

Lecture 06

Dynamic Memory

CS211 – Fundamentals of Computer Programming II
Branden Ghena – Fall 2021

Slides adapted from:
Jesse Tov

Administrivia

- Homework 2 due today
 - Homework 2 self-eval opens tomorrow
 - Remember that you can use slip days
 - Make sure that at least the final submission has both partners on it
- Lab04 will release later today
- Homework 3 will release late today or sometime tomorrow

Today's Goals

- Explore memory and how C organizes it
- Understand how dynamic memory works
 - And what to be careful about
 - Discuss related ideas:
 - How much memory do C types need?
 - How do we avoid common dynamic memory mistakes?

Getting the code for today

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

```
tar -xkvf ~cs211/lec/06_dynamic.tgz
```

```
cd 05_dynamic/
```

Outline

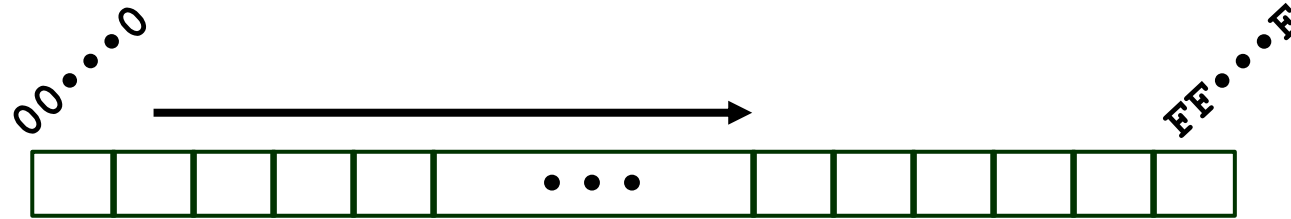
- **Memory**
- Dynamic Memory Allocation
 - Dynamic Memory Example
- Memory Sizes of C Types
- Ownership

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)



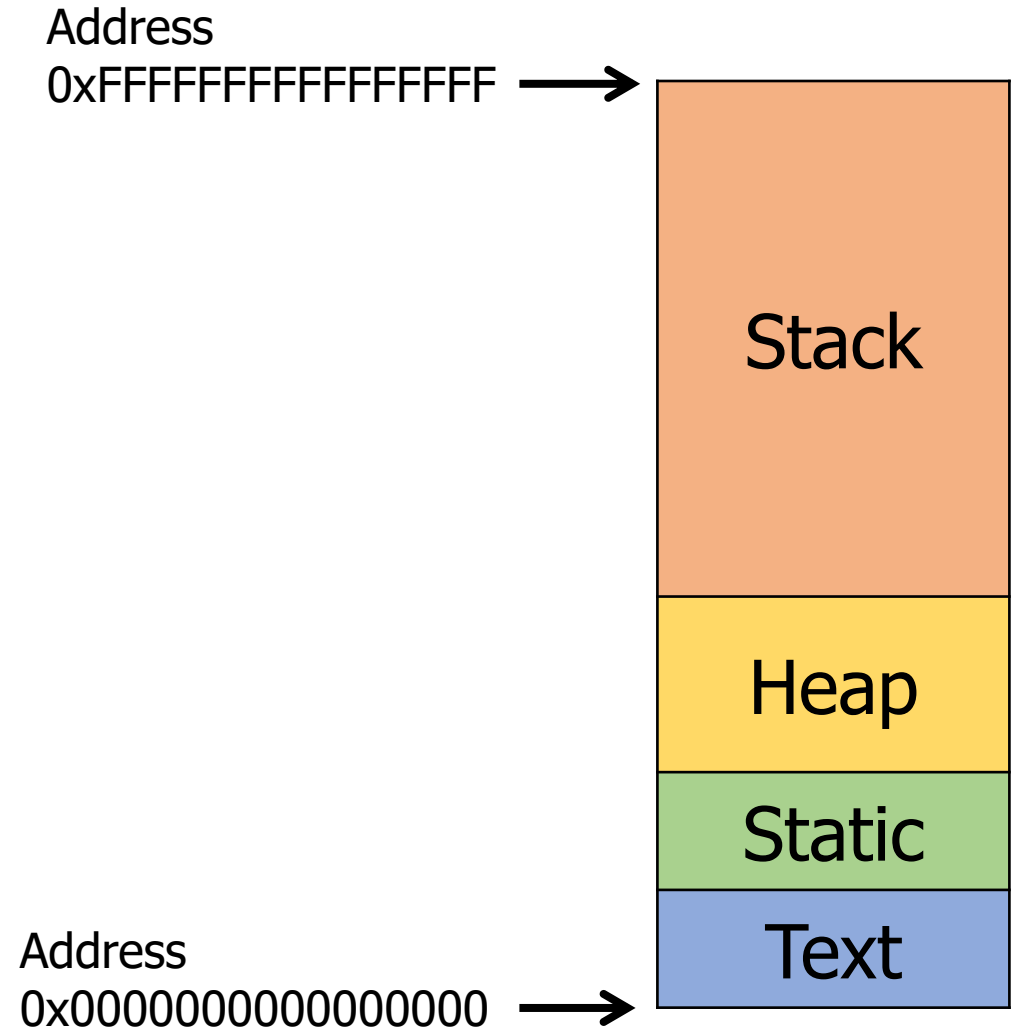
What is memory conceptually?



