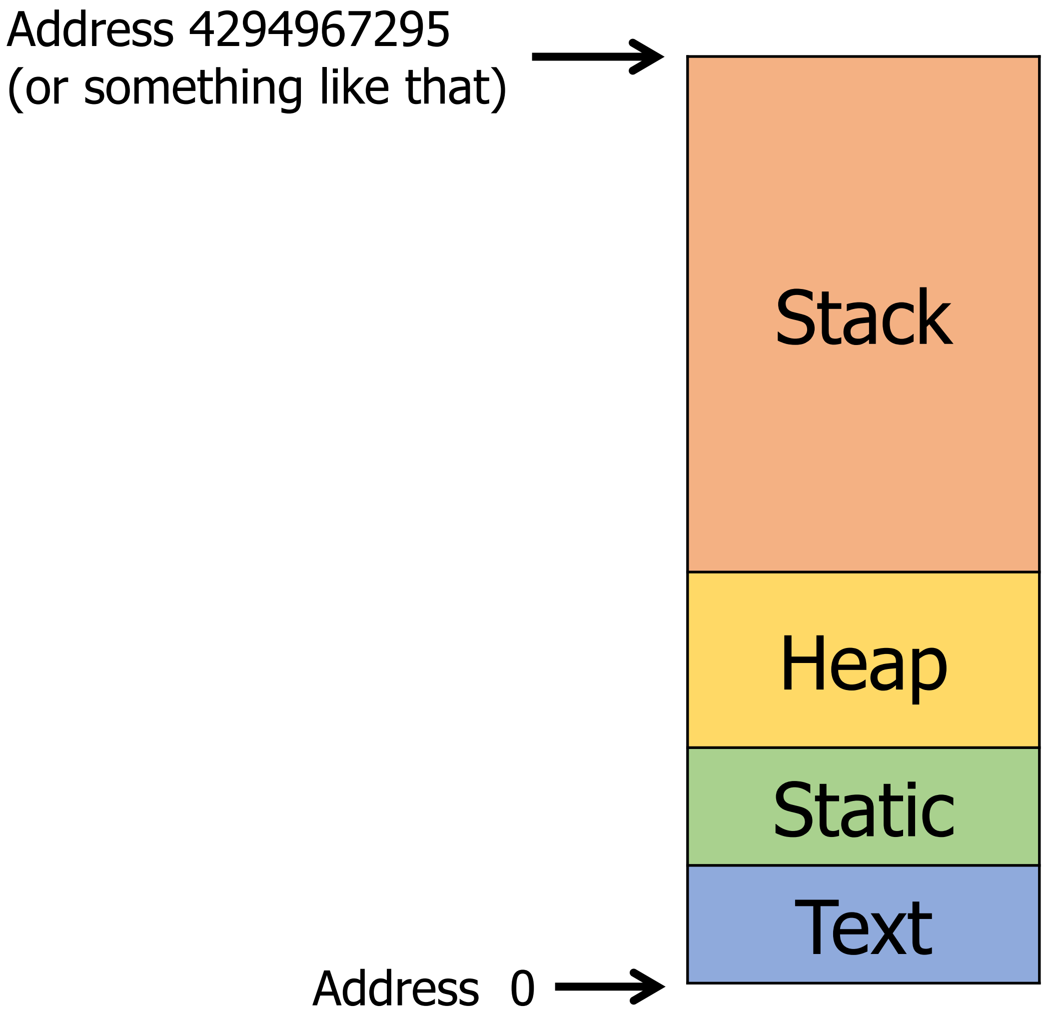


C memory layout

```
int a;
void foo(short b) {
     static int c = 3;

     char* d;
     d = (char*) malloc(4);

     printf("Hello CS211\n");
}
```

