CS16, 10S, **Handout to go with H14** (more on structs) (printable PDF)

Available online at: http://www.cs.ucsb.edu/~pconrad/cs16/10S/homework/H14/handout
The assignment is available at http://www.cs.ucsb.edu/~pconrad/cs16/10S/homework/H14

This handout—which is your reading assignment for H14—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. This builds on the handout from H13

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)

Passing structs to functions

We can pass a struct to a function, and return a struct from a function. In this respect, structs act just like other variables.

Suppose we have a struct Point, and we want to determine its distance from the origin.

As a reminder: the distance formula tells us that the distance between two arbitary points (x_1 y_1) and (x_2, y_2) is given by:

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

But when (x_1 y_1) is the origin, i.e. (0,0), then the formulat reduces to:

$$d = \sqrt{{x_2}^2 + {y_2}^2}$$

Formula images from Wikipedia "Distance" and "Midpoint" pages, used under CC-BY-SA license

Or, renaming the variables (x_2, y_2) to simply (x, y) we have that the distance of (x, y) from the origin is given by:

$$d = \sqrt{x^2 + y^2}$$

Recall that:

- we can use `#include <math.h>` and compile with `-lm` to get access to the built-in function `sqrt` to do square root.
- we get access to members of structs by using the dot operator (review the handout from H13, if you don't recall this.)

So, here's the definition of a struct Point:

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

And here is a function that returns the distance of a struct Point p from the origin:

```
double distanceFromOrigin(struct Point p)
{
   return sqrt(p.x * p.x + p.y * p.y);
}
```

We can also **pass two structs to a function**—here's the distance formula, using structs:

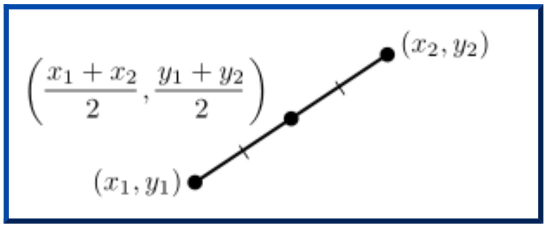
```
double distanceBetween(struct Point p1, struct Point p2)
{
   return sqrt( pow(p2.x - p1.x, 2.0) + pow(p2.y - p1.y, 2.0) );
}
```

Recall that `pow(x,y)` is `#include <math.h>` library function for x^y (where `x`, `y` and the result returned are all of type double.)

Please turn over for more...

Continued from other side

Returning a struct from a function



Recall that the midpoint of a line segment connecting two points is defined as shown in the image at right. Keeping that in mind, we can write a function that takes two struct Point instances as parameters, and returns a struct Point instance reresenting the midpoint between those two points.

Note that the return type of the function is `struct Point` and that we have to declare a variable of type called `result` to return as the result of the function.

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

```
struct Point midPoint(struct Point p1, struct Point p2)
{
   struct Point result;
   result.x = (p1.x + p2.x)/2.0;
   result.y = (p1.y + p2.y)/2.0;
   return result;
}
```

Syntax tip: the struct definition has a semicolon (`;`) after the closing brace. The function definition doesn't.

Printing the value of a struct

One difference between variables that have a struct type and regular int, double, char, char * variables is that printf can't handle struct variables directly. That is, while there are %i, %lf, %c, and %s format specifiers for int, double, char and char * (respectively) there is nothing similar for a struct variable.

So, if want to print out a struct variable, we typically must write our own function to do it. Here's an example of what that might look like for a `struct Point`

```
void printPoint(struct Point p)
{
   printf("(%lf,%lf)",p.x, p.y);
}
```

Here's an example of a main that uses this function, and the output that might go with it.
Note that we can put a `printf` on the same line before and after the `printPoint` call to make the line of code appear more natural—more as if we had done something like `printf("x=%i\n",x);` for an `int` variable called `x`.

main function	output

`int main()`
`{`
`  struct Point p1={3.0,4.2};`
`  struct Point p2={-1.0,1.0};`
 	
`  // print points`
`  printf("p1="); printPoint(p1); printf("\n");`
`  printf("p2="); printPoint(p2); printf("\n");`
 	
`  return 0;`
`}`
	`-bash-4.1$ ./printPoint`
	`p1=(3.000000,4.200000)`
	`p2=(-1.000000,1.000000)`
	`-bash-4.1$`

In future homework assignments having to do with structs, we'll cover:

- using assignment statements with entire structs, or members of structs
- using structs with pointers,
- arrays of structs
- arrays inside structs.

The examples here illustrate one of the useful aspects of structs—namely the fact that they allow us to work with more data in a single operation. This is much more convenient that having to manipulate each individual number separately. For instance, we can think in terms of a "point" rather than having to work with the x and the y as separate items.

Thus structs are a basic tool of *abstraction*, which is one of the most important ideas in Computer Science.

End of handout that goes with H14