v20191027-2345

Chapter S: Structures

Up to this point, we've seen five core types[1] plus `vector`s that can contain one of the others. There are many things out in the world that those core types can model by themselves, and vectors let us model large collections of values.

But sometimes—frequently, even—the real-world data we have to handle is a little more complex than singular values disconnected from anything else. For instance, when modeling locations on a discrete 2D Cartesian plane, we need to provide coordinates, often written as an (x, y) pair. These could in some circumstances be specified as two separate variables, but what if we want to keep a collection of multiple locations? What if we wanted to return a location from a function (which can only return one value at a time)? Even when we *can* use separate variables to track related data, it sometimes hinders human reading of the program if they have to keep track of separate things, and always update them together or print them together or whatever.

Once you think to look for it, this kind of multi-part data is everywhere. Books have titles and authors (and maybe some other things). Automobile brands have a make and a model. Customer info typically will have a name and an account number. Street addresses have a whole bunch of distinct pieces.

C++ lets us model this kind of data by bundling it together as a single value, with component parts that can be accessed by name; the overall value is called a *struct* (after the keyword that builds it) or an *object*, and the component parts are usually called *fields*.

S.1 Defining a new `struct` type

To define a new `struct`, we need to specify:

- the name for the overall type, and
- a type and name for each field.

Since we want to be able to use these new types pretty much everywhere, they are defined *outside* of any function, typically near the top of a `.h` file named for the type.

[1] `int`, `double`, `string`, `char`, and most recently `bool`

Consider the example described above about modeling locations on a discrete 2D Cartesian plane. In a file named `Location.h`, we could type

```
struct Location
{
  int x;
  int y;
};
```

By convention (not universal among C++ programmers, but fairly common), we begin the name of the type with a capital letter to indicate that it is distinct from types provided with the language and standard library. Syntactically, the word `struct` and the curly brackets are mandatory, as is the semicolon after the closing bracket—unlike statement blocks, which are not followed by semicolons.[2]

Notice that each line within the `struct` looks just like declaring a local variable. That's not an accident! The syntax is intentionally the same: first the type, then the name, then a semicolon. Another word for *field* is *instance variable*, after the fact that each separate *instance* of the struct can have its own separate value for that field.

Let's see how this works in practice.

Once we have defined the type, we can declare and init variables of the type:

```
Location origin = Location{0,0};
Location quad1 = Location{3,4};
```

The only new syntax here is that the initial value for the variables uses the name of the type, and then specific values for its fields inside curly brackets, a bit like we learned to do when initialising vectors.[3] The values in the curly brackets represent the values to assign to the fields of the object, in the order they were declared.

If we want to access an individual field of an object, we follow the name of the object with the name of the field, e.g. "`origin.x`". So to print out the contents of the `quad1` variable we might write a statement like

```
cout << "quad1 is at (" << quad1.x << "," << quad1.y << ").\n";
```

which in this case would print

[2]The reason for the semicolon here is largely historical and has to do with variable declaration. Most modern compilers will at least give you a good error if you forget it.

[3]Indeed, in this context we can use the exact same syntax as we did for vectors, e.g.

```
Location quad3 = { -5, -12 };
```

but by preceding the curly brackets with the type name we can use it in a few other places too, as we'll soon see.

```
quad1 is at (3,4).
```

Of course, if we only ever accessed the contents of one of these struct values by separately checking each field, they wouldn't bring much advantage. In the next section we'll see one important way to use them.

S.1.1 Review

1. Write code to define a struct to model items stocked in a grocery store, which have names and prices.
2. Write code to define a struct to model full names, which can be separated into first names and last names. (This is an oversimplification! See the end-of-chapter exercises to work towards fixing it.)

S.2 Structs as function parameters

We've long been able to write functions that take multiple parameters, so we could imagine declaring a function such as this one:

```
/** determines whether the location corresponding to the given x and
  * given y value lies on either axis */
bool onAxis (int x, int y);
```

The downside to such a design is that the function description is a bit clunky and the parameters are not intrinsically linked to each other. However, now that we have a coherent single type called `Location` for modelling locations, we can rewrite this header as

```
/** determines whether the given Location lies on either axis */
bool onAxis (Location loc);
```

The semantics of the function should be fairly clear from the description:

The expression	should be
`onAxis(Location{0,0})`	`true`
`onAxis(Location{0,7})`	`true`
`onAxis(Location{-3,0})`	`true`
`onAxis(Location{3,4})`	`false`
`onAxis(Location{5,-12})`	`false`

Note, though, that in order to specify these test cases we can make concrete values of type `Location` on the spot by making use of the name notation we used when

initialising variables. Of course, if we have the variables declared (as from the previous section) we can use those too:

The expression	should be
`onAxis(origin)`	`true`
`onAxis(quad1)`	`false`

since variables like `origin` and `quad1`, declared as being of type `Location`, can be used anywhere a `Location` object is required, including as the parameter to `onAxis`.