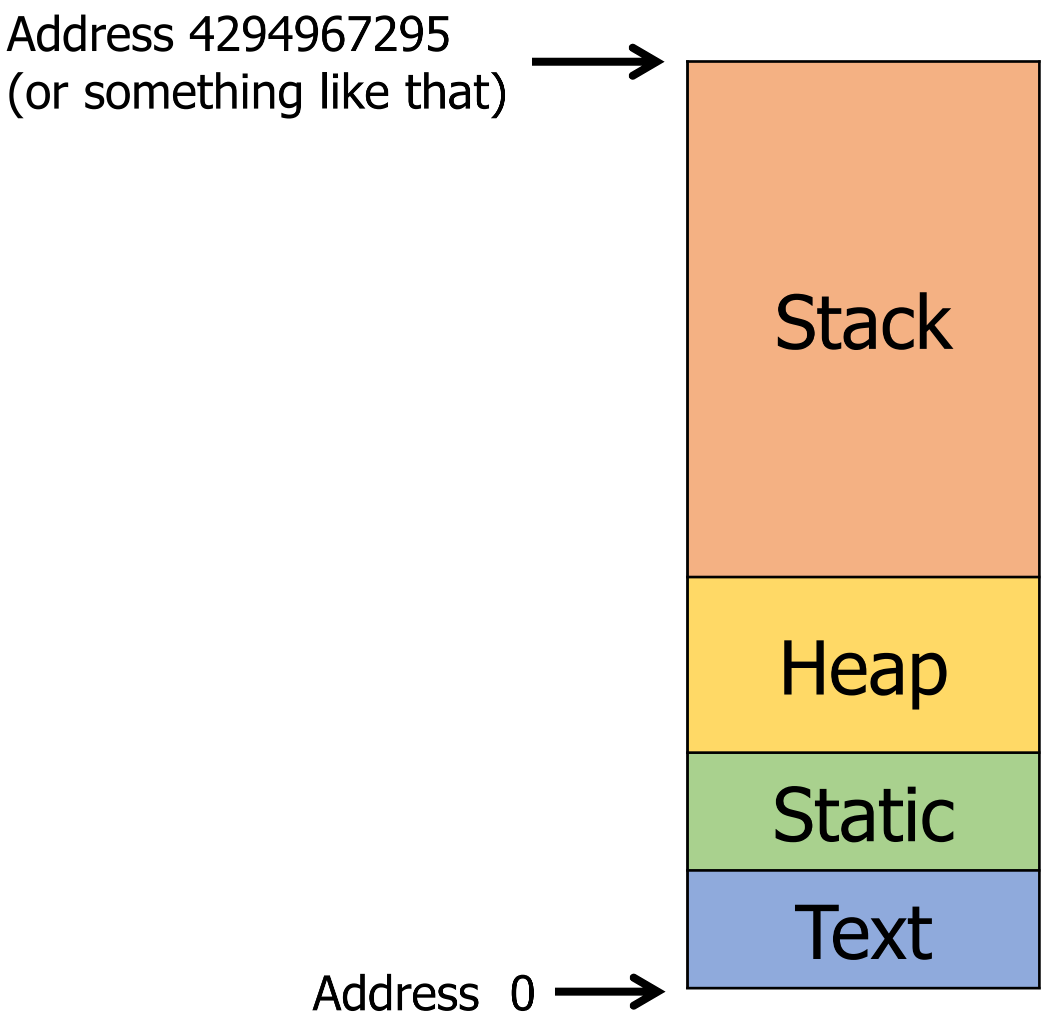


C memory layout

```
int a;
void foo(short b) {
     static int c = 3;

     char* d;
     d = (char*) malloc(4);

     printf(“Hello CS211\n”);
}
```

