# Section 8
# Stack and Heap

CS 107A, Autumn 2021
Andrew Benson (adbenson@)

# Don’t forget to start recording

- Moving your cursor and deleting around the command line
    - <CTRL+A>: moves your cursor to the beginning of the line
    - <CTRL+E>: moves your cursor to the end of the line

    - <CTRL+W>: deletes the last word behind the cursor
    - <CTRL+U>: deletes everything before the cursor
    - <CTRL+K>: deletes everything after the cursor

- 1:1s
  - I think all of these are scheduled or have occurred, ping me if you believe otherwise
- Walkthrough for assign3 posted

# Bird’s Eye View

| Day | Week 4 Wednesday | Thursday | Friday | Week 5 Monday | Tuesday | Wednesday | Thursday | Friday |
|---|---|---|---|---|---|---|---|---|
| CS 107A | | Section: Stack and Heap | | | Section: Generics | | Section: More Generics | |
| CS 107 | Lab 3: Arrays / Pointers | | Lecture: `void*`, Generics | Lecture: More Generics | | Lab 4: `void*` / Function Pointers | | Lecture: Intro x86 |
| CS 107 assignments | assign2 due, assign3 released | | | | | assign3 due, assign4 released | | Tuesday: Midterm |

# Agenda

- Structs + Struct Exercise
- Stack vs Heap vs Data Segment
- Stack/Heap Gotchas
- `malloc` and Friends
- `malloc` Exercises
- Code Exercises

Structs + Struct
Exercise

# Struct Syntax

```
// Normal declaration syntax
struct struct_name {
  int int_field;
  char* char_ptr_field;
}; // don’t forget semicolon

struct struct_name *struct_ptr_var =
    malloc(sizeof(struct struct_name));
assert(struct_ptr_var != NULL);

// Typedef declaration syntax
typedef struct {
  int int_field;
  char* char_ptr_field;
} struct_alias;

struct_alias *struct_ptr_var =
    malloc(sizeof(struct_alias));
assert(struct_ptr_var != NULL);
```

```
// Arrow syntax
// USE WHEN YOU HAVE A POINTER TO A
STRUCT
// (this is pretty common)
struct_alias *foo =
malloc(sizeof(struct_alias));

foo->int_field = 6;
foo->char_ptr_field = strdup(“hello”);
assert(foo->char_ptr_field != NULL);

// Dot syntax
// USE WHEN YOU HAVE AN ACTUAL STRUCT
VALUE
struct_alias foo;

foo.int_field = 6;
foo.char_ptr_field = strdup(“hello”);
assert(foo->char_ptr_field != NULL);
```

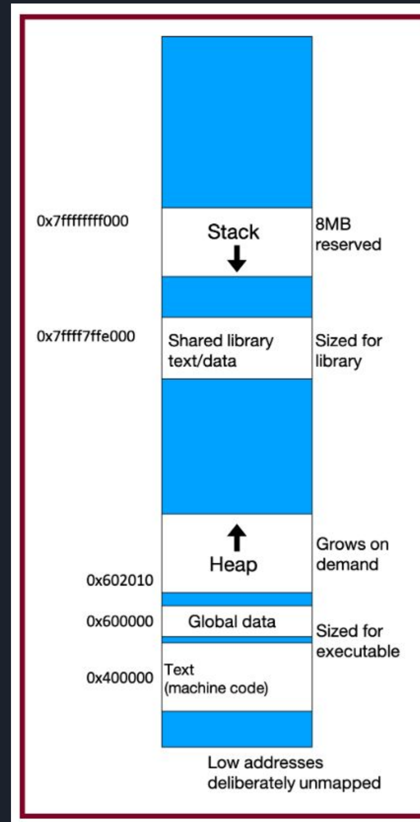
# What does it print out?

```
struct word_pair {
  char *first;
  char *second;
};

void swap_words(struct word_pair *pair) {
  char *temp = pair->first;
  pair->first = pair->second;
  pair->second = temp;
}

int main() {
  struct word_pair stanford;
  stanford.first = “Stanford”;
  stanford.second = “University”;

  swap_words(&stanford);
  printf(“%s %s\n”, stanford.first, stanford.second);
  return 0;
}
```

Based on practice problems I originally wrote for CMU 15-122 Fall 2015 Recitation 12.

