Lecture 05
Lifetimes and Memory

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Fall 2021

Slides adapted from:
Jesse Tov, Vincent St-Amour

Northwestern

Administrivia

- Be sure to list your partner in Gradescope (if any)
  - Otherwise, we're going to end up accusing you of cheating
  - Because your code is going to match perfectly…
  - Once you're partnered on Gradescope, only one of you needs to submit
  - You might need to mark your partner each time you submit?
    - Not sure
- Quiz at the end of class today
  - I'll stop at 1:30ish

Successful homework pattern

1. Submit to Gradescope often
   - There is no penalty for doing so
   - Make sure it at least compiles locally first

2. Check your results on Gradescope

3. Write local tests that duplicate the Gradescope ones

4. Revise your code so the tests pass
   - Use `printf()` to help you debug!

5. Submit to Gradescope again and repeat

Example Gradescope output

Unit test: overlapped_circles 1 (0.0/4.0)

```
#Test: Two circles at different locations that intersect.
#Input:
0 0 2
1 0 2
#Expected Output:
overlapped
#Received Output:
not overlapped

#[X] FAILED
```

- Failure is that Expected and Received Output did not match
- You can duplicate this test locally, which is easier to fix!
  - Create a new test that runs `overlapped_circles()` on `{0,0,2}` and `{1,0,2}`

Be sure to actually test your code locally

- Just running `make` compiles and runs tests

- I'll recompile my code every few lines
  - That way there are never too many bugs to fix at once

- Then I make sure that I'm passing all the tests before uploading
  - And I add new tests whenever I see something weird I'm failing on Gradescope

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- Discuss variable lifetimes: when is a variable no longer valid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_lifetime_memory.tgz
cd 05_lifetime_memory/
```

Outline

- **Strings**
- Arguments to main
- Address Sanitizer
- Variable Lifetimes
- Memory

Strings in C

- C strings are arrays of characters, ending with a null terminator
  - Null terminator: ‘\0’ character, which is the integer value zero
  - No relation to NULL pointers

- String literals in code are arrays of characters
  - And a ‘\0’ is placed at the end of them automatically

“Hello!\n”

MUST use double quotes in C when referring to strings

‘H’	‘e’	‘l’	‘l’	‘o’	‘!’	‘\n’	‘\0’

Working with strings

```
const char* phrase = “The cake is a lie”;

printf(“%s\n”, phrase);      // prints “The cake is a lie\n”
printf(“%c\n”, phrase[0]); // prints “T\n”

char letter = phrase[2];
```

‘T’	‘h’	‘e’	‘ ’	‘c’	‘a’	‘k’	‘e’	‘ ‘	‘i’	‘s’	‘ ‘	‘a’	‘ ‘	‘l’	‘i’	‘e’	‘\n’	‘\0’

phrase:

letter: ‘e’

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’)

The null terminator marks the end of the string

- So, strings are arrays of characters
- And there's no way to know the length of an array in C
- So how does `printf` know when to *stop* printing characters?

- It looks for the null terminator!

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

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!

C has a library for working with strings

`#include <string.h>`

- https://www.cplusplus.com/reference/cstring/
  - Particularly useful:
    - `strlen()` finds the length of a string (not including null terminator)
    - `strcpy()` copies the characters of a string
    - `strcmp()` compares two strings to determine alphabetic order
      - Note: you cannot compare two strings with ==
      - That would just check if the pointers are the same

Outline

- Strings
- **Arguments to main**
- Address Sanitizer
- Variable Lifetimes
- Memory

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

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

- Strings
- Arguments to main
- **Address Sanitizer**
- Variable Lifetimes
- Memory

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!

  - Error locations:
    - 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)
```

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

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

Other AddressSanitizer errors

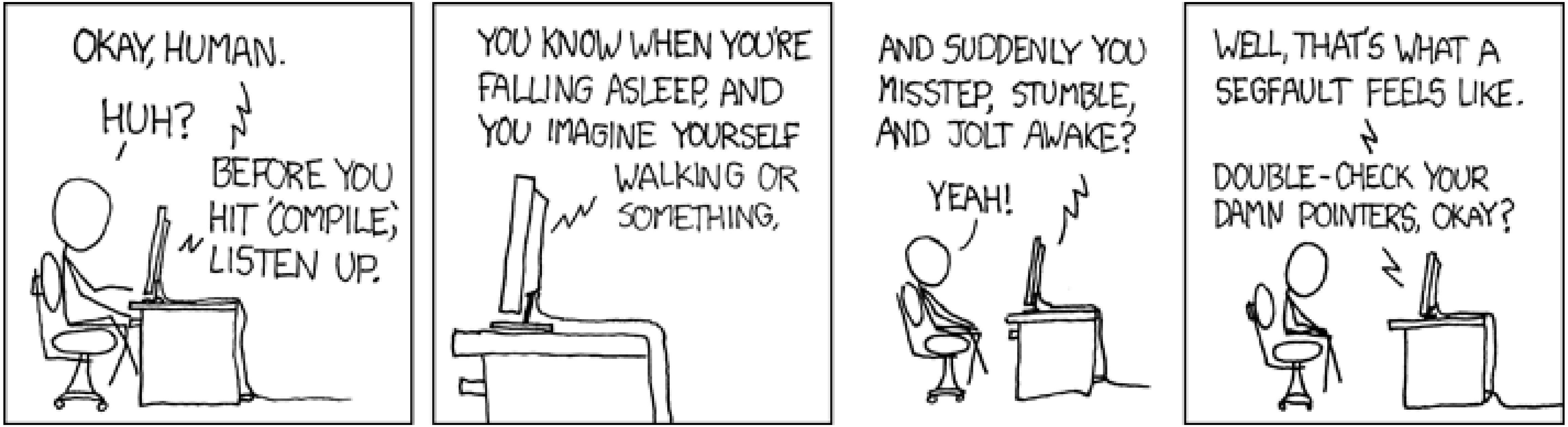
string_print.c

- 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 + relevant xkcd



https://xkcd.com/371/

Outline

- Strings
- Arguments to main
- Address Sanitizer
- **Variable Lifetimes**
- Memory