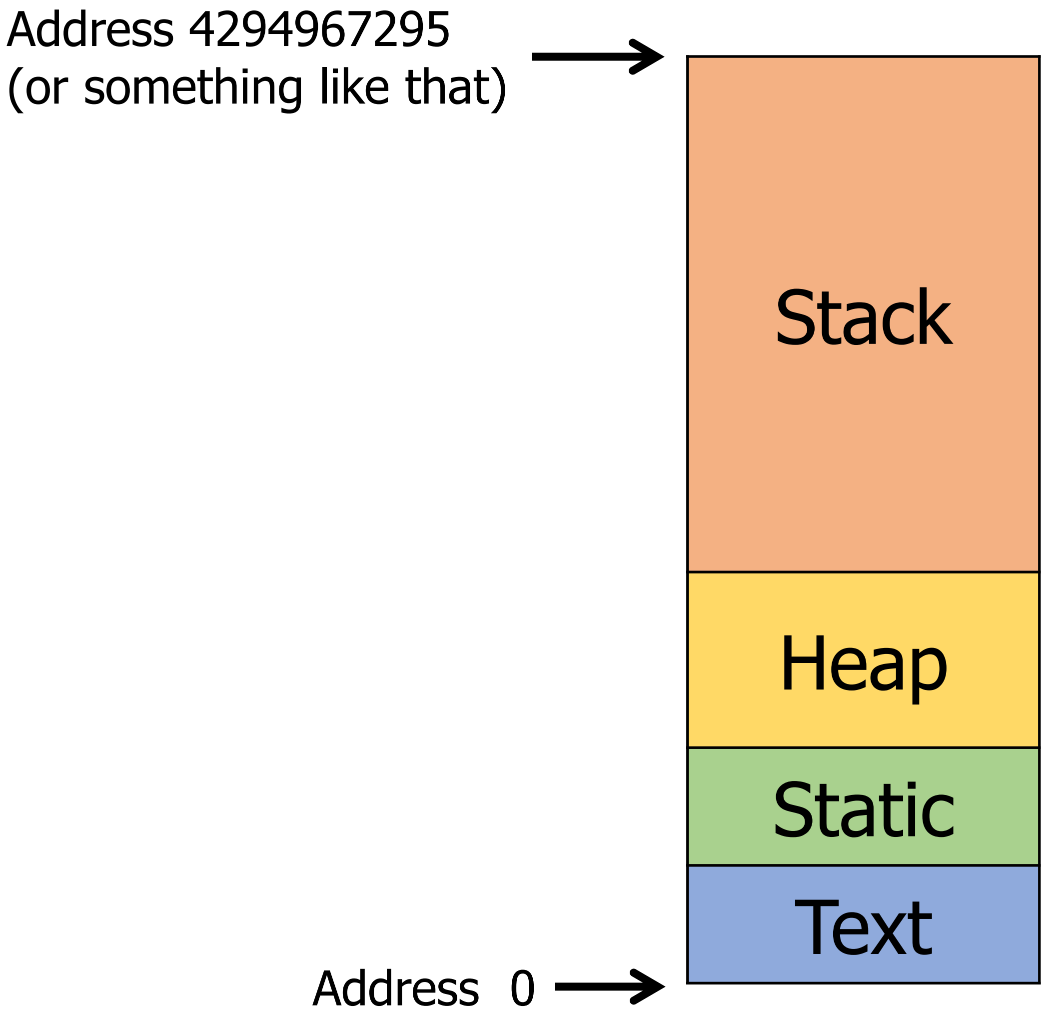


C memory layout

```
int a;
void foo(short b) {
    static int c = 3;

    char* d;
    d = (char*) malloc(4);

    printf("Hello CS211\n");
}
```

