Lecture 06
Dynamic Memory

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Spring 2023

Slides adapted from:
Jesse Tov

Administrivia

- Exercise 4 due today
  - Includes malloc() which was in the chapter readings
  - But we'll also talk about it in class today

- Homework 2 due Thursday
  - Be sure to start on it ASAP. Homeworks keep getting harder!
  - Starter video available on piazza
  - Make sure you're also reading the writeup though

Today's Goals

- 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?

- Begin exploring dynamic data structures: dynamic arrays

Getting the code for today

```
cd ~/cs211/lec/
```

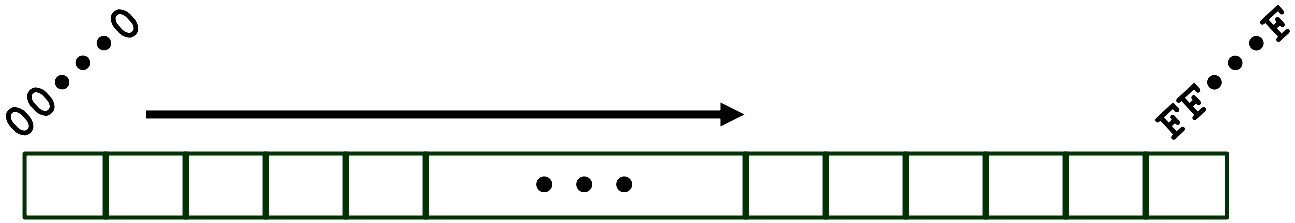
(or wherever you put stuff)

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

Outline

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

Review: What is memory conceptually?



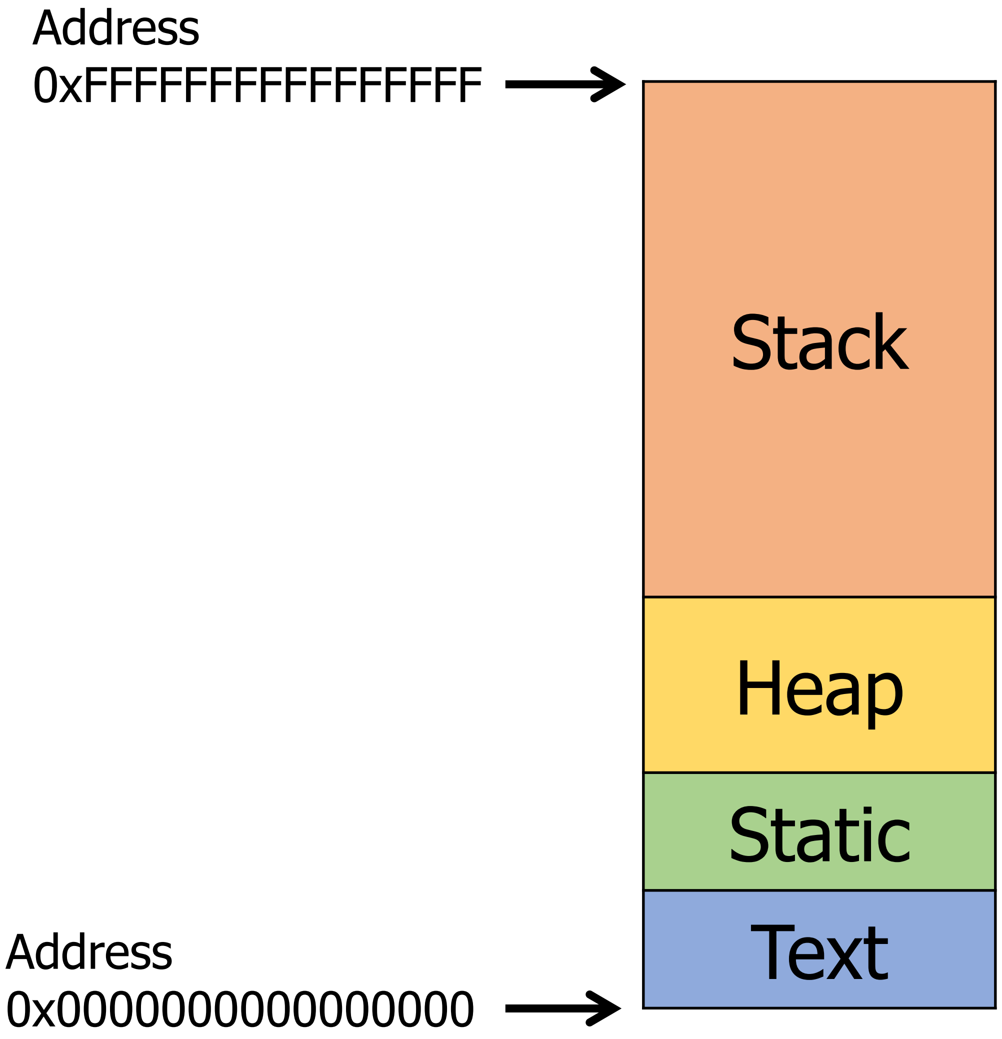
- A nearly infinite series of slots that can be used to hold data
  - Units of memory are known as bytes