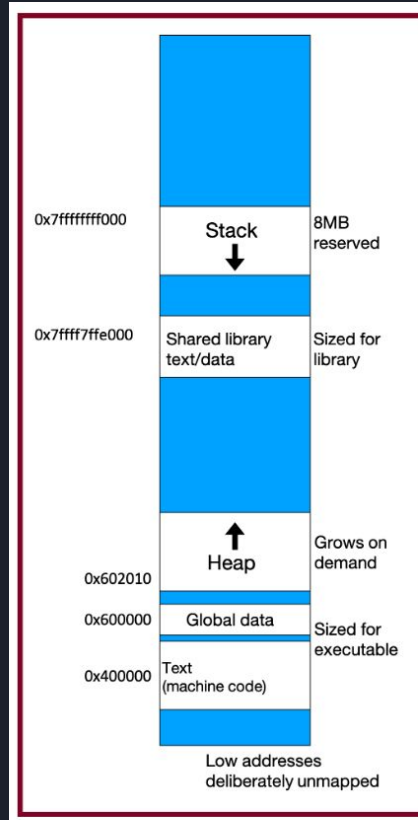
Stack vs Heap vs
Data Segment

# Memory Layout



# http://www.stackgrowsdown.com/

And naturally,
http://www.stackgrowsup.com/



| | Stack | Heap | Data Segment |
|---|---|---|---|
| What's it for? | For local variables | For values that need to last beyond this function | For literal strings hardcoded into the program |
| How long does it last? | Temporary until function returns | Persists until freed | Part of the program itself, can't be freed |
| How fast is it to allocate? | Predictably fast | Unpredictably slow, relatively | It was never allocated, so N/A? |
| Any gotchas? | Don't use a piece of stack memory after the function it's part of returns | Lots - don't double free, don't go out of bounds, don't forget to free, etc | Read-only - don't modify |

Stack / Heap
Gotchas

# Common Gotchas

```
int* add_sorta_maybe(int a, int b) {
  int x = a + b;
  return &x;
}

int main() {
  int *sum = add_sorta_maybe(5, 7);
  int *sum2 = add_sorta_maybe(3, 4);
  printf("sum: %d\n", *sum);
  printf("sum2: %d\n", *sum2);
}
```

```
int main() {
  int *arr = malloc(int);
  arr[0] = 3;
  arr[1] = 4;
  printf("sum: %d\n", arr[0] + arr[1]);
}
```

Based on practice problems I originally wrote for CMU 15-122 Fall 2015 Recitation 12.

malloc and
Friends

# malloc

- `void *malloc(size_t size);`
- Allocates you `size` bytes of memory on the heap
- Always calculate your size with `sizeof`
- In certain situations, malloc can return `NULL`
  - In CS 107, `assert(ret_value != NULL)`
  - In real life, either assert, don't worry about it, or handle that case

# `free`

- `void free(void *ptr);`
- Signals to the computer that the memory pointed to by `ptr` can be reclaimed
- During the lifetime of a program, every `malloc` should have a corresponding `free`
  - Not necessarily in the code, since loops can hide how many calls exist
- If you're concerned, wait to add calls to `free` until you pass all test cases and your code is correct

- `void *calloc(size_t count, size_t size);`
  - 1) Use `malloc` to get `count*size` bytes.
  - 2) Set every byte's value to 0.
  - 3) Return that.
- `void *realloc(void *ptr, size_t size);`
  - 1) `free(ptr);` // Or at least you can think of it that way
  - 2) `return malloc(size);`

malloc Exercises

```c
int *is_it_two(int *ptr) {
  if (*ptr == 2) {
    return ptr;
  }
  return NULL;
}

int main() {
  int *ptr = malloc(sizeof(int));
  ptr = is_it_two(ptr);
  free(ptr);
}
```

```c
void print_int(int *i) {
  printf(“%d\n”, *i);
  free(i);
}

int main() {
  int x = 6;
  print_int(&x);
  return 0;
}
```

Based on practice problems I originally wrote for CMU 15-122 Fall 2015 Recitation 12.

```
git clone /afs/ir/class/cs107a/WWW/git/section8
```