# Chapter 9 - Structures

## Compound values

Most of the data types we have been working with represent a single value—an integer, a floating-point number, a character value. Strings are different in the sense that they are made up of smaller pieces, the characters. Thus, strings are an example of a compound type.

Depending on what we are doing, we may want to treat a compound type as a single thing (or object), or we may want to access its parts (or member variables). This ambiguity is useful. It is also useful to be able to create your own compound values. C provides a mechanism for doing that: **structures**.

## Point objects

As a simple example of a compound structure, consider the concept of a mathematical point. At one level, a point is two numbers (coordinates) that we treat collectively as a single object. In mathematical notation, points are often written in parentheses, with a comma separating the coordinates. For example, $(0, 0)$ indicates the origin, and $(x, y)$ indicates the point $x$ units to the right and $y$ units up from the origin.

A natural way to represent a point in C is with two doubles. The question, then, is how to group these two values into a compound object, or structure. The answer is a struct definition:

typedef struct {

double x;

double y;

} Point\_t;

struct definitions appear outside of any function definition, usually at the beginning of the program (after the include statements).

This definition indicates that there are two elements in this structure, named x and y. These elements are called the **members** or **fields** of a structure.

It is a common error to leave off the semi-colon at the end of a structure definition. It might seem odd to put a semi-colon after curly-brackets, but you'll get used to it.

Once you have defined the new structure, you can create variables with that type:

Point\_t blank;

blank.x = 3.0;

blank.y = 4.0;

The first line is a conventional variable declaration: blank has type Point\_t. The next two lines initialize the fields of the structure. The "dot notation" used here is called the **field selection operator** and allows to access the structure fields.

## Accessing member variables

You can read the values of an member variable using the same syntax we used to write them:

double x = blank.x;

The expression blank.x means "go to the object named blank and get the value of x." In this case we assign that value to a local variable named x. Notice that there is no conflict between the local variable named x and the member variable named x. The purpose of dot notation is to identify which variable you are referring to unambiguously.

You can use dot notation as part of any C expression, so the following are legal.

printf("%0.1f, %0.1f\n", blank.x, blank.y);

double distance = blank.x \* blank.x + blank.y \* blank.y;

The first line outputs 3, 4; the second line calculates the value 25.

## Operations on structures

Most of the operators we have been using on other types, like mathematical operators (+, %, etc.) and comparison operators (==, >, etc.), do not work on structures.

On the other hand, the assignment operator does work for structures. It can be used in two ways: to initialize the member variables of a structure or to copy the member variables from one structure to another. An initialization looks like this:

Point\_t blank = {3.0, 4.0};

The values in squiggly braces get assigned to the member variables of the structure one by one, in order. So in this case, x gets the first value and y gets the second.

Unfortunately, this syntax can be used only in an initialization, not in an assignment statement. So the following is illegal:

Point\_t blank;

blank = {3.0, 4.0}; /\* WRONG !! \*/

You might wonder why this perfectly reasonable statement should be illegal; I'm not sure, but I think the problem is that the compiler doesn't know what type the right hand side should be. You must specify the type of the assignment by adding a typecast:

Point\_t blank;

blank = (Point\_t){ 3.0, 4.0 };

It is legal to assign one structure to another. For example:

Point\_t p1 = { 3.0, 4.0 };

Point\_t p2 = p1;

printf ("%f, %f\n", p2.x, p2.y);

The output of this program is 3, 4.

## Structures as parameters

You can pass structures as parameters in the usual way. For example,

void PrintPoint (Point\_t point)

{

printf ("(%0.1f, %0.1f)\n", point.x, point.y);

}

PrintPoint() takes a point as an argument and outputs it in the standard format. If you call PrintPoint(blank), it will output (3.0, 4.0).

As a second example, we can rewrite the ComputeDistance() function from Section 1.2 so that it takes two Points as parameters instead of four doubles.

double ComputeDistance (Point\_t p1, Point\_t p2)

{

double dx = p2.x - p1.x;

double dy = p2.y - p1.y;

return sqrt (dx\*dx + dy\*dy);

}

## Call by value

When you pass a structure as an argument, remember that the argument and the parameter are not the same variable. Instead, there are two variables (one in the caller and one in the callee) that have the same value, at least initially.

If PrintPoint() happened to change one of the member variables of point, it would have no effect on blank. Of course, there is no reason for PrintPoint() to modify its parameter, so this isolation between the two functions is appropriate.

This kind of parameter-passing is called "pass by value" because it is the value of the structure (or other type) that gets passed to the function.

## Call by reference

An alternative parameter-passing mechanism that is available in C is called "pass by reference." By now we already know that C uses pointers as references. This mechanism makes it possible to pass a structure to a procedure and modify it directly.

For example, you can reflect a point around the 45-degree line by swapping the two coordinates. The most obvious (but incorrect) way to write a ReflectPoint() function is something like this:

void ReflectPoint (Point\_t point) /\* Does not work! \*/

{

double temp = point.x;

point.x = point.y;

point.y = temp;

}

This won't work, because the changes we make in ReflectPoint() will have no effect on the caller.

Instead, we have to specify that we want to pass the parameter by reference. Our function now has a struct pointer argument Point\_t \*ptr.

void ReflectPoint (Point\_t \*ptr)

{

double temp = ptr->x;

ptr->x = ptr->y;

ptr->y = temp;

}

When we are accessing the struct member variables through a pointer we can no longer use the "field-selection-operator" (.). Instead we need to use the "pointing-to" operator (->).