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

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

  return 0;
}
```

a: 5

Examples of variable lifetimes

```c
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);
}
```

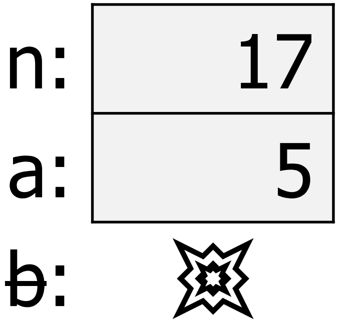
n:	17
a:	5
b:	16

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);
}
```

n:	17
a:	5

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;
}
```

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

So what does the pointer point to???

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

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

- Strings
- Arguments to main
- Address Sanitizer
- Variable Lifetimes
- **Memory**

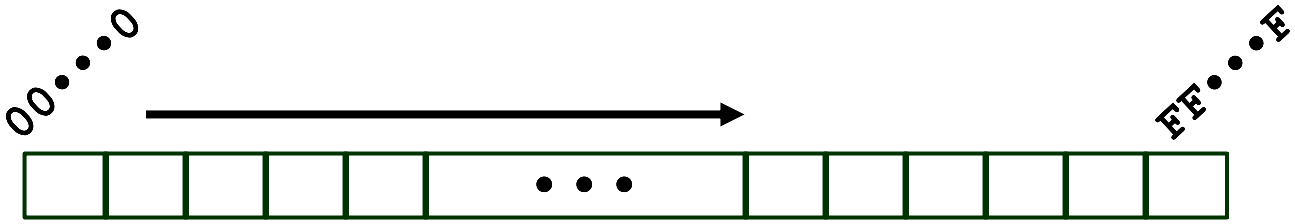
Memory

- 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?



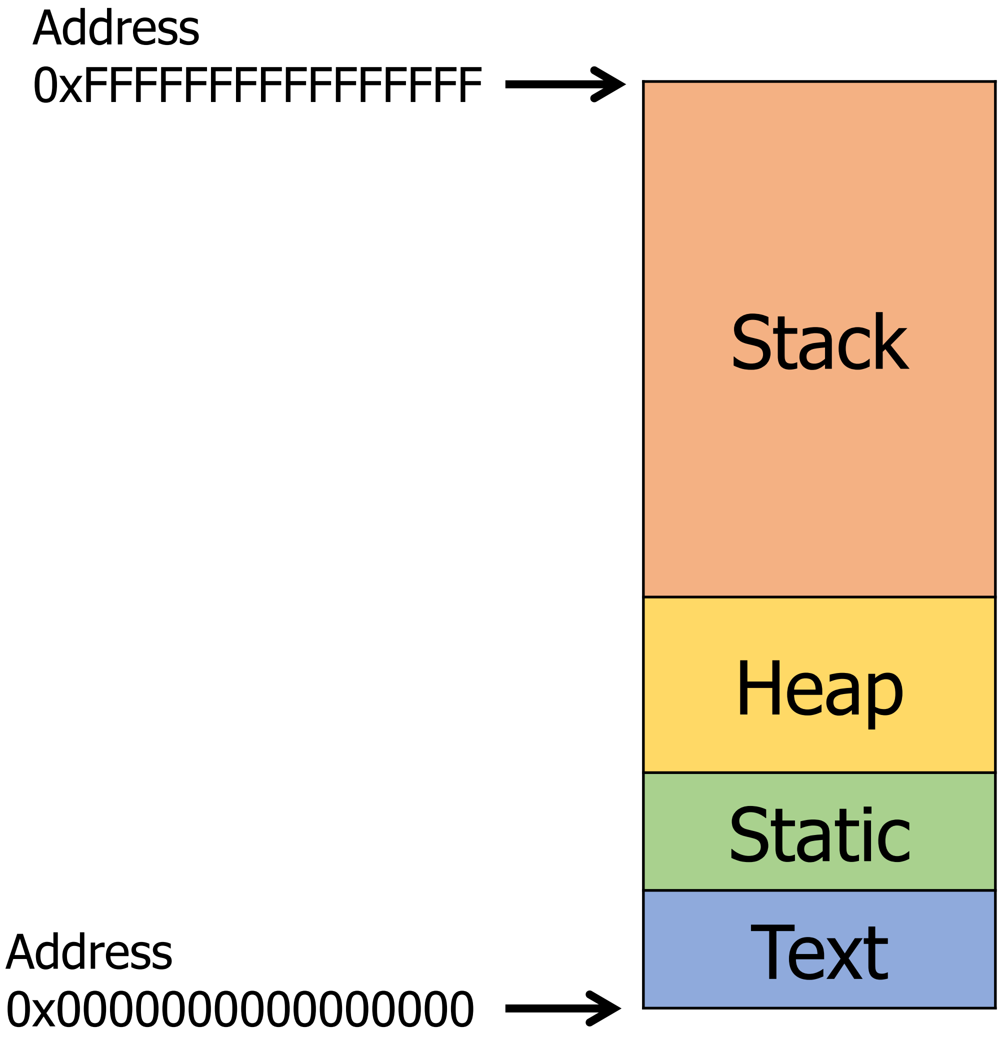
- A nearly infinite series of slots that can be used to hold data
  - Units of memory are known as bytes

  - 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

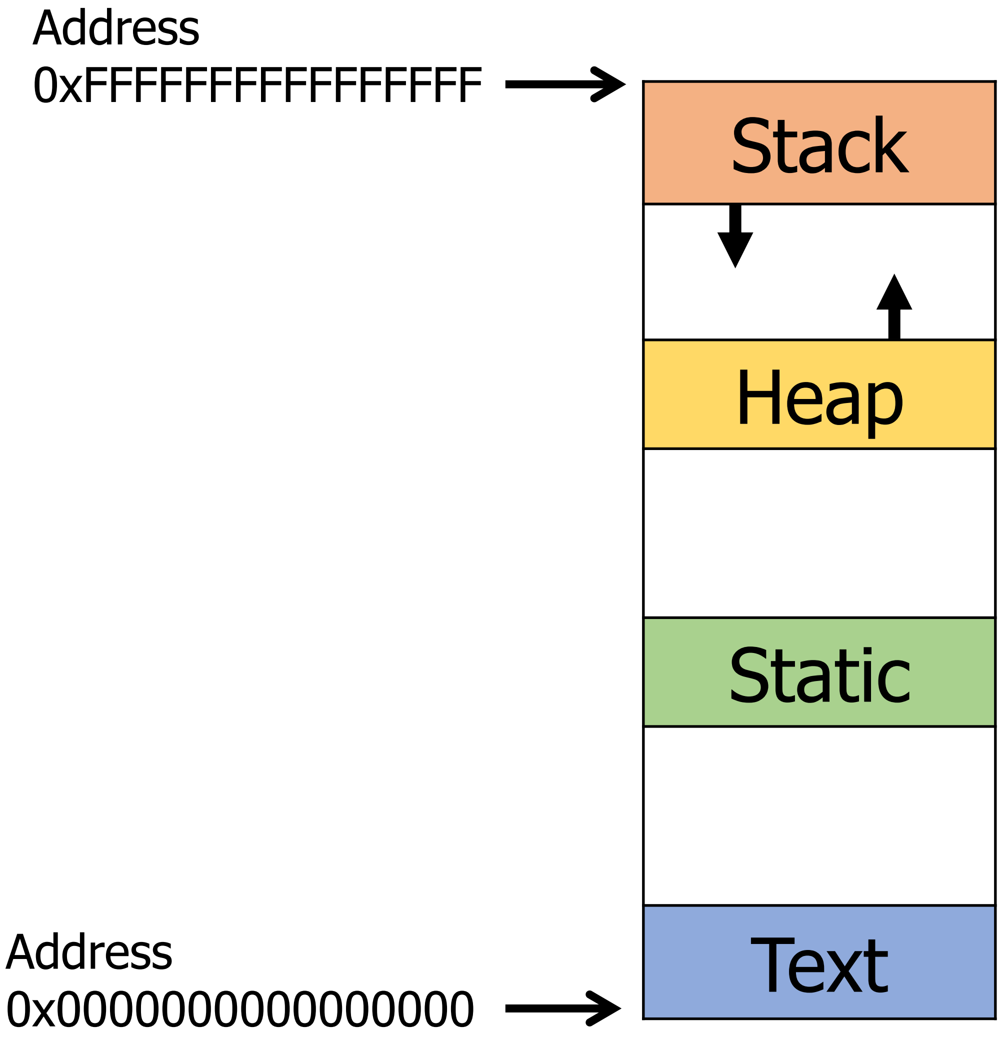
C memory layout

- 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

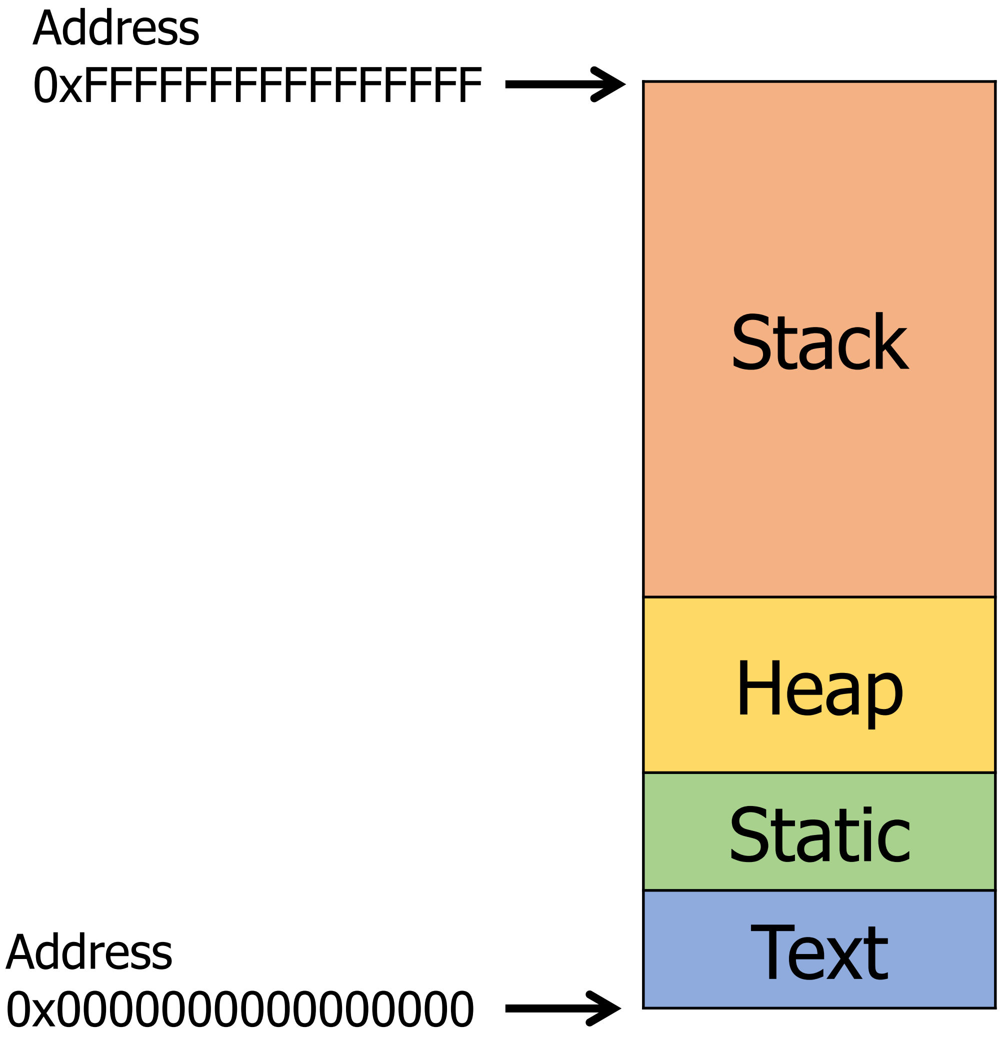


C memory layout