- 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

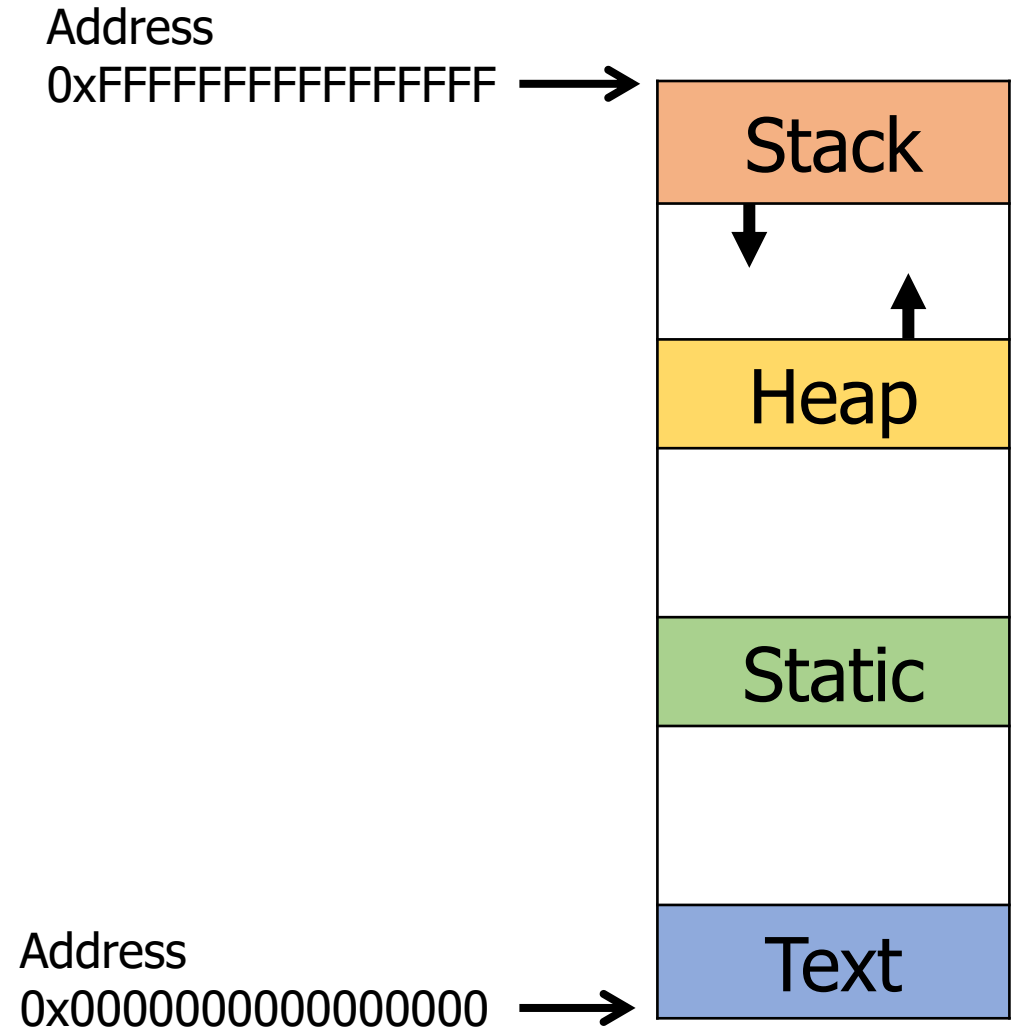
C memory layout

- 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



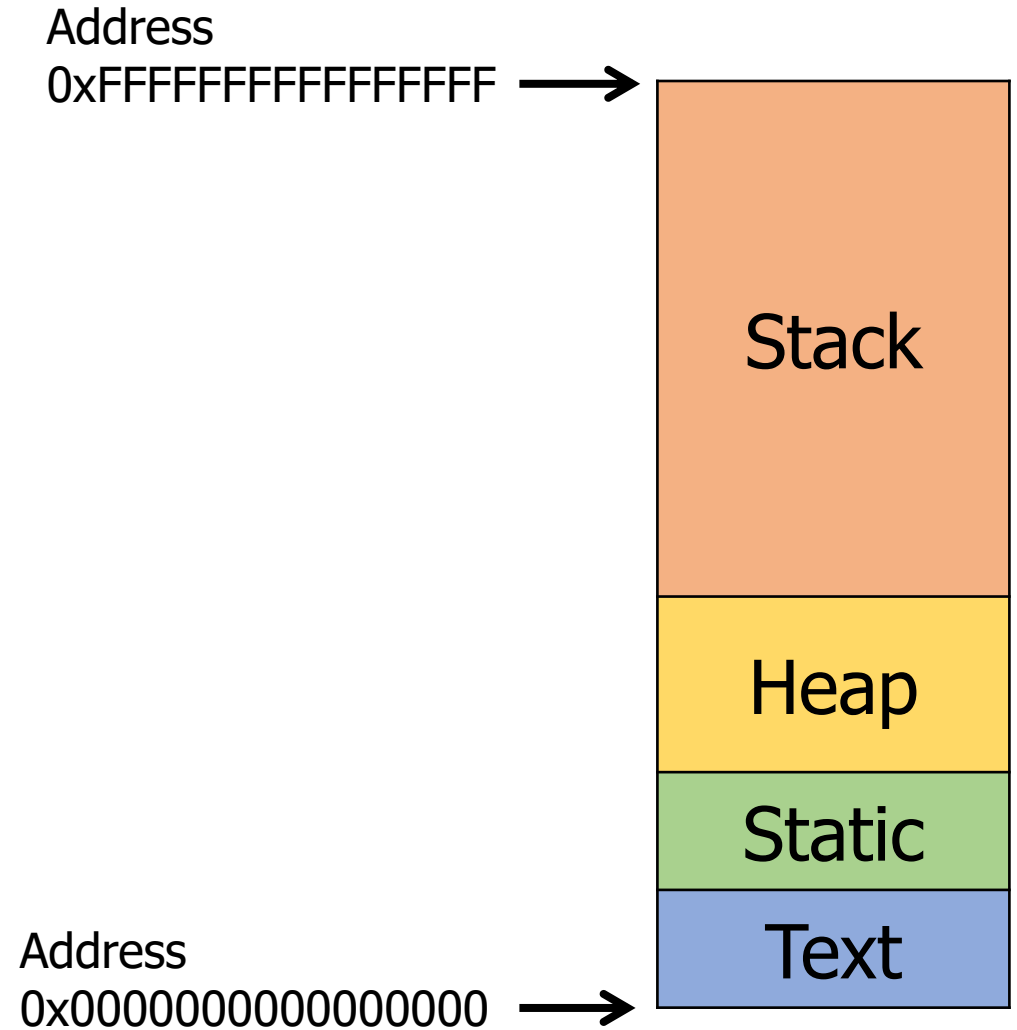
C memory layout

- 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");
```

```
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

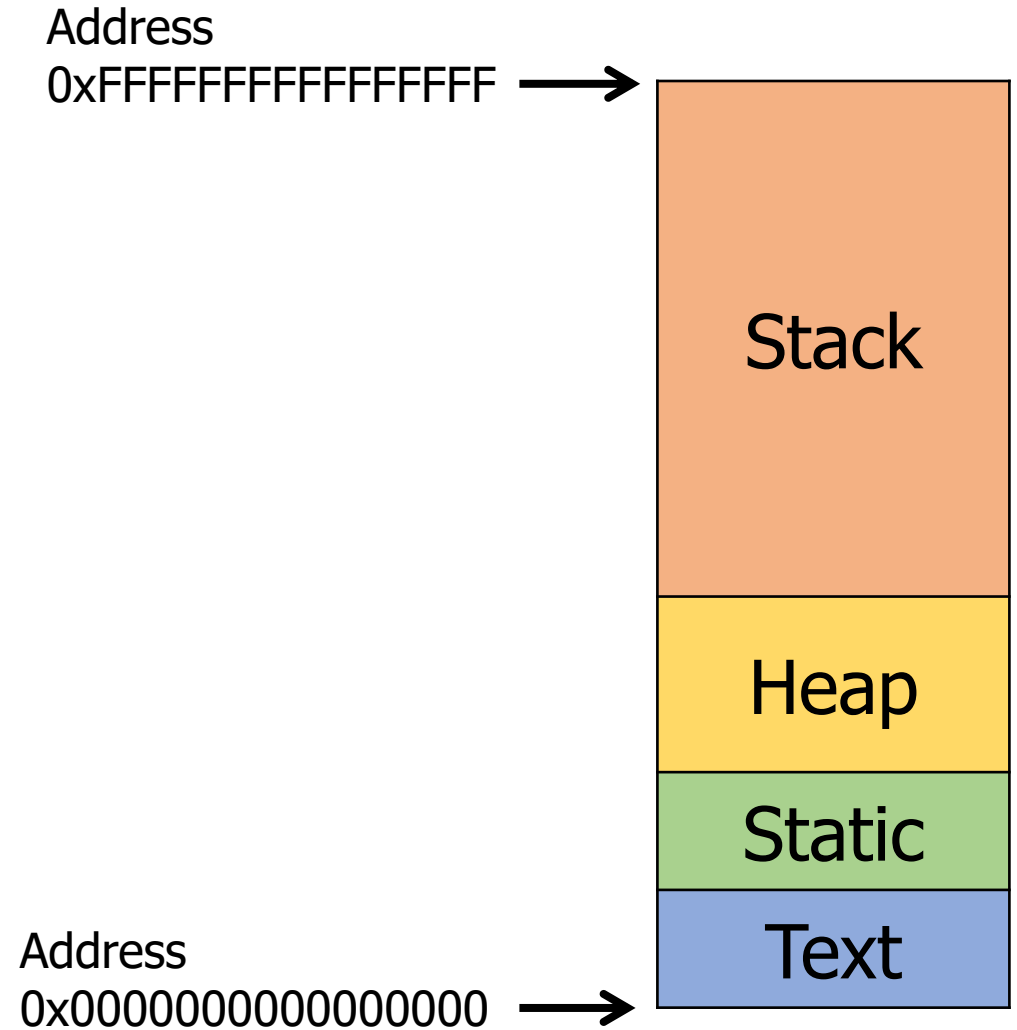
Text

Address

0x0000000000000000 →

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");
```

```
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
int a;
```

```
void foo(short b) {
```

```
    static int c = 3;
```

```
    char* d;
```

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

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