We pass a reference of our struct parameter by adding the "address-of" operator (&) to the structure variable when we call the function:

PrintPoint (blank);

ReflectPoint (&blank);

PrintPoint (blank);

The output of this program is as expected:

(3.0, 4.0)

(4.0, 3.0)

The parameter ptr is a reference to the structure named blank. The usual representation for a reference is a dot with an arrow that points to whatever the reference refers to.

The important thing to see in this diagram is that any changes that ReflectPoint() makes through ptr will also affect blank.

Passing structures by reference is more versatile than passing by value, because the callee can modify the structure. It is also faster, because the system does not have to copy the whole structure. On the other

## Rectangles

Now let's say that we want to create a structure to represent a rectangle. The question is, what information do I have to provide in order to specify a rectangle? To keep things simple let's assume that the rectangle will be oriented vertically or horizontally, never at an angle.

There are a few possibilities: I could specify the center of the rectangle (two coordinates) and its size (width and height), or I could specify one of the corners and the size, or I could specify two opposing corners.

The most common choice in existing programs is to specify the upper left corner of the rectangle and the size. To do that in C, we will define a structure that contains a Point\_t and two doubles.

typedef struct

{

Point\_t corner;

double width, height;

} Rectangle\_t;

Notice that one structure can contain another. In fact, this sort of thing is quite common. Of course, this means that in order to create a Rectangle\_t, we have to create a Point\_t first:

Point\_t corner = { 0.0, 0.0 };

Rectangle\_t box = { corner, 100.0, 200.0 };

This code creates a new Rectangle\_t structure and initializes the member variables. The figure shows the effect of this assignment.

Example of a rectangle with its upper left corner at (0, 0), width 100, and height 200.

We can access the width and height in the usual way:

box.width += 50.0;

printf("%f\n", box.width);

In order to access the member variables of corner, we can use a temporary variable:

Point\_t temp = box.corner;

double x = temp.x;

Alternatively, we can compose the two statements:

double x = box.corner.x;

It makes the most sense to read this statement from right to left: "Extract x from the corner of the box, and assign it to the local variable x."

While we are on the subject of composition, I should point out that you can, in fact, create the Point and the Rectangle at the same time:

Rectangle\_t box = { { 0.0, 0.0 }, 100.0, 200.0 };

The innermost squiggly braces are the coordinates of the corner point; together they make up the first of the three values that go into the new Rectangle. This statement is an example of **nested structure**.

## Structures as return types

You can write functions that return structures. For example, FindCenter() has a Rectangle\_t parameter and returns a Point\_t that contains the coordinates of the center of the rectangle:

Point\_t FindCenter (Rectangle\_t box)

{

double x = box.corner.x + box.width/2;

double y = box.corner.y + box.height/2;

Point\_t result = {x, y};

return result;

}

To call this function, we have to pass a Rectangle\_t as an argument (notice that it is being passed by value), and assign the return value to a Point\_t variable:

Rectangle\_t box = { {0.0, 0.0}, 100, 200 };

Point\_t center = FindCenter (box);

PrintPoint (center);

The output of this program is (50, 100).

We could have passed the structure as a reference to the function. In this case our function would look like this:

Point\_t FindCenter (Rectangle\_t \*box)

{

double x = box->corner.x + box->width/2;

double y = box->corner.y + box->height/2;

Point\_t result = {x, y};

return result;

}

Notice, how we had to change the access to the members of the structure, since box is now a pointer. We would also have to change the function call for FindCenter():

Point\_t center = FindCenter (&box);

## Passing other types by reference

It's not just structures that can be passed by reference. All the other types we've seen can, too. For example, to swap two integers, we could write something like:

void Swap (int \*x, int \*y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

We would call this function in the usual way:

int i = 7;

int j = 9;

printf (" i=%i, j=%i\n", i, j);

Swap (&i, &j);

printf (" i=%i, j=%i\n", i, j);

The output of this program shows that the variable values have been swapped. Draw a stack diagram for this program to convince yourself this is true. If the parameters x and y were declared as regular integer variables (without the \*s), Swap() would not work. It would modify x and y and have no effect on i and j.

When people start passing things like integers by reference, they often try to use an expression as a reference argument. For example:

int i = 7;

int j = 9;

Swap (&i, &j+1); /\* WRONG!! \*/

Presumably the programmer wanted to increase the value of j by 1 before it is passed to the function. This does not work as expected, because the expression j+1 now is interpreted a pointer value and in now pointing to a memory location beyond the variable j. It is a little tricky to figure out exactly what kinds of expressions make sense to be passed by reference. For now a good rule of thumb is that reference arguments have to be variables.

## Glossary

**structure:**

A collection of data grouped together and treated as a single object.

**member variable:**

One of the named pieces of data that make up a structure.

**reference:**

A value that indicates or refers to a variable or structure. In a state diagram, a reference appears as an arrow.

**pass by value:**

A method of parameter-passing in which the value provided as an argument is copied into the corresponding parameter, but the parameter and the argument occupy distinct locations.

**pass by reference:**

A method of parameter-passing in which the parameter is a reference to the argument variable. Changes to the parameter also affect the argument variable.

Permission is granted to copy, distribute, transmit and adapt this work un- der the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 Inter- national License: https://creativecommons.org/licenses/by-nc/4.0/. If you are interested in distributing a commercial version of this work, please contact the author(s). The LATEX source and code for this book is available from: https://github.com/mkpenta/ThinkC