CS113: Lecture 8

Topics:

- Defining Your Own Types
- Structures

Create your own types: typedef

Here's an example:

```
#define N 3

typedef double scalar;  /* note defs outside fns */
typedef scalar vector[N];

typedef scalar matrix[N][N];
  /* alternatively:
     typedef vector matrix[N];  */

/* add(x,y,z) adds the vectors y and z,
   placing the result in x */
void add( vector x, vector y, vector z ) {
   int i;
   for( i = 0; i < N; i++ ) {
      x[i] = y[i] + z[i];
   }
}
```

- Declaring a `scalar` works like declaring an array of size 3.
- But the *type* of a `scalar` is pointer-to-double.

A syntactic heuristic: to use `typedef`, write a statement to declare a *variable* of the type you want to redefine, giving the variable the name of the new type. Then put a `typedef` in front, and voila, you have a new type!

Structures

- The structure mechanism allows us to aggregate variables of different types

- `struct` definitions are usually placed outside of functions so that they are in scope throughout the file, as in the following example:

```
struct card_struct {
   int number;
   char suit;
}; /* note the semicolon after the definition! */

void some_function() {
   struct card_struct a, b;

   a.number = 3;
   a.suit = 'D';
   b = a;
}
```

- The "." in `a.num` is the "structure member operator", which connects the structure name and the member name. (A "member" is a variable in a structure.)

- Assignment works just as you would expect, as if there were a separate assignment for each structure member.

More on structures

The reason a semicolon follows the `struct` definition is that the definition is a statement. Also, you can declare a variable of that `struct` type using basically the same syntax:

```
struct card_struct {
   int number;
   char suit;
} my_card;
```

You're then free to use `struct card_struct` to define new structures.

You can also define "anonymous", one-time structures, as in:

```
struct {
   int number;
   char suit;
} my_card;
```

Here's a correct but totally useless declaration that defines neither a structure type nor a new variable:

```
struct {
   int number
   char suit;
};
```

Typedef with Structs

In the previous example the type of the structure was `struct card_struct`, which is clunky. We can use typedef to define an equivalent type with a more concise name:

```
struct card_struct {
   int number;
   char suit;
};

typedef struct card_struct card;

void some_function() {
   card a, b;

   a.number = 3;
   a.suit = 'D';
   b = a;
}
```

You can also define a structure type like this, without explicitly "naming" the structure, to give it a one-word type name:

```
typedef struct {
   int number;
   char suit;
} card;
```

Example: points in the plane

```
#include <math.h>

struct point_struct {
   double x;
   double y;
};

typedef struct point_struct point;

double distance( point p1, point p2 ) {
   double dx, dy, dist;
   dx = p1.x - p2.x;
   dy = p1.y - p2.y;
   dist = sqrt((dx * dx) + (dy * dy));
   return( dist );
}

void main() {
   /* here's a convenient notation for
      structure initialization: */
   point a = { 3.5, 4.5 };
   point b = { 6.5, 0.5 };
   printf( "Distance: %f\n", distance( a, b ));
}
```

What happens here?

```
typedef struct {
   char name[50];
   int age;
} employee;

void main() {
  employee tom1, tom2;
  strcpy( tom1.name, "Thomas Wolfe" );
  tom1.age = 104;
  tom2 = tom1;
  tom2.name[0] = 'G';
  printf( "Name: %s", tom1.name );
}
```

Note that since `name` is defined as an array, assignment doesn't work outside of a structure. Therefore, you'd think that assignment would cause a compilation problem here.

BUT: the code not only compiles, but the assignment provides a *deep copy* of the structure. The array is copied element-by-element, rather than just by pointers. So `tom1` stays “Thomas Wolfe”, while poor `tom2` becomes “Ghomas Wolfe”.

So you can copy entire arrays by embedding them in structures!

A Comparison Function

There's no standard way to compare structures. You can't try `tom1 == tom2` in the previous example, for example. (Or `tom1 < tom2`.)

You can always write comparison code, if you need to:

```
int compare_employees( employee e1, employee e2 ) {
  return (e1.age == e2.age) &&
     (strcmp(e1.name,e2.name) == 0);
}
```

Structures work seamlessly with functions. A structure is a type, so it can be the type of a function parameter (as here), or a return type:

```
point sum( point p1, point p2 ) {
  point psum = {p1.x + p2.x, p1.y + p2.y};
  return psum;
}
```

You can also define an array of structures:

```
int i; employee employee_list[100];
for(i=0;i<100;i++) {
  /* initialize employee i */
}
```

More on the “sizeof” operator

The `sizeof` operator takes an “object”, like a type or variable or array, and returns the size, in bytes, of the object. It’s useful for:

- Allocating memory dynamically (stay tuned)
- Determining the size of an array (or string!)

How to determine the number of elements in an array:

```
int array_size;
employee employee_list[100];
array_size = sizeof(employee_list) / sizeof(employee);
```

You could even `#define` it:

```
#define NUM_EMPLOYEES
   (sizeof (employee_list) / sizeof(employee))
/* or */
#define NUM_EMPLOYEES
   (sizeof (employee_list) / sizeof(employee_list[0]))
```

Like a Horse and Carriage...

Pointers and structures are a powerful combination, and are one that is used all the time by serious C programmers.

Suppose that `p` is a pointer to a structure that has an `int` member `x`. Using standard notation, if we want to access `x` we first have to dereference `p` using the "*" operator, and then use the "." operator to get the member, as in:

```
y = (*p).x;
```

(The parentheses are necessary because the structure member operator "." has higher precedence than the "*" operator.)

This is done so often that there's a special notation for it:

```
y = p->x;
```

Which is much cleaner (especially for more complex expressions!) and, to me, seems more intuitive. (It's an arrow, so it even looks like a "pointer"!)

Self-referential structures

What's wrong with this:

```
struct tree_node_struct {
   char *name;
   struct tree_node_struct left;
   struct tree_node_struct right;
};
```

The structure itself is recursive, and there's no way to figure out what it means! (Shouldn't this structure be infinitely big?)

You can, however, define structures recursively, by including pointers to other structures of the same type!

```
struct tree_node_struct {
   char *name;
   struct tree_node_struct *left;
   struct tree_node_struct *right;
};
```

(Draw a tree!)

When dealing with self-referential structures, it's common to think of "pointer-to-type" as the type of the structure, and deal with all the structure elements using -> notation. You can even make it explicit using a typedef:

```
/* define the type */
typedef struct tree_node_struct *node;

/* do stuff */
printf(a_node->name);
left_node = a_node->left;
```

This will be even more obviously useful after we've seen dynamic memory allocation.