```
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

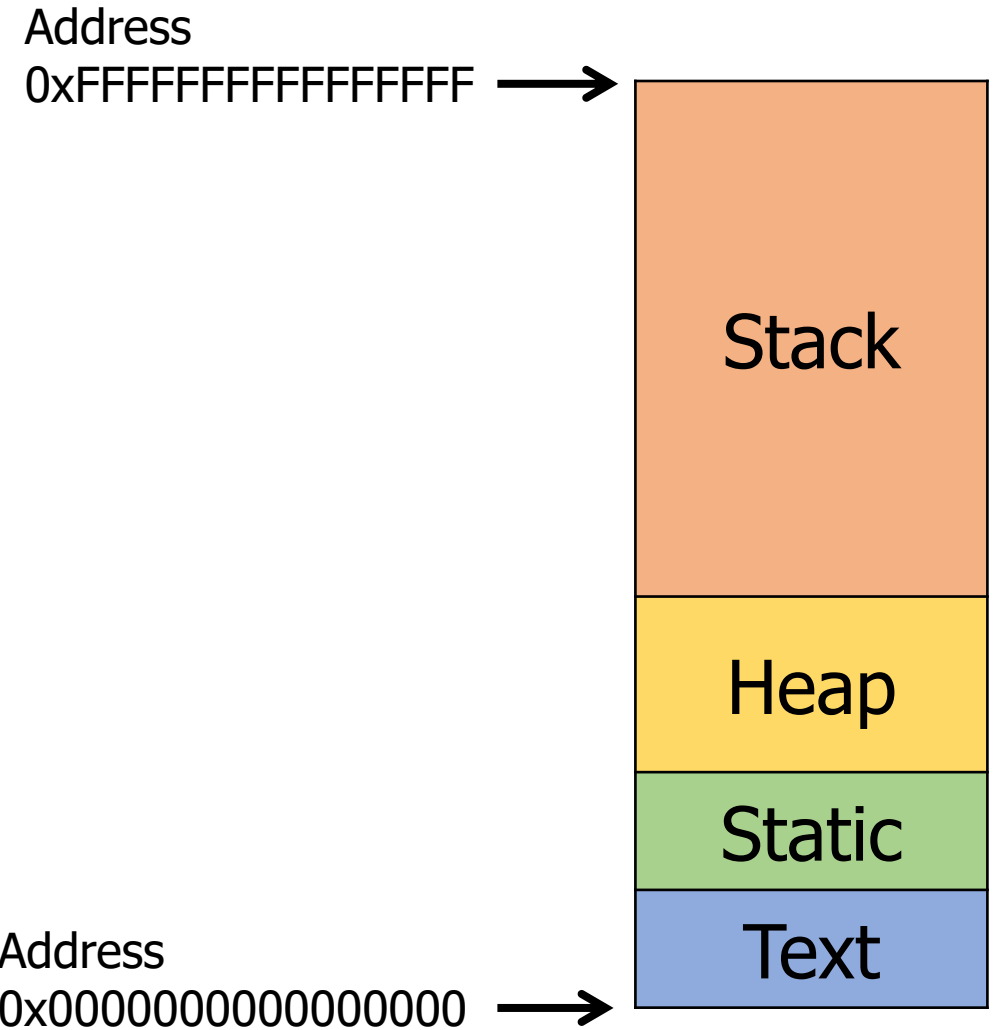
Text

Address

0x0000000000000000 →

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");
```

```
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
int a;  
void foo(short b) {  
    static int c = 3;  
  
    char* d;  
    d = (char*) malloc(4);  
  
