CS 521: Systems Programming

Structs and Dynamic Memory Allocation

Lecture 6

Before we dive in

- Let's talk about the `cat` lab some more
- Let's look at Project 1!
  - And two relevant code examples
  - See the `find` program for reference
- And we should talk about Wordle (because why not?)
  - Actually, something we're going to do today will help

Today's Schedule

- Structs
- Dynamic Memory Allocation

Today's Schedule

- **Structs**
- Dynamic Memory Allocation

Structs

- In C, a `struct` (structure) allows us to create **groupings** of data
  - And the elements (*members*) of a struct don't have to all be the same type, unlike arrays
- Structs are about as close as we get to classes in Java/Python
- The big distinction: they **only** represent data
  - No mixing of functions and data
  - To create functions that operate on structs, you'll pass the struct in as an argument

Defining a Struct [1/3]

Let's create a struct to contain some numbers:

```c
struct struct_name {
    int first_integer;
    int second_integer;
    float single_float;
};
```

Note the semicolon `;` at the end of the declaration

Defining a Struct [2/3]

Or, arrays can be struct members. Here, we see a couple of strings:

```c
struct user_data {
    int account_number;
    char first_name[100];
    char last_name[100];
};
```

Defining a Struct [3/3]

A struct can contain another struct, but they cannot be self-referential (contain themselves). However, a pointer to the struct type **can** be a member:

```
struct user_data {
    int account_number;
    char first_name[100];
    char last_name[100];
    struct user_preferences prefs;
    struct user_data *children; /* <-- This could be an array */
};
```

Initializing a Struct

```c
/* Creating a struct: */
struct struct_name s; /* <-- Values may be uninitialized */

/* Creating a struct and populating it: */
struct struct_name s1;
s1.first_integer = 3;
s1.second_integer = 9;
s1.single_float = 3.3f;

/* The same thing, but defined inline: */
struct struct_name s2 = { 3, 9, 3.3f };

/* Initializing everything to 0: */
struct struct_name s3 = { 0 };
```

Setting Values

As you've seen, we use "**dot notation**" to set members of a struct:

```c
struct user_data user1;
user1.account_number = 12;

/* But... this doesn't work: */
user1.first_name = "Matthew";
/* Why? */

/* ...and how can we fix it? */
```

Copying in Arrays and Strings

```c
/* For strings */
struct user_data user1;
user1.account_number = 12;
strcpy(user1.first_name, "Matthew");
printf("%s\n", user1.first_name);

/* Copying... anything! (including arrays): */
size_t arr_sz = sizeof(arr) / sizeof(*arr);
memcpy(user1.some_array, arr, arr_sz);
```

Pointers to Structs

If you have a ***pointer to a struct***, then members are accessed via "**arrow notation**":

```
void check_account(struct user_data *user1) {
    user1->account_number = 100;
    printf("%s's account number set to 100\n", user1->first_name);
}

/* Equivalent: */
(*user1).account_number = 100;
```

Basically, you must dereference the struct before accessing its members. `->` is just shorthand for this.

Declaring a struct

- The most common place to put structs is at the top of your .c file or in a header.
  - Yes, you can actually declare a struct inside a function!
  - One-time use: `struct my_struct { ... } struct_name` (defines and creates a struct named 'struct_name' in one step)
- You **can** forward declare a struct:
  - `struct my_struct;`
  - However, usage is limited: since we don't know anything about the struct members, you can't refer to them
    - (mostly helpful when declaring a pointer to the struct or functions that take the struct as a parameter...)

Struct Q&A

- **Q**: Are structs passed like our regular primitives (by value), or like arrays (essentially passed by reference)?
  - **A**: by value
- **Q**: In other words, do we make copies when we pass a struct around?
  - **A**: Yes. Including when we `return` a struct!
- **Q**: Can we have structs inside of structs?
  - **A**: Absolutely! But if the member is of the same type then it needs to be a pointer.

Bitfields [1/2]

You can explicitly set the storage size of struct members to a particular number of bits:

```c
struct settings {
   unsigned int discombobulate_thrusters : 1;
   unsigned int hyperdrive_enabled : 1;
   unsigned int anti_gravity_mode : 2;
};
```

- This can save a lot of space!
- You will most likely **only** use bitfields with `unsigned int`.

