CS16, 10S, **Handout to go with H13** (intro to structs) (printable PDF)

Available online at: http://www.cs.ucsb.edu/~pconrad/cs16/10S/homework/H13/handout
The assignment is available at http://www.cs.ucsb.edu/~pconrad/cs16/10S/homework/H13

This handout—which is your reading assignment for H13—covers structs, which are not covered in the textbook until Chapter 7. We are covering them a bit earlier than the textbook coverage, because they will open up some more interesting problem solving opportunities.

However, if you want to read some additional material about structs, you may read Section 7.1 and 7.2 in the Etter text, or Section 12.1 of the online Oualline text (on-campus link, off-campus link)

Introduction to structs in C

In addition to the data types that we've seen so far—data types such as int, double, char, char *, etc.—C gives us the ability to define new data types by combining the existing ones.

For example, we can define a new data type called `struct Time` that represents a combination of hours and minutes.

```
struct Time
{
   int hours;
   int minutes;
};
```

Or, we can define a struct for a GPS coordinate that includes both latitude and longitude:

```
struct GPSCoord
{
   double lat;
   double lon;
};
```

The things inside a struct like `hours` and `minutes` are called *fields* in the struct, or *members* of the struct.

It is possible to mix data types in a struct, and to include arrays in a struct—for example:

```
struct Student
{
   char name[20];
   int permNumber;
   double gpa;
};
```

Syntax tip: notice that a struct definition has a semicolon after the final closing brace.

Better style, in this case, would be to use a #define or a const int for the length of the array, as in these two examples:

using a #define

```
#define STUDENT_NAME_LEN 20
struct Student
{
   char name[STUDENT_NAME_LEN];
   int permNumber;
   double gpa;
};
```

using a const int

```
const int STUDENT_NAME_LEN=20;
struct Student
{
   char name[STUDENT_NAME_LEN];
   int permNumber;
   double gpa;
};
```

Syntax tip: when defining a constant with a `#define`, **don't** use = and don't put a semicolon on the end. But, when using `const int`, you **do** need = and a semicolon.

Using a `#define` or `const int` declaration for the size of the array, i.e. `STUDENT_NAME_LEN` is useful when we we need to know the size of the array later in the code—for example, to check whether some name we want to store in the `name` field—from a file, or from the user's input—will fit or not. If it doesn't fit, we have to trim it down—if we just `strcpy` it into the `name` field, and it is too big, it might overflow into the `permNumber` or `gpa` fields.

Whether to use a `#define` or a `const int` is a style issue. My opinion is that either is fine as long as you are consistent—but you'll see both in other people's code, so it is helpful to know both ways.

Please turn over for more...

...continued from other side

A struct definition does NOT create a variable, or allocate any memory.

To make this point more clearly, lets us contrast this with the following declaration of the variable `x1`:

```
double x1=3.5;
```

For this declaration, the C compiler (or Ch interpreter) sets aside space—typically 64 bits, though this is machine dependent—for the variable `x1`. A new variable is created called `x1`, with an address, a type, and an initial value of 3.5.

By contrast, this declaration:

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

sets aside NO space in memory. It only defines a new type. Creating a variable of type `struct Point` is a separate step, and it looks like this:

```
struct Point p1;
```

That declares a variable `p1` of type `struct Point`—just like the variable x1 above is of type double. The fields in the variable are not initialized to any values in particular.

To initialize the variable `p1`, we can write two assignment statements.
The `dot operator` (.) is used to "get inside the struct" and get access to the fields—it follows the variable (in this case `p1`) and comes before the field name (in this case, either `x` or `y`)

The variables p1 and p2 are called *instances* of a `struct Point`

```
  p1.x = 3.5;
  p1.y = 6.0;
```

We can also declare a variable of type struct Point and give it an initial value all at the same time, like this:

```
struct Point p2 = {1.0,-1.0};
```

struct definition are like blueprints—or cookie cutters

- A blueprint helps you build a house—but you can't live in a blueprint.
  On the other hand, from one blueprint, you can build many houses that are identical, except for their location
- A cookie cutter is not a delicious treat—you can't eat a cookie cutter.
  But you can use it to create many identically shaped cookies.

So, by analogy, a struct definition does not create any place in memory that we can store something—but once we have it, we can create many variables that all have an identical structure.

There's more to learn

Additional topics having to do with structs include (a) passing them as parameters to functions (b) returning them as the result of a function (c) using assignment statements with entire structs, or members of structs (d) using structs with pointers, (e) arrays of structs (f) arrays inside structs. But save that for the next assignment—what you have on this handout is enough to get started.

We should also mention that working with structs in C is laying the foundation for one of the most important topics you'll study in the lower division CS curriculum—namely the idea of classes in C++. C++ classes are covered in the next two courses that follow on after CS24 and CS32. The idea of a C++ class builds directly on the idea of a struct in C, so it is important to get very comfortable with structs in C to be well prepared for CS24 and CS32.

If it is not yet clear why structs are useful, don't worry—there will be examples in lecture that will help to make this more clear.

End of handout that goes with H13