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

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

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

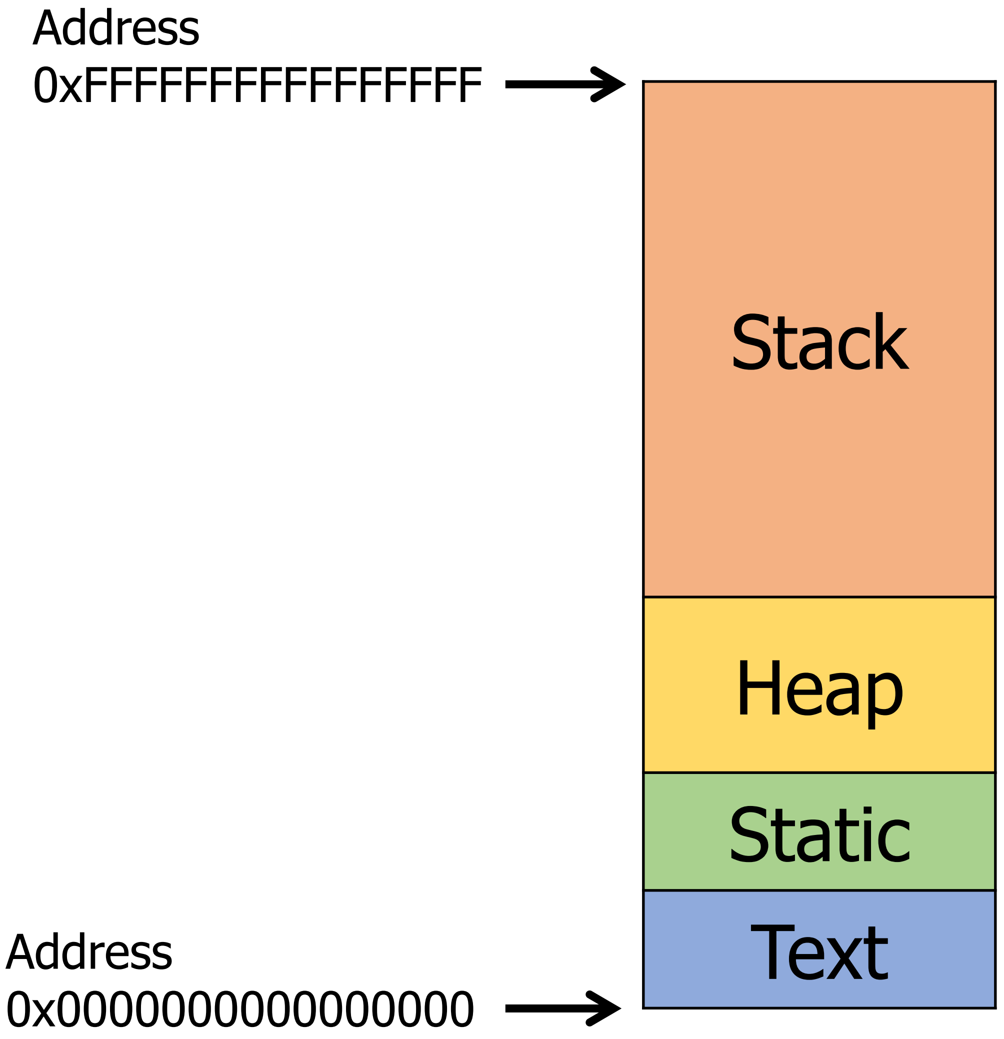


C memory layout

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

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

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

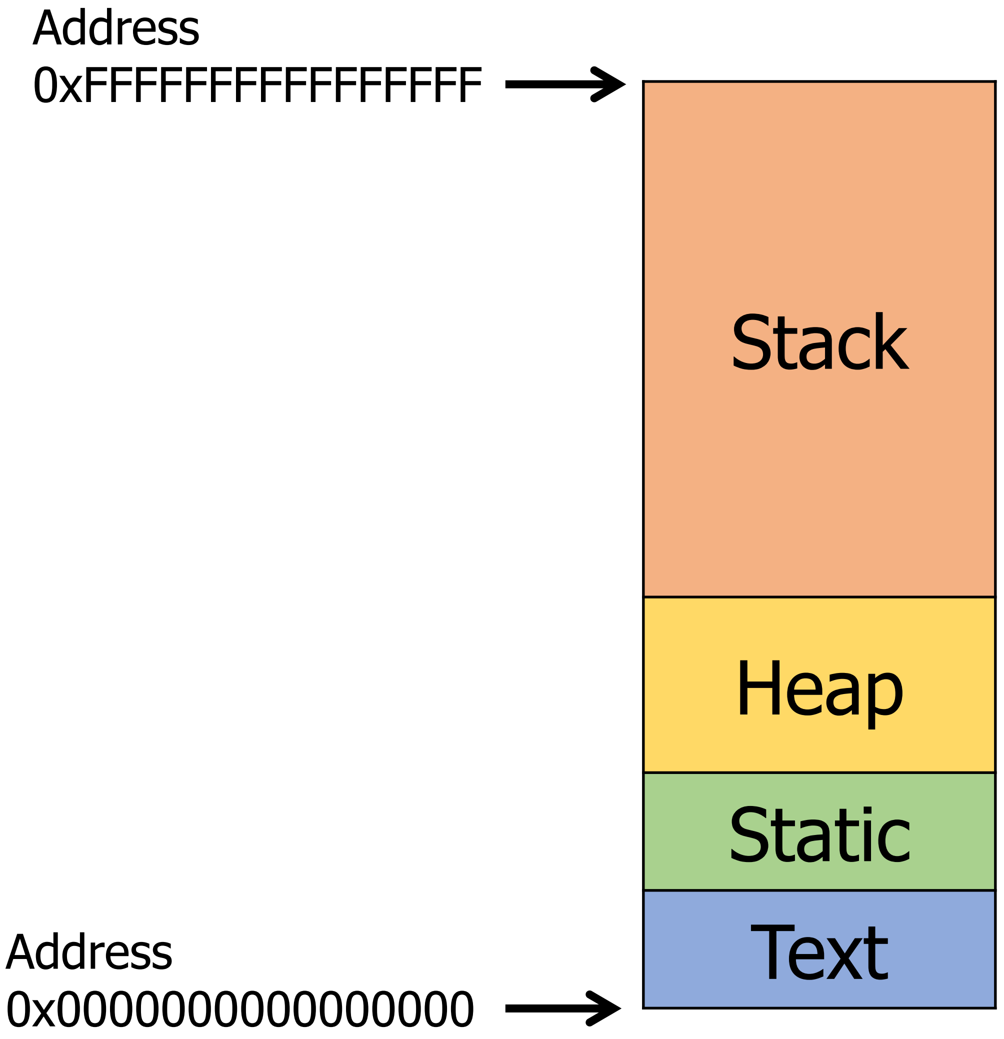


C memory layout

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

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

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