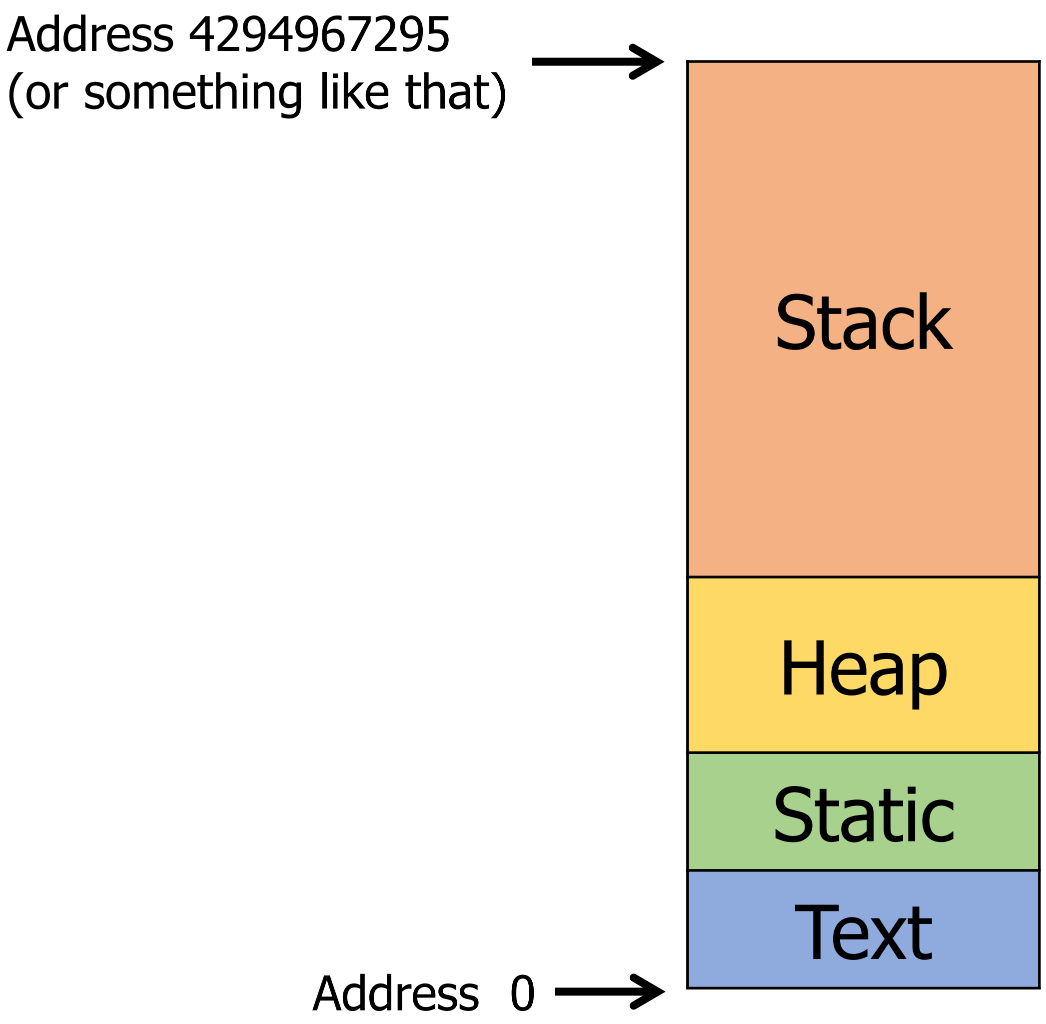


C memory layout

```
int a;
void foo(short b) {
    static int c = 3;

    char* d;
    d = (char*) malloc(4);

    printf("Hello CS211\n");
}
```