  - 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

Review: 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



Review: 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"

Review: 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 (uninitialized)

- 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 stored as 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)`

- Deallocates the memory at the pointer
- 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)
  - This includes a pointer that has been modified from the one returned by malloc
  - `free(NULL)` is always fine though

- Once memory is freed, it must NEVER be used again
  - Or else… **UNDEFINED BEHAVIOR** (surprise!)
  - Definitely don't free it twice

- AddressSanitizer will helpfully crash your code in both of these cases!

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

testfunction.c

What values does this program print?

```
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;
}
```

Break + Question

testfunction.c

```
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

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

Live coding example

dynamic_string-starter.c
dynamic_string-solution.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

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

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
  - This is not a good design

- HOWEVER, if you work on a modern 64-bit computer, you can *carefully* make some assumptions
  - And we’ll talk about those assumptions
  - Note: no need to memorize these for this class

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, size_t
  - 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 (8 slots * 4 bytes/slot)

- Structs
  - Complicated! (we'll explore more in CS213)
  - At minimum, 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

  - Example
    - `sizeof(int)`
    - `sizeof(double)`
    - `sizeof(bool*)`
    - `sizeof(x)`
    - `sizeof(*vc)`

Outline

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

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*);
```

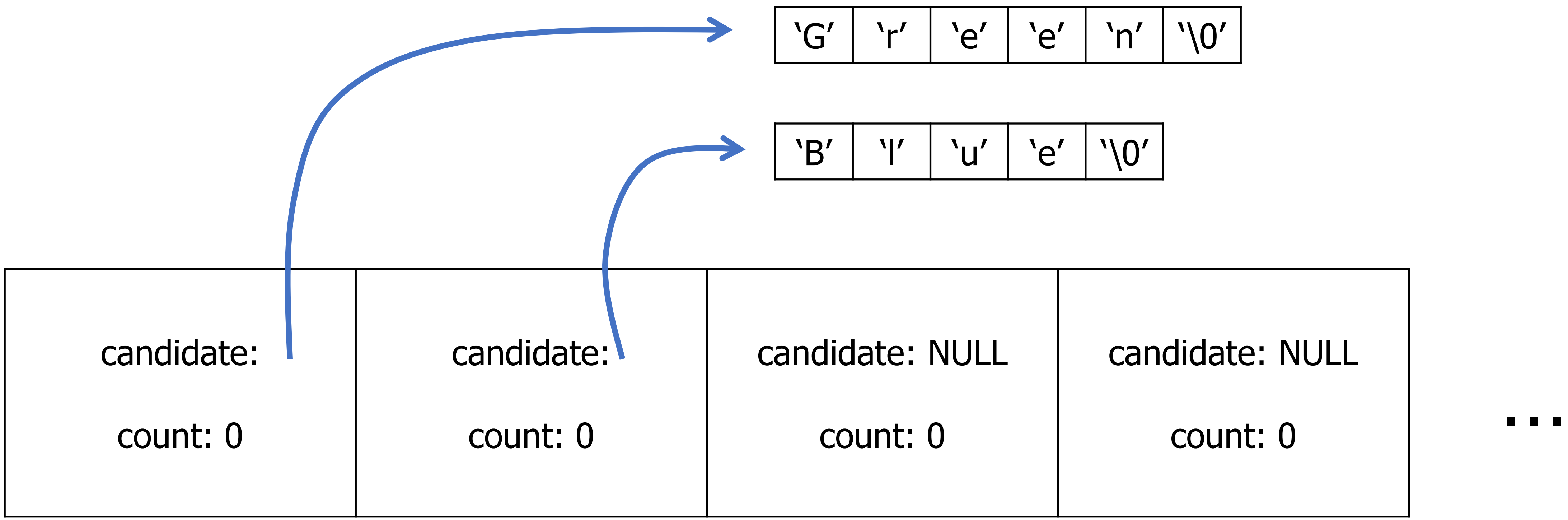
- Borrows the string transiently
  - (Accesses it temporarily, but does not take ownership)