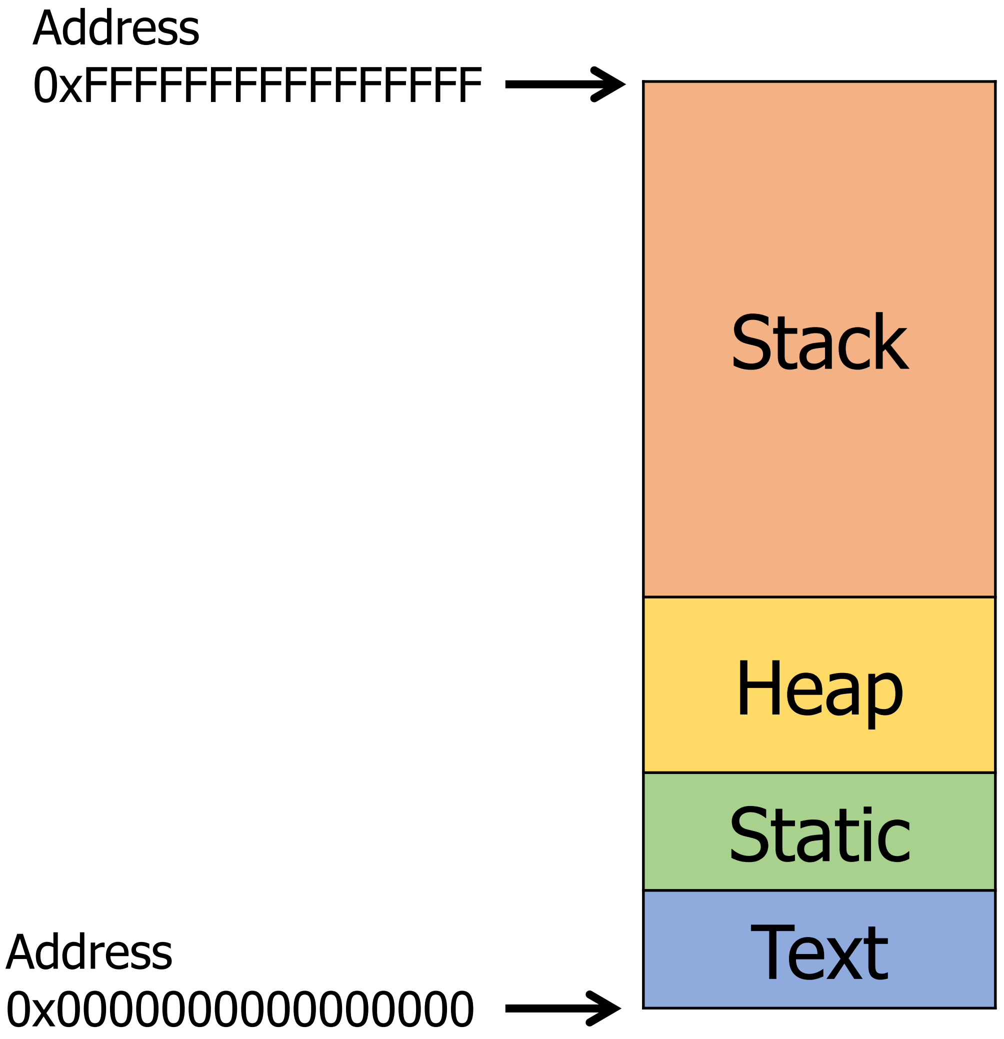


C memory layout

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

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

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

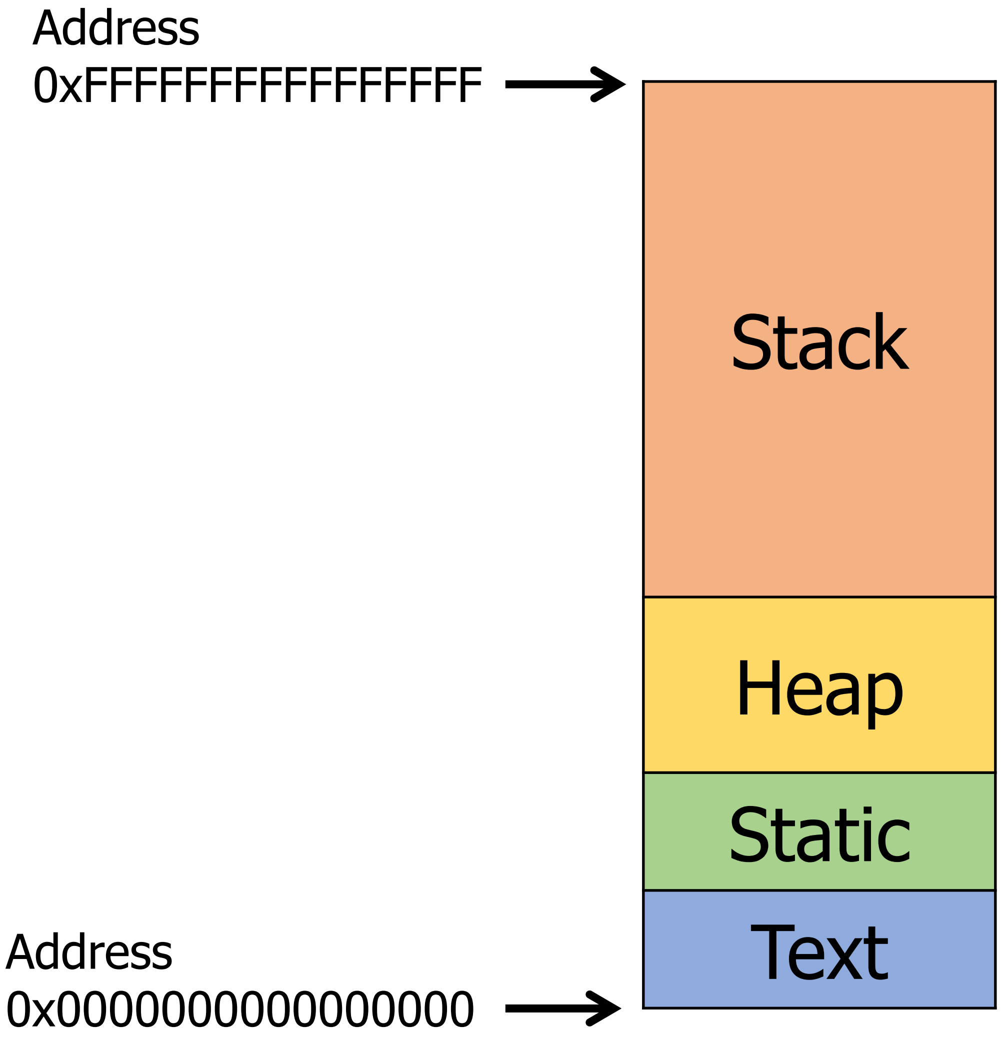


C memory layout

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

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

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

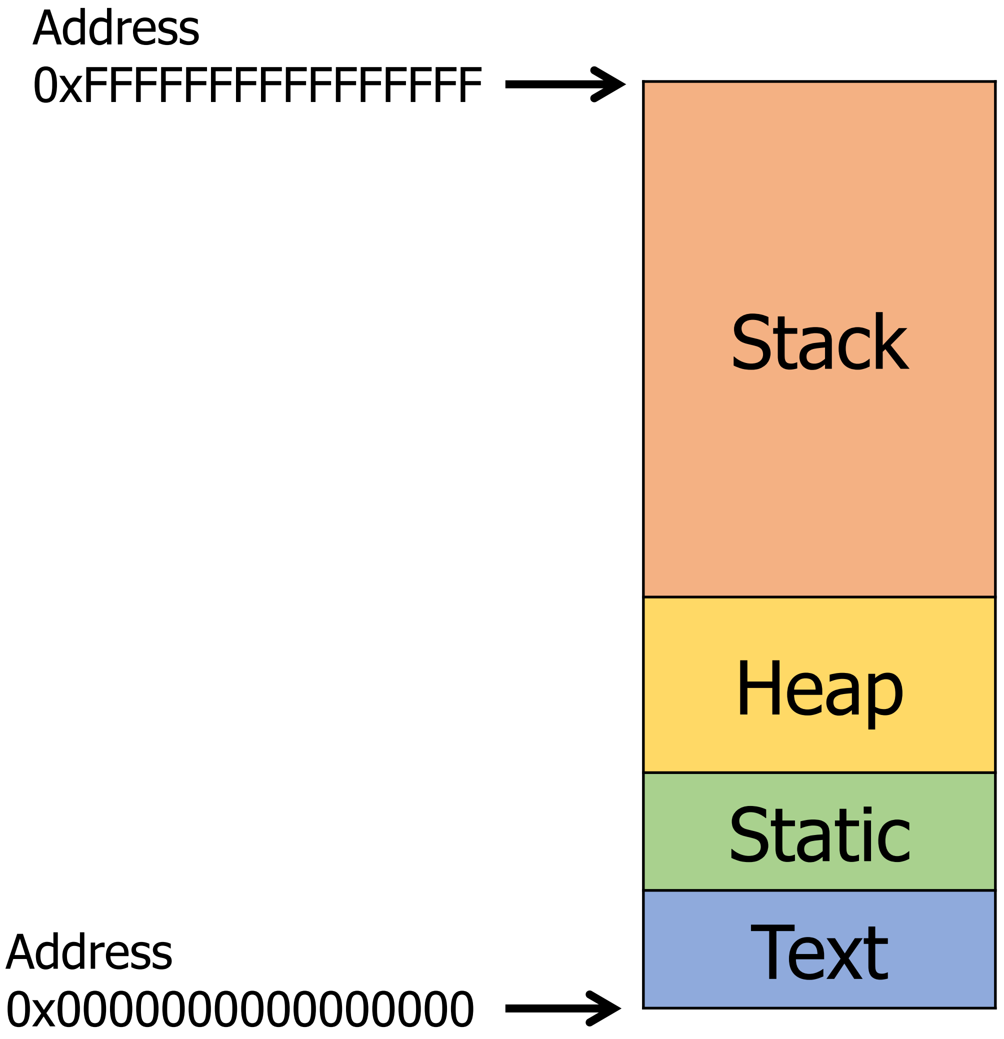


C memory layout

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

