Unit 28: Structures

Learning Objectives

After this unit, students should

- be aware `struct` as a compound data type in C
- know how to (i) declare a struct, (ii) assign values to the member of a struct, (iii) pass a struct into a function as value and as reference, (iv) return a struct from a function
- aware of `.` and `->` notations for accessing members of a struct
- know how to use `typedef` to define a new type

Compound Data Type

We have so far been working with numbers, characters, and strings. Not all real-world objects can be easily abstracted and represented with numbers and characters. It is useful to create our *compound data type* that represents real-world objects. Each object typically has one or more attributes, which can be of different types: A module has a code, a title, and the number of MCs; A person has a name, height, weight, and age; A phone has a model, price, and brand.

If you look back at the code that we have written, often multiple variables are related to each other and "belong together." When we pass one as an argument into a function, often we need to pass another. It is useful to group them into a compound data type as well. For instance: A 1D array and its length; A 2D array, its width, and its height; A pixel, its row, its column, and its color.

Structure in C

In C, we can define a compound data type using a structure, through the C keyword `struct`. The syntax looks like this:

```
struct matrix {
  double** array;
  long num_of_rows;
  long num_of_columns;
};
```

In the definition above, the structure is given a name, `matrix`. The structure contains three *members*. The first is a pointer to a 2D array, the other two are the number of rows and number of columns of that array.

Note that we need a semicolon `;` after the definition of a `struct`.

Here are a few more examples:

```
struct circle {
  double x_of_center;
  double y_of_center;
  double radius;
};
```

```
struct module {
  char *code;
  char *title;
  long mc;
};
```

Declaring and Initializing a Structure Variable

Let's see an example of how we can declare and initialize a structure variable:

```
struct module cs1010;
cs1010.code = "CS1010";
cs1010.title = "Programming Methodology";
cs1010.mc = 4;
```

Line 1 above declares a variable called `cs1010`. Lines 2-4 initialize each member of the structure. Note that we use `.` to access each member.

An alternative is to use a *compound literal*:

```
struct module cs1010 = {
  .code = "CS1010",
  .title = "Programming Methodology",
  .mc = 4
};
```

Using compound literal is convenient in certain cases, as uninitialized members are set to 0 (similar to initializers of arrays).

You can read and write to individual members of a structure variable just like any other variable.

```
```

```
2
   cs1010.mc = hours_spent_per_week/2.5;
   cs1010_println_long(cs1010.mc);
```

Structure as Parameters

We can pass a structure variable into a function just like a non-array variable. Unlike an array, a `struct` is called by value, i.e., it is copied onto the call stack of the function.

Hence, the code below does not actually update the MCs of CS1010:

```
1  void update_mc(struct module cs1010, long hours_spent_per_week) {
2    cs1010.mc = hours_spent_per_week/2.5;
3  }
```

To call a structure by reference, we can pass in its pointer.

```
1  void update_mc(struct module *cs1010, long hours_spent_per_week) {
2    (*cs1010).mc = hours_spent_per_week/2.5;
3  }
```

The latter example is more common idiom. Since a structure can contain multiple fields and usually occupies more bytes than a pointer, it is less efficient to copy a structure onto the call stack compared to copying its pointer.

Since this is common, C provides another syntax for accessing the member of a structure through its pointer, using the "arrow" notation"

```
1  void update_mc(struct module *cs1010, long hours_spent_per_week) {
2    cs1010->mc = hours_spent_per_week/2.5;
3  }
```

Returning a Structure

A function can return a structure. Remember in Unit 17 we said that C functions can return only one value and one way to get around this limitation is to use call by reference and the other is to use `struct`? Here is how we use `struct` to return more than one values:

```
1  struct answer {
2    long max_n;
3    long max_num_steps;
4  };
```

```
1  struct answer find_max_steps(long n) {
2    struct answer ans = {
3      .max_n = 1,
```

```
    .max_num_steps = 0
  };
  for (long i = 1; i <= n; i += 1) {
    long num_of_steps = count_num_of_steps(i);
    if (num_of_steps >= ans.max_num_steps) {
      ans.max_n = i;
      ans.max_num_steps = num_of_steps;
    }
  }
  return ans;
}
```

When a function returns a `struct`, the structure gets *copied* back to the caller.

Defining a Structure as a Type

To avoid writing the keyword 'struct `every time we declare a variable or parameter, let's introduce another keyword in C:` typedef.

C allows programmers to define their type based on the existing types. Suppose that I want a type that represents a color component (red, green, or blue) of a pixel. Each pixel can have a value between 0 to 255. So we can represent it with an `unsigned char`. I can define:

```
typedef unsigned char color_t;
```

Recall that we use the suffix `_t` convention to denote user-defined type. You have probably seen these two types elsewhere `size_t` and `time_t` in the past.

Now that we have defined `color_t` as a new type that is equivalent to `unsigned char`, we can use it just like another type:

```
color_t r;

void paint(color_t r, color_t g, color_t b);
```

Using `typedef` on `struct` frees us from typing the word `struct` every time. We can do so with either:

```
typedef struct module {
  char *code;
  char *title;
  long mc;
} module;
```

or

```
typedef struct {
  char *code;
  char *title;
  long mc;
} module;
```

In either case, we can just use `module` like any other type:

```
void update_mc(module cs1010, long hours_spent_per_week) {
  cs1010.mc = hours_spent_per_week/2.5;
}
```

If you use third-party libraries or C libraries, chances are you will come across such type. The use of `typedef` on `struct` is controversial. There is a school of thoughts that think it makes the code harder to read as it obscured the fact that a variable is a struct. Hidden costs in copying the variable onto the call stack as a parameter or returned value become non-obvious. Interested students can read Linux's Kernel Coding Style for the pros and cons of this approach.