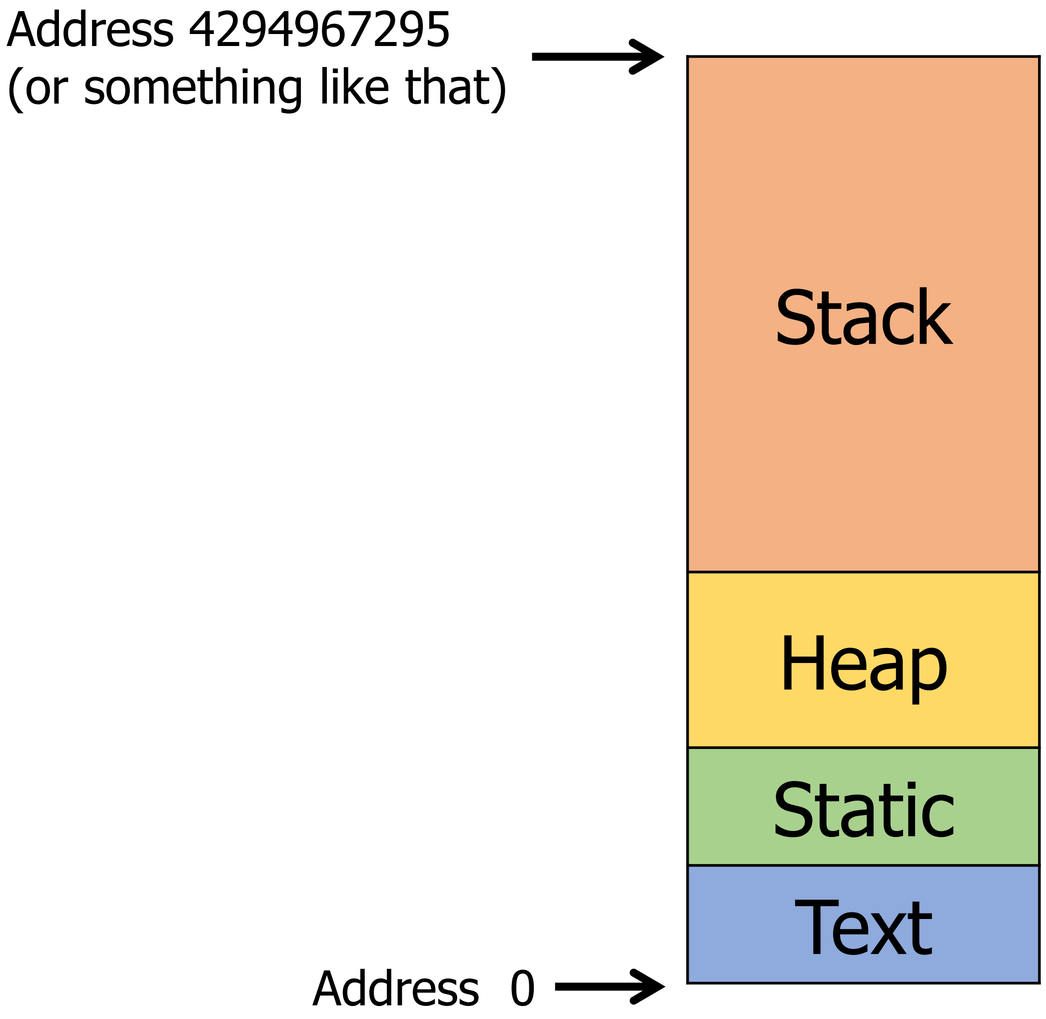


C memory layout

```c
int a;
void foo(short b) {
    static int c = 3;

    char* d;
    d = (char*) malloc(4);

    printf("Hello CS211\n");
}
```