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

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

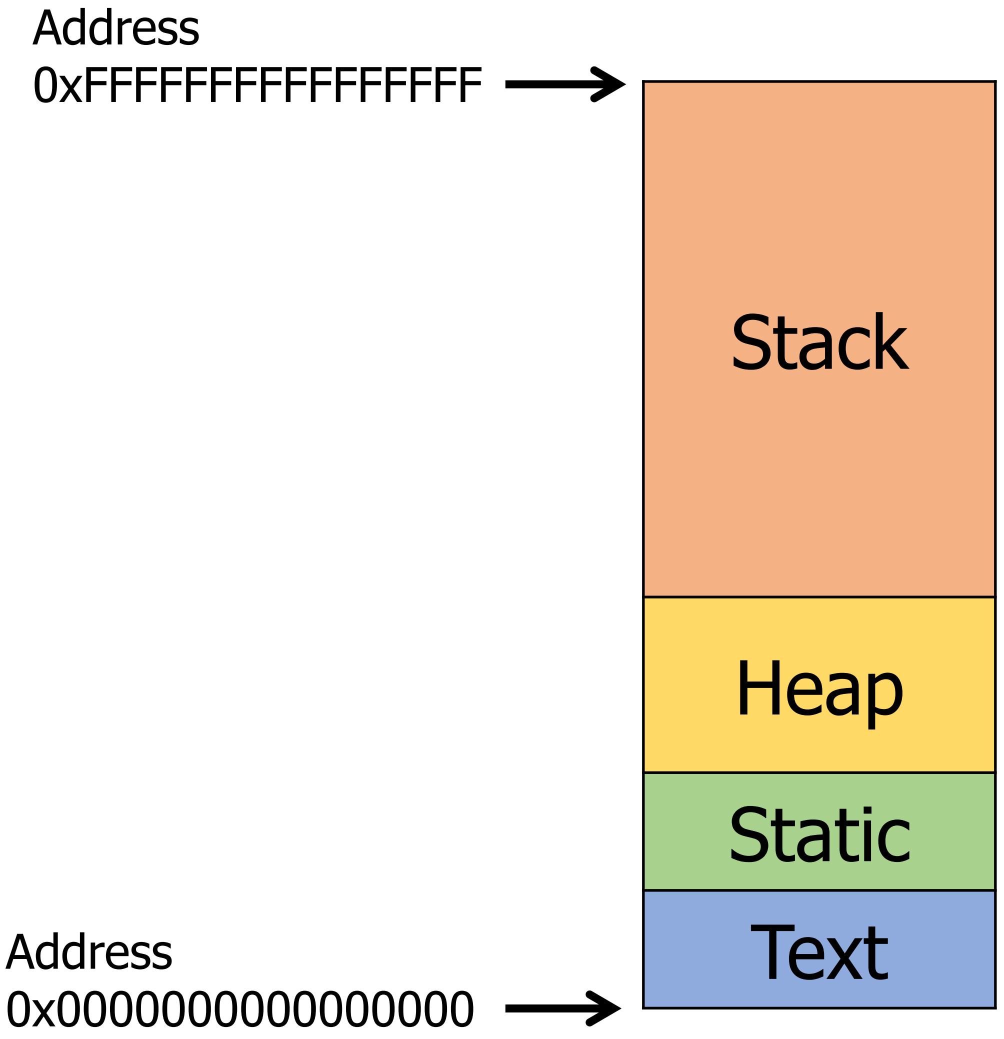


C memory layout

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

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

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

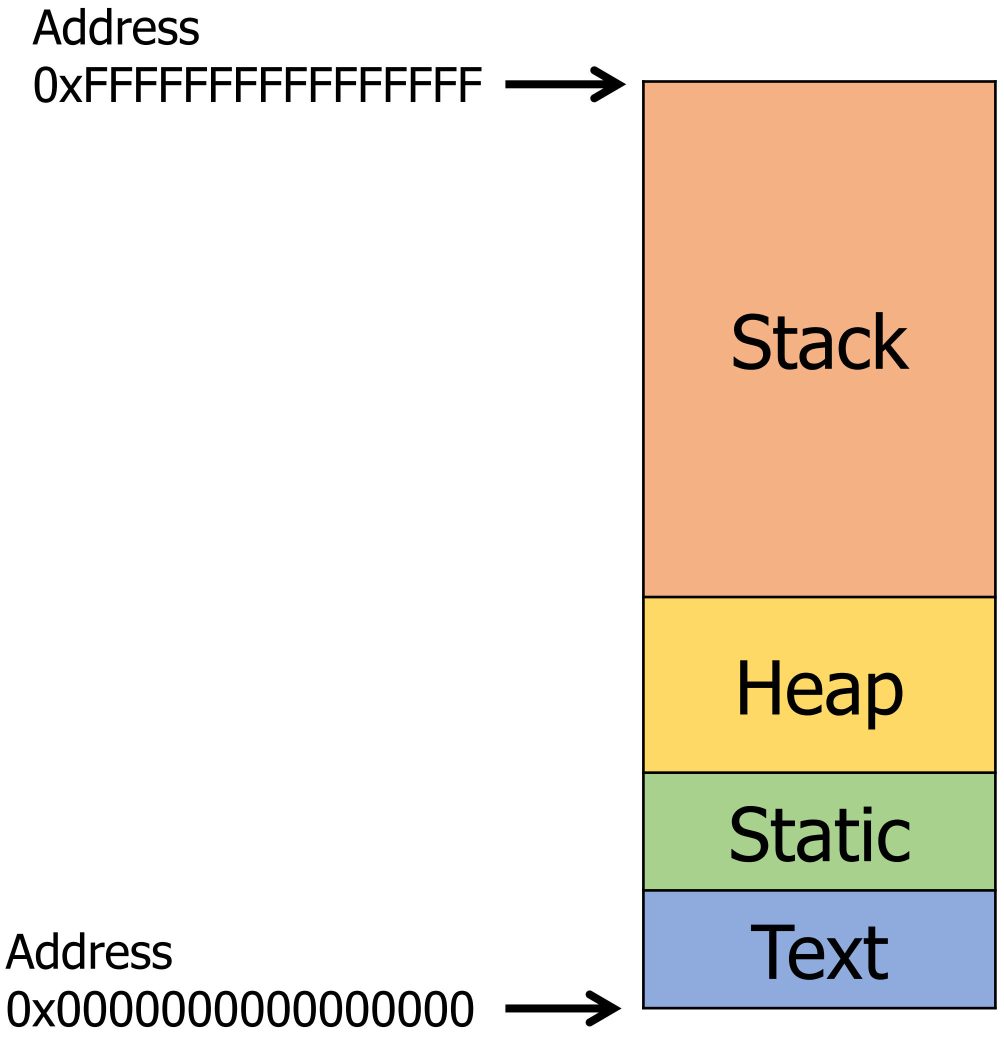


C memory layout

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

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

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



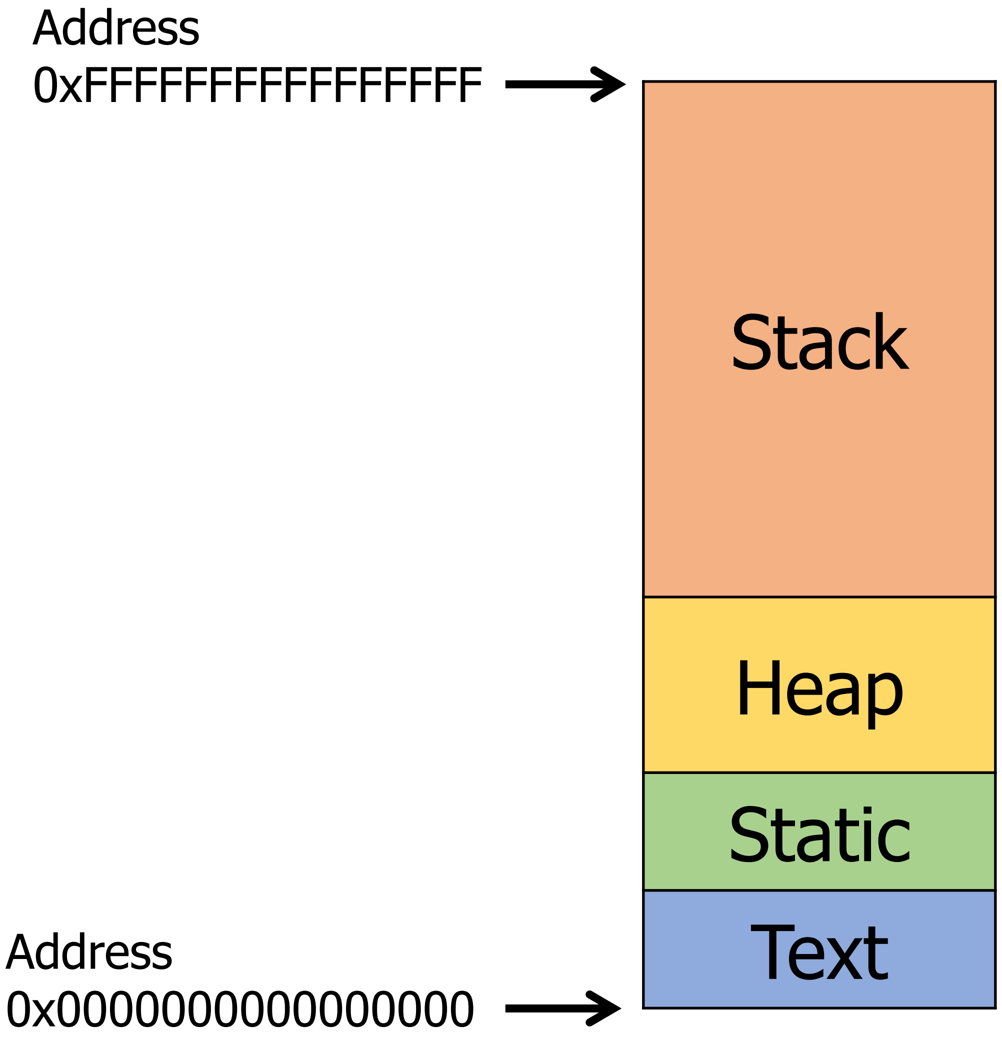
C memory layout

