Structs, enums

Custom data types

Michael Burrell

April 2, 2024

Objectives

- We will see keywords for `struct`, `enum class`
- We will see how to group data together to start building towards more complex problems

The `struct` keyword

- `struct` introduces a new **struct**ure type in C and C++
- A structure is a user-defined type that looks a look like a class
- `structs` are different in C and C++
  - In C++, they are *technically* almost exactly the same as classes
  - There is a difference in convention between them, though

Differences between classes and structs (in C++)

- Everything is public by default in structs

Differences between classes and structs (in C++)

- Everything is public by default in structs
- We rarely put code (methods, constructors, etc.) in them (though it is possible)

Differences between classes and structs (in C++)

- Everything is public by default in structs
- We rarely put code (methods, constructors, etc.) in them (though it is possible)
- They are used just for holding data

Example

```
1 struct student {
2     string name;
3     int year_of_birth;
4 };
```

- `struct` is a keyword
- Fields are introduced, one at a time
- End it with a semicolon!

Example

Class exercise

Let's write a program which reads in the x and y coordinates for two points and computes the distance between them.

Making a function

To be fancy, we'll make a function like:

```
double distance(point p1, point p2);
```

With pointers

```
1  struct x {
2      double y;
3  };
4
5  void foo(x const *z) {
6      cout << (*z).y << endl;
7  }
8
9  int main() {
10     x r;
11     r.y = 2.5;
12     foo(&r);
13     return 0;
14 }
```

Pointers to structs

- Pointers-to-structs are just like pointers to any other data (pointers to ints, pointers to doubles, etc.)
- In C and C++, we use pointers-to-structs *a lot*
- We do it enough that C and C++ give us a convenient syntax for accessing fields via a pointer
- `(*z).y` is written more conveniently as `z->y`

With pointers

```
1  struct x {
2      double y;
3  };
4
5  void foo(x const *z) {
6      cout << z->y << endl;
7  }
8
9  int main() {
10     x r;
11     r.y = 2.5;
12     foo(&r);
13     return 0;
14 }
```

One more convenience

- Initializing a struct as a local variable happens a lot and there is a convenient syntax for that, too
- Instead of `x r; r.y = 2.5;` as two separate statements, we can say `x r = { .y = 2.5 };`
- Any fields not explicitly mentioned in the initializer will be implicitly initialized to 0

Initialization example

```
1  struct x {
2      double y;
3  };
4
5  void foo(x const *);
6
7  int main()
8  {
9      x r = { .y = 2.5 };
10     foo(&r);
11     return 0;
12 }
```

Concluding structs

- `structs` are a basic mechanism in C and C++ to group together data
- We typically don't put any (or much) code in them
- `.` notation is used when we have a `struct`, and `->` is used when we have a pointer to a `struct`

Enums

- C++ inherited *simple* `enums` from C
- They are mostly just `int` constants that are defined in the global scope
- You do not have to know how to use simple `enums` (but most know they exist)

Enum example

```
1  enum animal {
2      rat, ox, tiger, rabbit, dragon, snake, horse, sheep,
           monkey, rooster, dog, pig
3  };
4
5  int main()
6  {
7      animal x = rat; // OK
8      int y = tiger; // weirdly also OK
9      int z = ox * snake; // what?
10     // ...
11 }
```

Enums as ints

- You may have picked up on a common theme in C and C++
  - Booleans don't actually exist: they're just integers
  - Characters don't actually exist: they're just integers
  - Enums don't actually exist: they're just integers
- As a systems-level language, C and C++ are designed to map everything on to general-purpose (integer) registers, if possible
- But this can lead to type problems, so a new system was developed in C++11....

Enum and int relationship

- In C (and early C++), `enum` and `int` are always interchangeable
- In C++, an `enum` is implicitly converted to `int`
  - E.g., `int x = rat;`
- In C++11, the other way around requires a cast
  - E.g., `animal x = (animal)0;`

Enum classes

- C++11 added *enum classes*, which are slightly better than *enum*s
  - Note these only exist in C++11
  - Not in previous versions of C++
  - Not in C
- The enumeration constants defined do *not* have global scope
- They *cannot* be implicitly converted to `int`s or other `enum` types

Enum class example

```
1  enum class animal {
2      rat, ox, tiger, rabbit, dragon, snake, horse, sheep,
           monkey, rooster, dog, pig
3  };
4  enum class phase { wood, fire, earth, metal, water };
5
6  int main()
7  {
8      phase x = phase::wood;
9      animal y = animal::pig;
10     x++;    // NO
11     y = phase::earth; // NO
12     x == y; // NO
13 }
```

Why use enums

- I often get asked "what would you use an enum/enum class for?"

Why use enums

- I often get asked "what would you use an enum/enum class for?"
- You should think of `enum class` and `switch` as two sides of the same coin
  - If you're ever using a `switch`, you *probably* want an `enum`
  - If you're using an `enum`, it's *probably* because you intend to use it with a `switch`
  - Enums/switches are useful for categorizing certain values

Example with enums

Class exercise

Let's make a date calculator function. E.g., how many days are between June 6 and August 30?

General guidelines

- As of C++11, there are two views of `enums`
- Use `enums` if you need compatibility with C (not applicable to this course)
- Use `enum classes` if you need type safety (we will always do that in this course)

Unions

- There is a dual form to the `struct`
- It is called a `union`
- In a `struct`, *all* of the fields mentioned exist in one object
- In a `union`, only *one* of the fields mentioned exists (at any given time)
- `unions` are used very rarely in this course, and do not have to be understood in great detail

Union example

```
1  union int_or_float {
2      long x;
3      double y;
4  };
5
6  int main()
7  {
8      int_or_float x = { .x = 3 };
9      int_or_float y = { .y = 2.5 };
10     x.y = 1.31626; // this is actually okay
11     cout << x.x << endl; // undefined behaviour
12     return 0;
13 }
```

How unions work

- Unions are implemented by overlaying all fields onto the same memory location
- When you update the value through one field, it overwrites that region of memory
  - Attempting to read from a different field to cause undefined behaviour

Why?

- Why would we ever use a `union`?

Why?

- Why would we ever use a `union`?
- It is a primitive form of polymorphism
- Unlike object-oriented polymorphism, however, the data itself does not know what kind of data it is
- We need to *tag* a union in order to use it effectively

Number example

```
1  struct number {
2      enum class number_type { integer, real } number_type;
3      union number_value {
4          int integer_value;
5          double real_value;
6      } number_value;
7  };
```

Here's a data type which represents a value which can be either an integer or a real number.

Anonymous

- `structs`, `unions` and `enums` can all be made anonymous if they are declared inside of something else
- This can avoid repetitive typing

Anonymous example

```
1 struct number {
2     enum class number_type { integer, real } number_type;
3     union {
4         int integer_value;
5         double real_value;
6     };
7 };
```

An example of an anonymous union.

Calculator example

Class exercise

Let's build a simple calculator that works with both `ints` and `doubles`.

Conclusion

- We've seen the simplest forms of user-defined data types in C++
  - `structs`
  - `unions`
  - `enums`
  - `enum classes`
- Next we will start on object-oriented programming with classes