Defining the function requires no new syntax beyond that introduced in the previous section to access the individual fields:

```
bool onAxis (Location loc)
{
  return loc.x == 0 or loc.y == 0;
}
```

Consider further a function designed to compute the distance between locations. Each location has two coordinates, so a function that tried to do this before structs would need *four* parameters, and the opportunities for confusion would be many. But with our new struct, we can declare the function as follows:

```
/** computes the distance (Euclidean) between two given Locations */
double distance (Location first, Location second);
```

Reusing the previously-named values for convenience:

The expression	should be
`distance(origin, origin)`	`0.0`
`distance(origin, Location{0,7})`	`7.0`
`distance(origin, Location{-3,0})`	`3.0`
`distance(origin, quad1)`	`5.0`
`distance(quad1, origin)`	`5.0`
`distance(Location{-2, 3}, Location{3, -9})`	`13.0`
`distance(Location{3, 3}, Location{4, 4})`	$\sqrt{2}$ or about `1.414`

Values that line up vertically or horizontally are easy to compute by hand, and some carefully-chosen Pythagorean triples make most of the rest work out nicely, but it's important to note that answers will not in general work out to integer values!

The body of the function again makes use of the dot to access fields, but is otherwise just an implementation of the Pythagorean/Euclidean distance formula:

```
double distance (Location first, Location second)
{
```

```
  int xdiff = first.x - second.x;
  int ydiff = first.y - second.y;
  return sqrt(xdiff * xdiff + ydiff * ydiff);
}
```

S.2.1 Review

1. Based on the grocery item struct you wrote in the previous review, write a function that determines whether a given grocery item costs more than $5.
2. ... and another function that computes the total cost of a given item when a given tax rate is applied.
3. Based on the full name struct you wrote in the previous review, write a function that builds a string view of a given full name in "Last, First" format.
4. ... and another function that builds a string view of a given full name in "First Last" format.

S.3 Structs as function return values

It's not only possible to use struct values as input (parameters) to a function; we can also use them as output (return values) from a function. Indeed, this usage is arguably even more important, since we *can* always take in additional parameters, but can only return one value from a function—so if we want to in effect return two things, we have to bundle them into a single (hopefully meaningful) object.

Returning a struct object from a function is syntactically no different than returning any other type of value from a function: Declare it as the return type in the header, and write a `return` statement (or more than one) that produces a value of that type.

For instance, this function header continues the `Location` example from the previous sections:

```
/** chooses which of two given Locations is closer to (0,0). */
Location closerToOrigin (Location a, Location b);
```

Its header looks just like other functions we've written, except that the return type is `Location`. The following table gives some illustrative test cases for the function. Note that the expected results are written in a human-readable (x, y) format rather than something that might be directly typed into a program.

The expression	should be
`closerToOrigin(origin, origin)`	(0,0)
`closerToOrigin(origin, quad1)`	(0,0)
`closerToOrigin(Location{5,0}, Location{0,7})`	(5,0)
`closerToOrigin(Location{0,7}, Location{5,0})`	(5,0)
`closerToOrigin(Location{4,0}, quad1)`	(4,0)
`closerToOrigin(Location{-4,0}, quad1)`	$(-4, 0)$
`closerToOrigin(Location{-6,0}, quad1)`	(3,4)

In this case, the return statement can be simply returning one or the other of the function's parameter values. (Note also that we can make very effective use of the `distance` function from the previous section!)

```
Location closerToOrigin (Location a, Location b)
{
  constexpr Location origin = Location{0,0};
  if (distance(origin, a) < distance(origin, b))
  {
    return a;
  }
  else
  {
    return b;
  }
}
```