```
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 { to }
  - 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

- Strings
- Arguments to main
- Address Sanitizer
- Variable Lifetimes
- Memory

Outline

- Bonus: Bits and Bytes

Positional Numbering Systems

- The position of a *numeral* (e.g., digit) determines its contribution to the overall number
  - Makes arithmetic simple (compared to, say, roman numerals)
  - Any number has one canonical representation

- Example: base 10
  - $10456_{10} = 1*10^4 + 0*10^3 + 4*10^2 + 5*10^1 + 6*10^0$

- Other bases are also possible
  - Base 2: $10010010_2 = 1*2^7 + 1*2^4 + 1*2^1 = 146_{10}$
  - Base 60, used by the Babylonians
    - The source of 60 seconds in a minute, 60 minutes in an hour
    - And 360 degrees in a circle
  - Base 20, used by the Maya and Gauls (bits remain in French today)

Base 2 Example

- Computer Scientists use base 2 a ***LOT***
- Let's convert 134_{10} to base 2
- We need to decompose 134_{10} into a sum of powers of 2
  - Start with the largest power of 2 that is smaller or equal to 134_{10}
  - Subtract it, then repeat the process

$$134_{10} - 128_{10} = 6_{10}$$
$$6_{10} - 4_{10} = 2_{10}$$
$$2_{10} - 2_{10} = 0_{10}$$

