CS50 Structures and Encapsulation

Overview

At a certain point, the usual suspect data types no longer suffice for the kind of work we need to do. Rather, we need to be able to encapsulate data more broadly, allowing us to group information together, but where all of that information relates to some large entity. For example, students have names (probably represented by strings) and ages (probably represented by integers) and grade-point averages (probably represented by floating-point numbers), but none of those things matters independently--instead, all of these things come together and are part of some larger overall entity: namely, the student. Wouldn't it be nice to be able to "bundle" these things together, perhaps allowing us to abstract some of the underlying specifics? In more modern programming languages, we might do this with a so-called object, but in C we have a more basic mechanism for this: the **data structure**.

Key Terms

- data structure
- struct
- member

```
1 #define STUDENTS 3
2 string names[STUDENTS];
3 int classyears[STUDENTS];
4 float gpas[STUDENTS];
```

```
1 typedef struct
2 {
3   string name;
4   int year;
5   float gpa;
6 }
7 student;
```

Arrays versus Structs

Up until now, if you wanted to group data together, you were limited to an array. Each element in the array needed to be of the same type and we had to declare the size of the array beforehand. Say we wanted to create a group of variables related to students using arrays. Each variable would need to be its own array and if we wanted to increase or decrease the amount of students we would need to change the `#define STUDENTS` line appropriately. The strength here is that we have random access to every student so long as we know the index associated with them and we are able to iterate through each of these arrays.

There is yet another way to group data together and that is with a **struct**. Structs allow us to make new types out of existing types. Here we created a type student, that has a string, int and float associated with it. We will refer to these as **members**. In this way we can refer to a specific member of the student type via the line `student.member` depending on what member you want to access. However, the trade up here is that you cannot iterate through each field like you could in an array. In C, arrays are static; so too are the fields in your struct. You must define what attributes you want that struct to have, for example, a name or a year. The advantage that we have storing data in a struct, is that we now can group data together that is not of the same data type. Another advantage to using structs, is that we don't have to declare how many 'students' there will be. In our array implementation, we had to include a `#define` line, in structs we can have as many students as we like without having to define that number somewhere in our code.

Implementing Structs

In the lines of code above we defined a new type called 'student'. Similar to how int is a type, so too is student a type after we define it. We can now pass in variables of type 'student' into a function or put them into other structs. To create a new a new variable of type 'student', we would type a line similar to that of one declaring a variable of type int, `student s1 = {'Zamyla', 2014, 4.0}`. Now if you want to access s1's gpa you can type `s1.gpa`. If you wanted to pass s1 to a function, we again look back at how we would do so with an int. For example, `int foo(student x)`, is totally valid. To take in Zamyla's student information specifically we would write `foo(s1)`, since s1 is the variable that is associated with Zamyla's information. Similarly if your function took an argument of type int, you could pass in just the year member of s1 by using the line `function(s1.year).`