    printf("Hello CS211\n");  
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

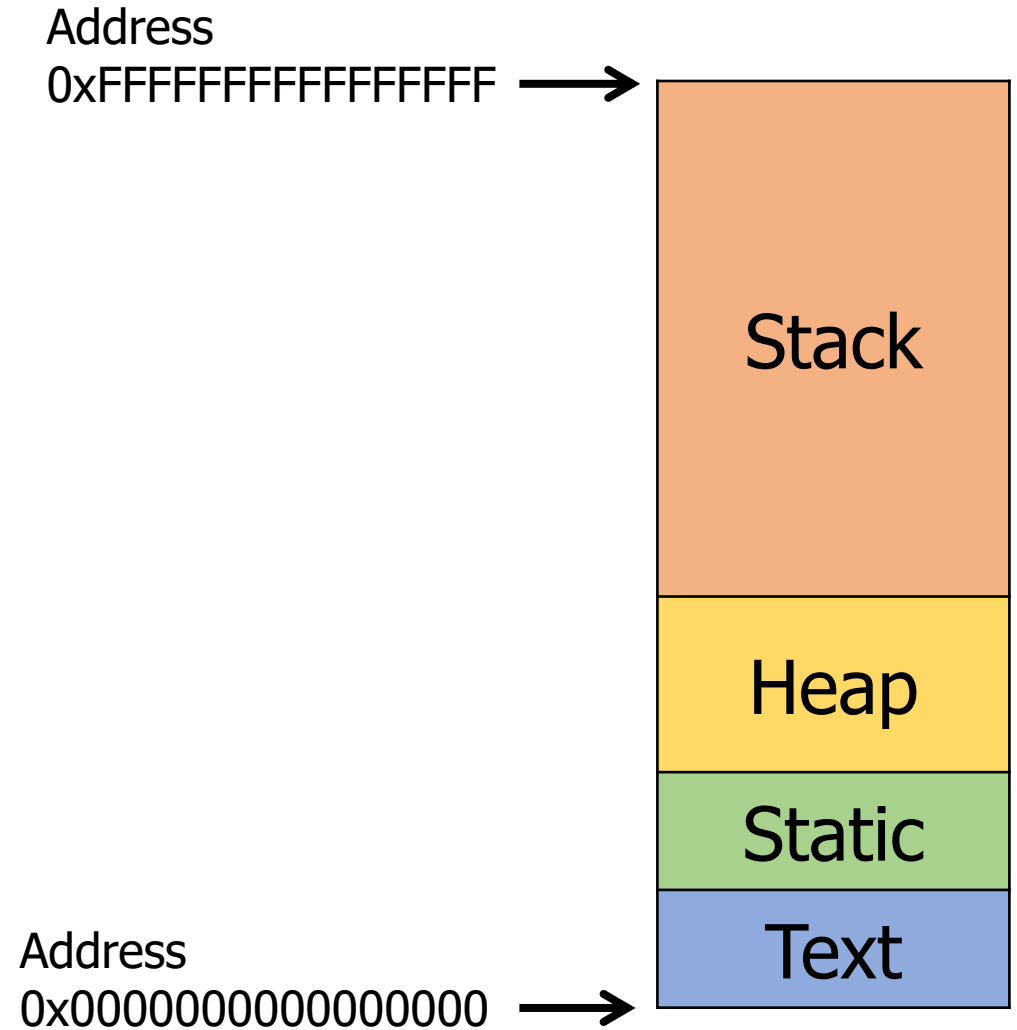
Address

0x0000000000000000 →

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)




Outline

- Memory
- **Dynamic Memory Allocation**
 - Dynamic Memory Example
- Memory Sizes of C Types
- Ownership

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”

Lifetimes go from creation to end brace }

```
test(17);
```

n: ✨

```
void test(int n) {
```

a: ✨

```
    int a = 5;
```

```
    if (n >= a) {
```

```
        int b = 16;
```

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

```
    }
```

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

→ }

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()`

Allocate memory with malloc()

```
void* malloc(size_t size)
```

- Requests `size` bytes of memory from the heap
- Returns a pointer to this new **object**
 - Not associated with any variable (sort of like string literals)
 - It has no value by default
- The object persists until it is manually deallocated
 - Deallocated through a call to `free()`

Malloc return value

```
void* malloc(size_t size)
```

- `void*` is a special pointer type in C
 - “A pointer to nothing” (or to *anything*)
 - Must be cast into the desired type before dereferencing

```
int* myptr = (int*)malloc(sizeof(int));
```
- `malloc()` can fail!!
 - The return value is `NULL` if it was unable to allocate the memory
 - You always need to check the return value of `malloc()` before using it

Deallocate memory with free()

```
void free(void* ptr)
```

- Returns the memory at the pointer to the heap
- Only works if the memory address was given by `malloc()`
- Must be called when you are finished with the memory
 - Or else you have a “memory leak”
- Memory leaks occur when `malloc()`’d memory is not `free()`’d
 - Process slowly accumulates memory that it was given, but can’t access anymore
 - Keeps using more and more memory when it runs for a long time
 - Until the OS eventually has to kill it

Free needs to be used carefully

```
void free(void* ptr)
```