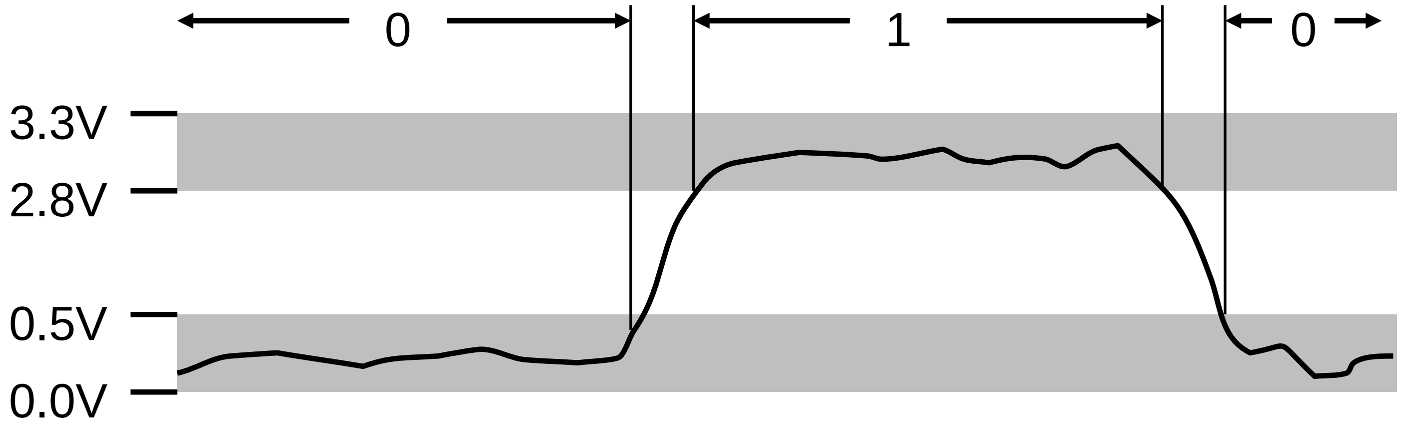
$$134_{10} = \underline{\mathbf{1}}\times 128 + 0\times 64 + 0\times 32 + 0\times 16 + 0\times 8 + \underline{\mathbf{1}}\times 4 + \underline{\mathbf{1}}\times 2 + 0\times 1$$

$$134_{10} = \underline{\mathbf{1}}\times 2^7 + 0\times 2^6 + 0\times 2^5 + 0\times 2^4 + 0\times 2^3 + \underline{\mathbf{1}}\times 2^2 + \underline{\mathbf{1}}\times 2^1 + 0\times 2^0$$

$$134_{10} = 10000110_2$$

Why computers use Base 2

- Simple electronic implementation
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires

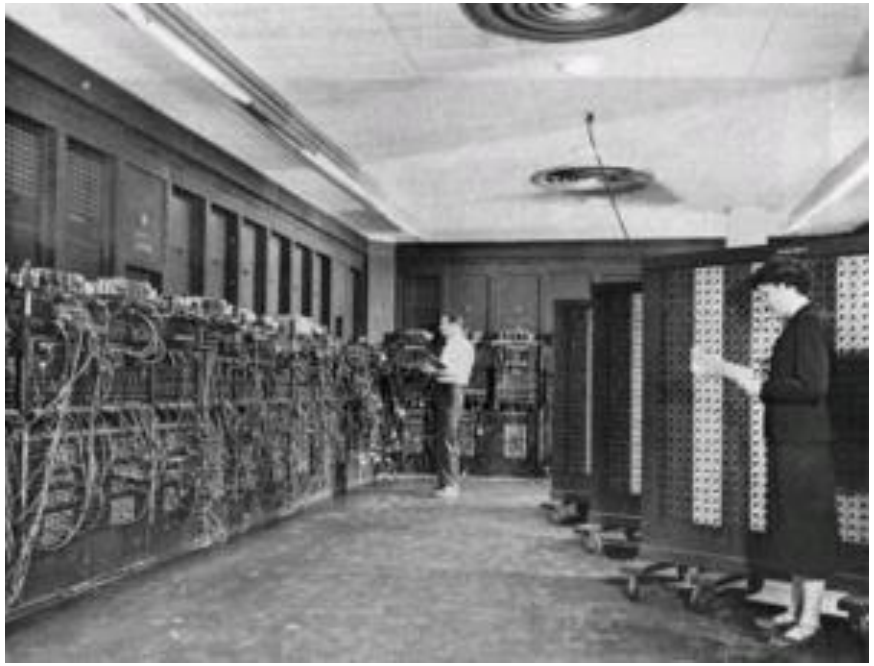


- Straightforward implementation of arithmetic functions

- (Pretty much) all computers use base 2

Why don't computers use Base 10?

- Because implementing it electronically is a pain
  - Hard to store
    - ENIAC (first general-purpose electronic computer) used 10 vacuum tubes / digit
  - Hard to transmit
    - Need high precision to encode 10 signal levels on single wire
  - Messy to implement digital logic functions
    - Addition, multiplication, etc.



Base 16: Hexadecimal

- Writing long sequences of 0s and 1s is tedious and error-prone
  - And takes up a lot of space on a page!
- So we'll often use base 16 (also called *hexadecimal*)
- $16 = 2^4$, so every group of 4 bits becomes a hexadecimal digit (or *hexit*)
  - If we have a number of bits not divisible by 4, add 0s on the left (always ok, just like base 10)
- Base 2 = 2 symbols (0, 1)
  Base 10 = 10 symbols (0-9)
  Base 16, need 16 symbols
  - Use letters A-F once we run out of decimal digits

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

0 0 1 0 | 1 0 0 1 | 0 1 1 1 | 1 0 1 1 ⟶ 0x297B

"0x" prefix = it's in hex

Bytes

- A single bit doesn't hold much information
  - Only two possible values: 0 and 1
  - So we'll typically work with larger groups of bits

- For convenience, we'll refer to groups of 8 bits as ***bytes***
  - And usually work with multiples of 8 bits at a time
  - Conveniently, 8 bits = 2 hexits

- Some examples
  - 1 byte: 0b01100111 = 0x67
  - 2 bytes: $11000100\ 00101111_2$ = 0xC42F

"0b" prefix = it's in binary

Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:

  - Convert binary to decimal:

Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:
    - 0x4 -> 0b0100 0x2 -> 0b0010 0x42 -> 0b 0100 0010
  - Convert binary to decimal:
    - $1*2^6 + 1*2^1 = 64 + 2 = 66$

Practice problem

- **Convert 0x42 to decimal**

- Critical thinking:
  - What are the maximum and minimum values?
    - Minimum 0 (0x00)
    - Maximum 255 (0xFF)

  - How big is 0x42 out of 0xFF?
    - ~25% (0x40, 0x80, 0xC0, 0x100)
    - So 255/4 ≈ 240/4 ≈ 60

Big idea: bits can be used to represent anything

- Depending on the context, the bits **`11000011`** could mean
  - The number 195
  - The number -61
  - The number -1.1875
  - The value `True`
  - The character '├'
  - The **`ret`** x86 instruction

- You have to know the **context** to make sense of any bits you have!
  - People and software they write determine what the bits actually mean