Bitfields [2/2]

- Some hardware devices use bits as on/off switches
  - Bitfields give us a way to model that in code without doing a lot of low-level bit manipulation
- Or, maybe you want to store a small number of states: if you only have say, 4 possible options, then a 2-bit field is perfect
- **NOTE**: sizeof() will ***not*** work on a bitfield.

Unions [1/2]

`union` is a close relative of the struct:

```
union my_union {
    int a;
    float b;
    struct user_data c;
}
```

- With one **HUGE** difference: they only store a single member.
- Useful for managing chunks of data that could be represented by multiple types

Unions [2/2]

```
union my_union {
    int a;
    float b;
    struct user_data c;
}
```

- Here, `a`, `b`, and `c` all have the same memory address.
- `sizeof(union my_union)` will return the size of the **largest** member (probably `c` in this case).
- Nothing stops you from doing this with pointers instead
  - Create a struct, store an `int` / `float` in the memory address
  - Unions are a well-defined, official way of achieving this

Wrapping up: Structs

- Structs can be very useful for modeling objects or groups of information
- Remember that they are copied by value, just like our primitive types
  - Consider passing large structs as “in/out args” to avoid the cost of copying during `return`
- Generally they are stored in memory as they are written, i.e., the same as if you’d just declared the members outside of a struct
  - However, the compiler is allowed to rearrange them!

Today's Schedule

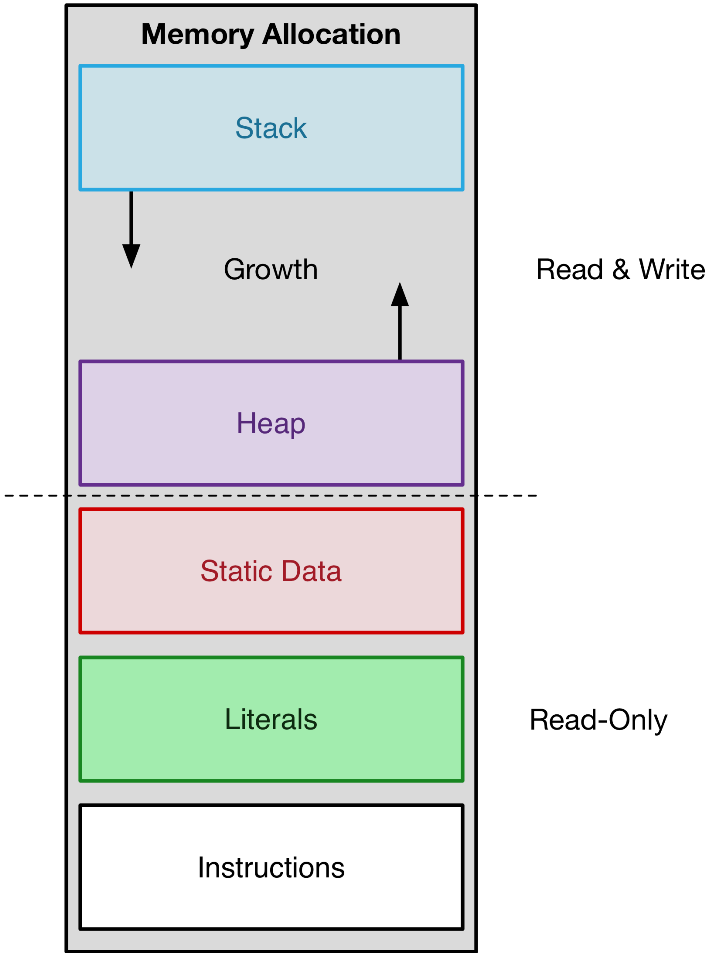
- Structs
- **Dynamic Memory Allocation**

Memory Allocation

- A running instance of a program is called a **process**
- Processes are allocated memory to store instructions, string literals, constants, and more
- At run time, there are two places memory is allocated:
  - Stack
  - Heap

Memory Layout

- **Stack**: Temporary data
  - Made up of stack frames
- **Heap**: long-lived data



The Stack

- Thus far, we've allocated everything to the stack
  - `int a = 5;`
- A good fit if we already know what data we're working with ahead of time
- If we know a user wants to enter, say, a number, we set aside some memory for them to do it
- If we don't know what data will be coming in ahead of time, then we need to place it on the **heap**

Demo: Returning Pointers on the Stack

- What happens if we have a function that returns a pointer to something that was stored on the stack?
  - ...