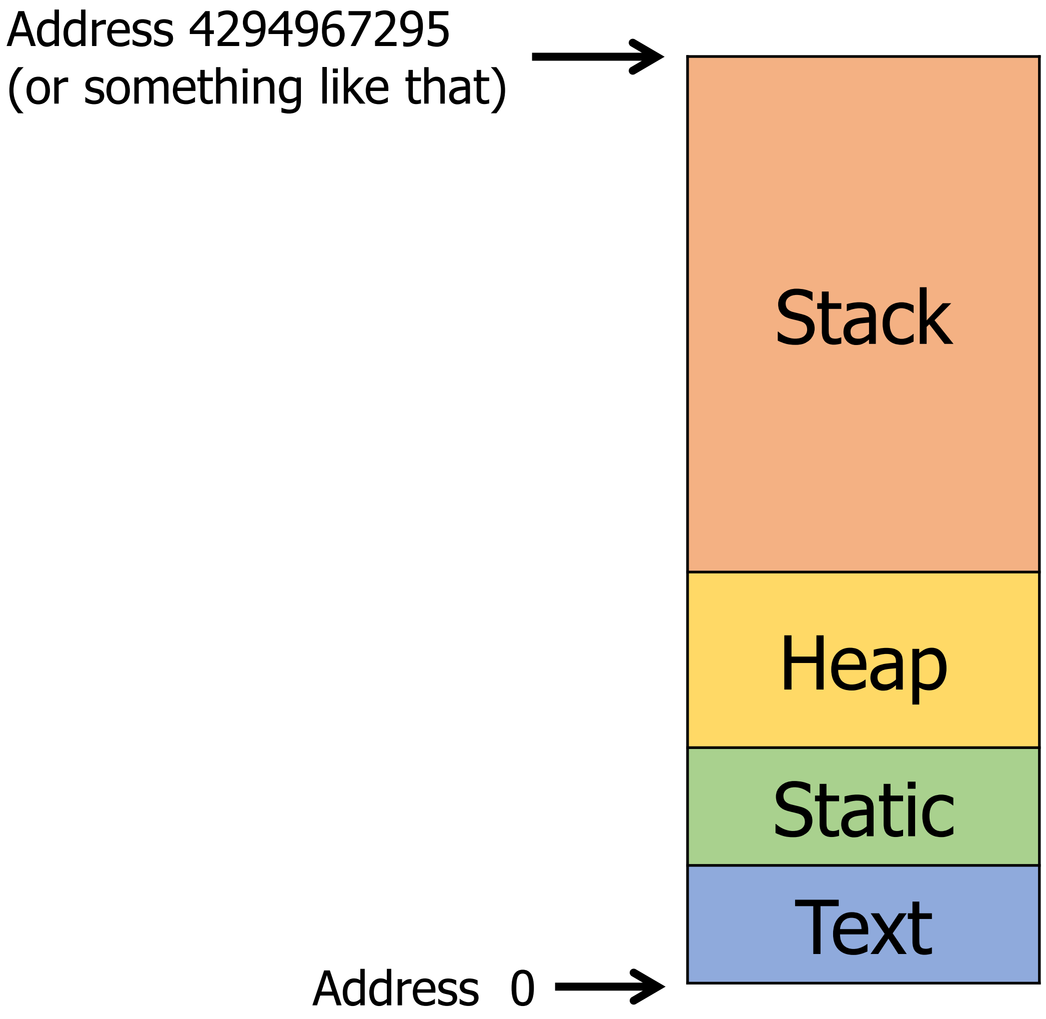
C memory layout

```
int a;
void foo(short b) {
    static int c = 3;

    char* d;
    d = (char*) malloc(4);

    printf("Hello CS211\n");
}
```

Program code goes in the Text section (machine instructions)



Relating memory sections back to lifetimes

- Stack memory has the lifetime of the "scope"
  - From open curly brace to close curly brace
  - Local variables are here

- Static memory has the lifetime of the process
  - From the start of `main()` until it returns
  - Strings are here

- What if you want memory that outlives a function, but doesn't live for the entire duration of the program
  - Heap memory! Claim with `malloc()`

Outline

- Pointers
- Address Sanitizer
- Arguments to main()
- Variable Lifetimes
- Memory Layout

Bonus

- Review: strings

Iterating through a string

string_print.c

```
void print_string_chars(char* string) {
  for (size_t i=0; string[i] != '\0'; i++) {
    printf("String[%d] = '%c'\n", i, string[i]);
  }
}
```

- Note that we didn’t need a length this time!
  - Just iterate until you find the null terminator

String literals cannot be modified

const_strings.c

- `const` in C marks a variable as constant (a.k.a. immutable)
  - Example:

```
const int x = 5;
x++; // Compilation error!
```

- String literals in C are of type `const char*`

```
const char* mystr = “Hello!\n”;
mystr[1] = ‘B’;  // Compilation error!
```

  - Just removing the “`const`” will result in a runtime crash instead…

Making modifiable strings

mutable_strings.c

Two options

1. Create a new character array with enough room for the string and then copy over characters from the string literal
   - Need to be sure to copy over the ‘\0’ for it to be a valid string!

2. Initialize an array with a string literal

```
char mystr[] = “abc”;
```

Creates a character array of length 4 (‘a’, ‘b’, ‘c’, and ‘\0’)