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 related information together. For example, students have names (probably represented by strings), ages (probably represented by integers), and grade-point averages (probably represented by floating-point numbers)--but none of those things matter independently. Instead, all of these things come together and are part of some larger overall entity: the student. Wouldn't it be nice to be able to "bundle" these things together, perhaps allowing us to abstract away 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

Arrays versus Structs

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

```
typedef struct
{
  string name;
  int year;
  float gpa;
}
student;
```

Up until now, if we wanted to group data together, we were limited to an array, each element in which needed to be of the same type. Furthermore, we had to declare the size of the array beforehand. To create a group of variables related to students using arrays, each variable needs to be its own array. And to increase or decrease the amount of students, we need to change the `#define STUDENTS` line accordingly. One advantage of this setup is that, so long as we know the index associated with them, we can directly access every student.

Another way to group data together is with a **struct**. Structs allow us to make new data types out of existing ones. Here we created a type student that has a string, an int, and a 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`, where "member" is the name of whichever member we want to access. A tradeoff of using structs is that we cannot iterate through each field like we can in arrays. In C, arrays are static; so too are the fields in structs. These attributes, such as a name or a year, must be defined. One of the main advantages of storing data in a struct is that we can group data of different types together. Another benefit is that we don't have to declare how many 'students' there will be. Remember that in our earlier implementation of an array, we had to include a `#define` line, but 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 – once we define it. We can now pass variables of type 'student' into functions or into other structs. To create a new a new variable of type 'student', we need to write a line similar to what we would write if we needed to declare a variable of type int: `student s1 = {'Zamyla', 2014, 4.0}`. Now, to access s1's gpa, we can type `s1.gpa`. To pass s1 to a function, let's again look back at how we would do so with ints. For example, `int foo(student x)`, is valid. Similarly, to pass in Zamyla's student information, we can write `foo(s1)`. And if the function takes an argument of type int, we could simply pass in just the year member of s1 by using the line `function(s1.year)`.