CS 220: Introduction to Parallel Computing

Structs and Dynamic Memory

Lecture 9

Today's Agenda

- Structs
- Memory Allocation

Today's Agenda

- **Structs**
- Memory Allocation

Structs

- C structs allow us to create **groups** of data
  - Do not have to be all the same type like arrays
- These structures can contain multiple variables
- With structs, we can implement something similar to object-oriented programming found in Java or Python
  - However, rather than embedding data and methods, structs only contain data
  - Pure separation of concerns

Defining a Struct (1/2)

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

Defining a Struct (2/2)

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

Creating a Struct

```
struct account user1;
/* Or, initialize to zero: */
struct account user1 = { 0 };
```

Setting Values

Use **dot** notation:

```
struct account user1;
user1.account_number = 12;
/* Doesn't work: */
user1.first_name = "Matthew";
/* Why? */
```

Copying in Strings

```
struct account user1;
user1.account_number = 12;
strcpy(user1.first_name, "Matthew");
printf("%s\n", user1.first_name);
```

Pointers to Structs

Here, we use **arrow** notation. Why?

```
void check_account(struct account *user1) {
    user1->account_number = 100;
    printf("%s\n", user1->first_name);
}
/* Equivalent: */
(*user1).account_number = 100;
```

A Few Questions…

- **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.
- **Q**: Can we have structs inside of structs?
  - **A**: Absolutely!

Today's Agenda

- Structs
- **Memory Allocation**

Dynamic Memory Allocation

- You may have wondered why we often set up our arrays with a fixed size ahead of time
- For example, `char line[500];`
- This simplifies programming in C
- However, we often need to cope with changing requirements in our programs
  - We need dynamic memory allocation!

The Heap

- Dynamic memory is allocated on the **Heap**
- Use dynamic memory when:
  - You need a large block of memory
  - You want to keep a variable around for a long time
- Great in theory, but can be difficult in practice
- We're used to languages like Java and Python that manage memory for us
  - In C, we need to do the heavy lifting

Allocating Memory: malloc

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

- Remember the size_t type from our **sizeof** operator?
- This sets aside a block of memory for us to use
  - We just need to give it the size
- Reminder: there is no guarantee the memory set aside is zeroed out

Freeing Memory: free()

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

- Every malloc() must also have a free()
  - Without freeing the memory, you introduce **memory leaks**
  - Imagine doing this inside an infinite loop

Use after free()

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

Dynamic Memory Functions

- calloc() – clears the memory and allocates it
  - `void * calloc(size_t num, size_t size);`

- realloc() – reallocates (resizes) dynamically-allocated memory
  - `void * realloc(void *ptr, size_t new_size);`