- If you pass in a pointer that wasn't created with `malloc()`:
 - **UNDEFINED BEHAVIOR** (often a segfault)
 - `free(NULL)` is fine though
- Once memory is freed, it must **NEVER** be used again
 - Or else... **UNDEFINED BEHAVIOR** (surprise!)
 - Definitely don't free it twice

Rules for dynamic memory allocation

1. Every pointer returned by `malloc()` must be NULL-checked
 2. Every object returned by `malloc()` must have its address passed to `free()` exactly once
 3. After an object is freed, it must not be accessed or freed again
 4. An object not obtained from `malloc()` must not be freed
- Breaking any of these rules leads to **UNDEFINED BEHAVIOR**

Pros/cons of dynamic memory allocation

- Pros

- You can create exactly as much memory as you want
- It lives for exactly as long as you need it
 - Not tied to any particular function

- Cons

- **UNDEFINED BEHAVIOR** *everywhere* if you're not careful
- Must be sure to later `free()` all memory given by `malloc()`

Other “dynamic memory family” functions

```
void* calloc(size_t num, size_t size)
```

- Allocates a block of memory for `num` elements, each of `size` bytes
- Zeros each element in the memory

```
void* realloc(void* ptr, size_t size)
```

- Changes the size of the memory block pointed to by `ptr`
- Might return the same pointer, might be a new pointer
 - Frees the old pointer if giving you a new one
 - Values in the memory are maintained
- Can be used to increase the size of a `malloc()`'d array!

Break + Question

```
int testfunction (int i) {  
    int* ptr = (int*)malloc(sizeof(int));  
    *ptr = i;  
    printf("Before: %d\n", *ptr);  
    free(ptr);  
    return *ptr;  
}
```

```
int main(void) {  
    printf("After: %d\n", testfunction(5));  
    return 0;  
}
```

What values does this program print?

Break + Question

```
int testfunction (int i) {  
    int* ptr = (int*)malloc(sizeof(int));  
    *ptr = i;  
    printf("Before: %d\n", *ptr);  
    free(ptr);  
    return *ptr;  
}
```

```
int main(void) {  
    printf("After: %d\n", testfunction(5));  
    return 0;  
}
```

What values does this program print?