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

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

Data structures can "own" memory

- In Homework 2, the `vc` owns the candidate name strings
  - Pointers to the memory are stored within it
  - It promises to free them when it is finished (`vc_destroy()`)



Ownership is a concept

- Bad news: 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 contract will be specified in the writeup for homeworks in CS211

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
  - Transfer: caller must have owned it previously and now give up ownership
  - No transfer: caller could also be borrowing. New function is borrowing

Break + Question

- Does this function "borrow" or "take ownership" of `message`?
- Does the caller "borrow" or "take ownership" of return result?

```
// Expects a malloc()'d string as input
// Creates a new uppercased string with malloc()
// Frees the input string
// Returns a pointer to the new string
char* make_uppercase(char* message);
```

Break + Question

- Does this function "~~borrow~~" or "**take ownership**" of `message`?
- Does the caller "~~borrow~~" or "**take ownership**" of return result?

```
// Expects a malloc()'d string as input
// Creates a new uppercased string with malloc()
// Frees the input string
// Returns a pointer to the new string
char* make_uppercase(char* message);
```

Outline

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

Dealing with dynamic input

- What if you want to read in data, but you don’t know how much data there might be?

- Arrays in C are a fixed size
- But you can `malloc()` as many times as needed
  - Request some memory
  - Use until you run out
  - Request more memory and copy existing values over

  - `realloc()` makes this simple

Example: expanding an array

```
// create array
int* array = malloc(sizeof(int) * 2);
array[0] = 5;
array[1] = 2;

// expand array
int* newarray = malloc(sizeof(int) * 4);
newarray[0] = array[0]; // copy over values
newarray[1] = array[1]; // copy over values
free(array);
array = newarray;

// use expanded array
array[2] = 1;
```

Example of dynamic memory: read_line()

```
char* read_line(void)
```

- Reads an entire line at a time from stdin
    - Can't know in advance how many bytes there will be to read
    - Keeps reading in bytes until '\n' character or end-of-file
    - Needs to request more memory until it holds the entire line
- Note: part of the 211 library, not standard C

Live coding: implement read_line()

readline-starter.c
readline-solution.c

```
char* read_line(void)
```

- Requirements
  - Read from stdin until '\n' or end-of-file (EOF)
    - Could `fread()` or just use `getchar()`
  - Allocate an array to hold the read characters
    - Make sure to end it with a '\0'
  - Returns
    - NULL pointer if EOF was reached immediately
    - Pointer to string otherwise (not including the newline character)

Realloc versus malloc

- We could just `malloc()` and copy ourselves, what does `realloc()` add?

- `realloc()` can be far more efficient
  - Doesn’t have to copy data at all if there is room in the heap to expand

- Also simpler for programmers
  - Can’t forget to free the old memory if `realloc()` does it for you

Default string size will change efficiency

- Memory efficiency
  - Pointer returned could have way more memory than characters
  - User might hold on to memory for a while before freeing
  - The less wasted memory, the less memory the program needs

- Runtime speed
  - `malloc()` and `realloc()` are slow
  - The fewer times we call them, the faster the program will run

- Need to pick a sweet spot to balance the two of these
  - Real program: starts at 80 characters, doubles size when reallocating

Does efficiency really matter though?

- If you’re writing a CS211 homework: **no**

- If you’re writing a Javascript interpreter for Firefox,
  - Which has millions of users
  - times hundreds of websites per day for each user
  - times hundreds of lines of code per website
  - and each line of code is read with `read_line()`

  - **YES**

Outline

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