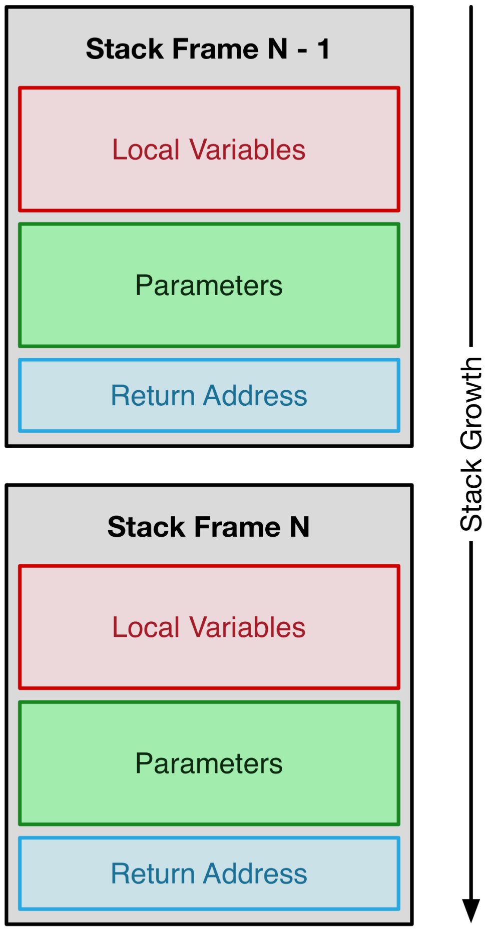
Stack Frames [1/2]

- Each function call has a stack frame
  - You may also see these called activation records
- The stack frame contains the local variables, return address, and parameters
  - In other words, the "execution environment" for each function call
- Stack frames get pushed onto the stack with each function call
  - Unchecked recursive functions can lead to stack overflow

Stack Frames [2/2]

```c
int main(int argc, char *argv[]) {
    hello(1);
    return 0;
}

int hello(int i) {
    int j = i + 1;
    printf("Hello world!\n");
    return j;
}
```



Stack Overflow

We can cause a stack overflow by making the stack grow too large.

Consider a recursive function:

```c
int foo()
{
    return foo();
}
```

Heap [1/2]

- The heap is where we **dynamically** allocate memory
- This is achieved using the `malloc()` function
- Allocating memory dynamically lets us cope with changing inputs
  - Perhaps a user wants to load a file: we can't just allocate a huge variable ahead of time and hope it fits
- How would we store a file in memory anyway? There's not exactly a "file" primitive type…

Heap [2/2]

- Use dynamic memory when:
  - You need a large block of memory
  - You want to keep a variable around for a long time
- Data that has been allocated via `malloc` is basically global: if you know where it is in memory (with a pointer), then you can manipulate it from anywhere

Allocating Memory: malloc

```
#include <stdlib.h>
void *malloc(size_t size);
```

- This sets aside a block of memory for us to use
    - We just need to give it the size
- The memory address of the new block is returned as a pointer to anything (`void *`)
- Reminder: there is no guarantee the memory set aside is zeroed out

Freeing Memory: free

```
#include <stdlib.h>
void free(void *ptr);
```

- Every `malloc()` must also have a corresponding `free()`
- Without freeing the memory, you introduce **memory leaks**
  - Imagine doing this inside an infinite loop
    - Or, maybe we don't have to imagine it…

Use After Free

```
/* What happens here? */
int *i = malloc(sizeof(int));
*i = 3;
printf("%d\n", *i);
free(i);
printf("%d\n", *i);
```

Allocate and Clear: calloc

This gives us a nice, zeroed-out memory block:

```
void *calloc(size_t nmemb, size_t size);
```

Note that `calloc` assumes you want to allocate more than one member; you can always pass in `nmemb=1`, though.

Resizing an Allocation

You can request an existing block of memory to be resized:

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

WARNING: you *must* check the return address of `realloc`, because it can relocate the memory block!

```
some_ptr = realloc(some_ptr, size);
```

Valgrind

- As you start working with dynamic memory allocation, don’t forget to watch out for memory leaks
- And invalid accesses
- Luckily, just like `gdb` can help us debug, `valgrind` helps us track down memory issues

Exercises

- Let’s:
  - dynamically allocate an `int`, `double`, and `char`
  - dynamically allocate an array
    - print its contents before initializing it
  - resize the array
  - free everything