That need not always be the case. This next function will need to construct a brand-new `Location` value on the fly.

```
/** computes the Location that is the reflection through (0,0) of the
  * given Location. */
Location reflectionThroughOrigin (Location loc);
```

Reflecting through a point just means finding the other point that's exactly the other side of the reflection point (here, the origin). That is,

The expression	should be
`reflectionThroughOrigin(Location{5,0})`	$(-5, 0)$
`reflectionThroughOrigin(Location{0,7})`	$(-7, 0)$
`reflectionThroughOrigin(quad1)`	$(-3, -4)$
`reflectionThroughOrigin(Location{-5,-12})`	$(5, 12)$
`reflectionThroughOrigin(origin)`	$(0, 0)$

The algorithm here may now be obvious: to reflect through the origin, just negate both components. The implementation of the function thus creates a value to return

using the same `Location{...}` syntax as we used when initialising variables, except this time the values for the fields are themselves computed with mathematical expressions:

```
Location reflectionThroughOrigin (Location loc)
{
  return Location{-loc.x, -loc.y};
}
```

In general, the expressions used for the fields of the newly-built struct object can be arbitrarily complex, as long as they produce values of the appropriate type for that field (in this case, `int`, since both `x` and `y` are declared as `int` values).

It's important to note that if a function is required to produce a value, simply modifying an existing value (such as a parameter) won't be enough. If we had instead tried to do this:

```
Location reflectionThroughOrigin (Location loc)  // WON'T WORK
{
  loc.x = -loc.x;
  loc.y = -loc.y;
}
```

we would have had a few problems. First of all, it says it returns a value, but doesn't do so! Even if we redeclared the function void, however, we run into the issue that (as with the core types), calling a function makes a *copy* of the parameter values, so any modifications to the parameter variables are local to the function and are lost when the function ends. Effectively, this implementation attempt would change some internal values, and then drop everything on the floor, having no effect on anything outside the function.

S.3.1 Review

1. Still building on the earlier full name example, Write a function that produces the full name associated with a given string written in "First Last" format. Assume there is exactly one space in the parameter.

2. Write a function that produces the full name associated with a given string written in "Last, First" format. Assume there is exactly one comma and one space in the parameter.

3. Still building the earlier grocery item example, write a function that produces a grocery item just like a given one, except with a price that is 25 cents higher.

4. Write a function that makes a grocery item to model a "pack" of a given number of a given other grocery item, with a name reflecting that fact and a price that is the correct multiple of the original. For instance, if one soda costs 30 cents, a pack of 12 might be called a "pack of soda" costing $3.60.

S.4 Vectors of structs

Yet another place that we can usefully use these new struct-type values is as contents of a vector. We can make a vector of objects directly in the initialisation of a value:

```
vector<Location> collection = { Location{5,7}, Location{-1,4},
                    Location{3,10}, Location{2,-5}, Location{-4,-4} };
```

which might be appropriate for building test cases. More typically (in practice), they would be read in from a file or from the user, or produced as the result of another process (which we'll see more of later).

Once we know that such vectors exist, we can write functions to process them. For instance,

```
/** finds the location in the given vector that is furthest from (0,0)
  * @precondition locs is not empty */
Location furthestFromOrigin (vector<Location> locs);
```

The expression	should be
`furthestFromOrigin( { quad1 } )`	$(3, 4)$
`furthestFromOrigin( { origin, quad1 } )`	$(3, 4)$
`furthestFromOrigin( { Location{-6,0}, quad1, Location{0,7} } )`	$(0, 7)$
`furthestFromOrigin(collection)`	$(3, 10)$

Writing a function that processes a vector of struct objects is not substantially different from the vector-processing functions we've written before. If we need to drill down and access the fields of the objects, we can use dot notation to do so; we can also make use of other functions that process the individual struct objects, where applicable.

```
Location furthestFromOrigin (vector<Location> locs)
{
  Location result = locs[0];  //error if locs is empty!
  for (Location loc : locs)
  {
    if (closerToOrigin(result, loc))
    {
```

```
        result = loc
      }
    }
    return result;
  }
```

S.4.1 Review

1. Write a function that counts the number of full names in a given vector that match a given first name.
2. Write a function that finds the full name in a given vector that is alphabetically first (by last name).
3. Write a function that computes the total cost of a given vector of grocery items.
4. Write a function that determines whether any of the grocery items in a given vector have a given name.