It prints: "Before: 5\n"

After that: **UNDEFINED BEHAVIOR**
"use-after-free" error

Outline

- Memory
- **Dynamic Memory Allocation**
 - **Dynamic Memory Example**
- Memory Sizes of C Types
- Ownership

Live coding example

dynamic_string_example.c AND toupper_starter.c

- Let's write a program that uses dynamic memory to create uppercase versions of string literals

- Functions:

```
char* make_mutable_string(const char*);
```

```
void uppercase_string(char*);
```

- Useful library function: toupper()

```
void print_and_destroy(char*);
```

```
int main(void)
```

Outline

- Memory
- Dynamic Memory Allocation
 - Dynamic Memory Example
- **Memory Sizes of C Types**
- Ownership

How much memory do various types in C take?

- Actually a complicated question
- Many types in C are defined as a “minimum size”
 - Where they are bigger on some machines and smaller on others
- **HOWEVER**, if you work on a modern 64-bit computer, you can make some assumptions
 - And we’ll talk about those assumptions

Standard sizes of C types on modern (64-bit) computers

- 1 byte
 - char, unsigned char, signed char
 - bool
- 2 bytes
 - short, unsigned short, signed short
- 4 bytes
 - int, unsigned int, signed int
 - float
- 8 bytes
 - long, unsigned long, signed long
 - double
 - Every pointer type!

What about more complex things?

- Arrays

- Easy
- Number of slots times the size of each slot
- Example: `int array[8]` is 32 bytes

- Structs

- Complicated (we'll explore more in CS213)
- At least the size of every field inside it
 - Plus more depending on the order of the fields for efficiency reasons

Don't assume you know these sizes in code

1. It's hard to remember all of this
 2. They could be different on a different computer system
 - Especially 32-bit systems, microcontrollers, or other special computers
- Use `sizeof()` to figure out the number of bytes a type is
 - Not a library function, actually an operator in C
 - Primarily used on types, but can be used on variables too
 - Results may sometimes be confusing though...
 - Example
 - `sizeof(int)`
 - `sizeof(double)`
 - `sizeof(bool*)`

Outline

- Memory
- Dynamic Memory Allocation
 - Dynamic Memory Example
- Memory Sizes of C Types
- **Ownership**

Ownership idea

- If all `malloc()`'d memory must later be `free()`'d
 - Then there must be some agreement on which function should free it
- This concept is known as "ownership"
 - Ownership is unique. An object cannot have multiple owners
- The part of the software that "owns" the memory must either:
 1. Eventually free that memory

OR

 2. Eventually transfer ownership

Ownership questions

- When memory is passed into or out of a function, two options:
 1. Ownership transfer
 2. “Borrowing” the memory
- Borrowing memory means that it can be accessed until the function returns
 - But the function won’t hold on to a pointer and try to access it later
 - Example:
 - `printf()` only ever borrows memory. It never frees the memory or tries to access that memory again during future calls to `printf()`

Ownership in our dynamic memory example

```
char* make_mutable_string(const char*);
```

- The caller takes ownership of the result
 - (This function creates memory, but is not in charge of freeing it)

```
void uppercase_string(char*);
```

- Borrows the string transiently
 - (Accesses it temporarily, but does not take ownership)

```
void print_and_destroy(char*);
```

- Takes ownership of the input string
 - (This function will free it)

Ownership is a concept

- Nothing in the compiler will enforce ownership
- No way to know if a function takes ownership or borrows without reading the documentation
- Ownership is a contract about how you promise to implement code
 - But if you follow it, it makes dynamic memory easier!

The full ownership protocol

- The owner of a heap-allocated object is responsible for deallocating it
 - No one else may do so
- Borrowers of an object may access or modify it
 - But they may not hold on to a reference to it or deallocate it
- Passing or returning a pointer *may or may not* transfer ownership
 - If so, caller must have owned it previously and now give up ownership
 - If not, caller could also be borrowing. The new function is also borrowing

Outline

- Memory
- Dynamic Memory Allocation
 - Dynamic Memory Example
- Memory Sizes of C Types
- Ownership

