Week 9 - Monday

CS222

Last time

- What did we talk about last time?
- Users and groups
- Time
- Lab 8

Questions?

Project 4

Quotes

On two occasions I have been asked: "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage

Structs

Structs

- A struct in C is:
  - A collection of one or more variables
  - Possibly of different types
  - Grouped together for convenient handling.
- They were called records in Pascal
- They have similarities to classes in Java
  - Except all fields are public and there are no methods
- Struct declarations are usually global
  - They are outside of **`main()`** and often in header files

Anatomy of a `struct`

```
struct name
{
    type1 member1;
    type2 member2;
    type3 member3;
    ...
};
```

Why should we bother?

- Some data is naturally grouped together
- For example, a roster of students where each student has a name, GPA, ID number
- You could keep an array of strings, **`double`** values, and **`int`** values that corresponded to each other
  - But then sorting by GPA would mean moving values in three different arrays
- Also, we'll need structs for linked lists and trees

Java examples

- In Java, a struct-like class would be used to group some data conveniently
- Examples:

A class to hold a point in space

```
public class Point
{
        private double x;
        private double y;
        //constructor
        //methods
}
```

A class to hold student data

```
public class Student
{
        private String name;
        private double GPA;
        private int ID;
        //constructor
        //methods
}
```

C examples

- The C equivalents are similar
  - Just remember to put a **semicolon** after the struct declaration
- A string can either be a **`char*`** (the memory for it is allocated elsewhere) or a **`char`** array with a maximum size
- Examples:

A struct to hold a point in space

```
struct point
{
        double x;
        double y;
};
```

A struct to hold student data

```
struct student
{
        char name[100];
        double GPA;
        int ID;
};
```

Declaring a struct variable

- Type:
  - **`struct`**
  - The name of the struct
  - The name of the identifier
- You have to put **`struct`** first!

```
struct student bob;
struct student jameel;
struct point start;
struct point end;
```

Accessing members of a struct

- Once you have a struct variable, you can access its members with dot notation (**`variable.member`**)
  - Members can be read and written

```
struct student bob;
strcpy(bob.name, "Bob Blobberwob");
bob.GPA = 3.7;
bob.ID = 100008;
printf("Bob's GPA: %f\n", bob.GPA);
```

Initializing structs

- There are no constructors for structs in C
- You can initialize each element manually:

```
struct student julio;
strcpy(julio.name, "Julio Iglesias");
julio.GPA = 3.9;
julio.ID = 100009;
```

- Or you can use braces to initialize the entire struct at once:

```
struct student julio =
{ "Julio Iglesias", 3.9, 100009 };
```

Assigning structs

- It is possible to assign one struct to another

```
struct student julio;
struct student bob;
strcpy(julio.name, "Julio Iglesias");
julio.GPA = 3.9;
julio.ID = 100009;
bob = julio;
```

- Doing so is equivalent to using **`memcpy()`** to copy the memory of **`julio`** into the memory of **`bob`**
- **`bob`** is still separate memory: it's not like copying references in Java

Putting arrays and pointers in structs

- It is perfectly legal to put arrays of values, pointers, and even other struct variables inside of a struct declaration
- If it's a pointer, you will have to point it to valid memory yourself

```
struct point
{
	double x;
	double y;
};

struct triangle
{
	struct point vertices[3];

};
```

Dangers with pointers in structs

- With a pointer in a struct, copying the struct will copy the pointer but will not make a copy of the contents
- Changing one struct could change another

```
struct person
{
        char* firstName;
        char* lastName;
};
struct person bob1;
struct person bob2;
```

```
bob1.firstName = strdup("Bob");
bob1.lastName = strdup("Newhart");
bob2 = bob1;
strcpy(bob2.lastName, "Hope");
printf("Name: %s %s\n", bob1.firstName, bob1.lastName);
//prints Bob Hope
```

Using arrays of structs

- An array of structs is common
  - Student roster
  - List of points
- Like any other array, you put the name of the type (**`struct name`**) before the variable, followed by brackets with a fixed size
- An array of structs is filled with uninitialized structs whose members are garbage

```
struct student students[100];
```

Pointers to structs

- Similarly, we can define a pointer to a struct variable
  - We can point it at an existing struct
  - We can dynamically allocate a struct to point it at
  - This is how linked lists are going to work

```
struct student bob;
struct student* studentPointer;
strcpy(bob.name, "Bob Blobberwob");
bob.GPA = 3.7;
bob.ID = 100008;
studentPointer = &bob;
(*studentPointer).GPA = 2.8;
studentPointer = (struct student*)
     malloc(sizeof(struct student));
```

Arrow notation

- As we saw on the previous slide, we have to dereference a struct pointer and then use the dot to access a member

```
struct student* studentPointer = (struct
student*) malloc(sizeof(struct student));

(*studentPointer).ID = 3030;
```

- This is cumbersome and requires parentheses
- Because this is a frequent operation, dereference + dot can be written as an arrow (**->**)

```
studentPointer->ID = 3030;
```

Passing structs to functions

- If you pass a struct directly to a function, you are passing it by value
  - A copy of its contents is made
- It is common to pass a struct by pointer to avoid copying and so that its members can be changed

```
void flip(struct point* value)
{
    double temp = value->x;
    value->x = value->y;
    value->y = temp;
}
```

Gotchas

- **Always** put a semicolon at the end of a struct declaration
- Don't put constructors or methods inside of a struct
  - C doesn't have them
- Assigning one struct to another copies the memory of one into the other
- **Pointers** to struct variables are usually passed into functions
  - Both for efficiency and so that you can change the data inside

Example

```
struct point
{
      double x;
      double y;
};
```

- Write a function that takes two **`point`** structs and returns the distance between them

Example

```
struct student
{
      char name[100];
      double GPA;
      int ID;
};
```

- Read in 100 student names, GPAs, and ID numbers
- Sort them by ID numbers
- Print out the values

Upcoming

Next time…

- More practice with structs
  - Linked lists
- **`typedef`**

Reminders

- Keep working on Project 4
  - **Due Friday**
- Keep reading K&R chapter 6