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

$$134_{10} - 128_{10} = 6_{10}$$

$$6_{10} - 4_{10} = 2_{10}$$

$$2_{10} - 2_{10} = 0_{10}$$

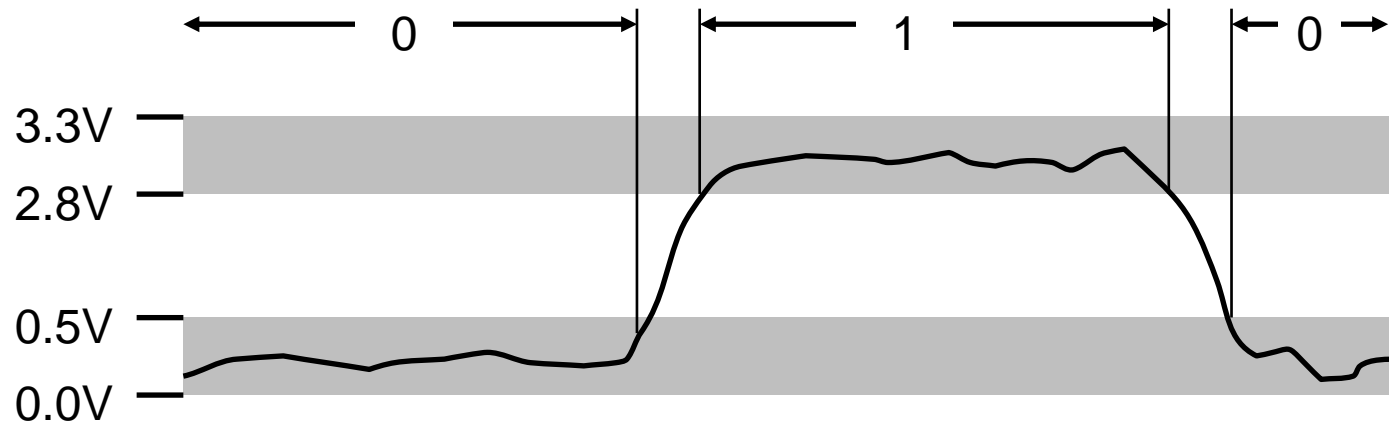
$$134_{10} = \underline{1} \times 128 + 0 \times 64 + 0 \times 32 + 0 \times 16 + 0 \times 8 + \underline{1} \times 4 + \underline{1} \times 2 + 0 \times 1$$

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

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

Why computers use Base 2

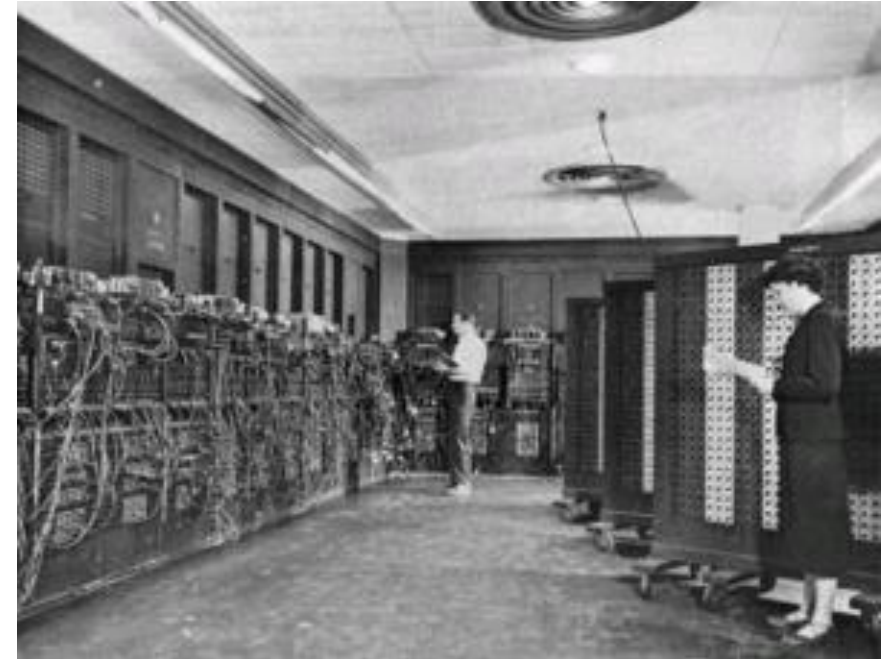
- Simple electronic implementation
 - Easy to store with bi-stable elements
 - Reliably transmitted on noisy and inaccurate wires



- 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:

- $0x4 \rightarrow 0b0100$ $0x2 \rightarrow 0b0010$ $0x42 \rightarrow 0b\ 0100\ 0010$

- 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 $255/4 \approx 240/4 \approx 60$

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