12.1 — Introduction to compound data types

In lesson 4.1 -- Introduction to fundamental data types, we introduced the fundamental data types, which are the basic data types that C++ provides as part of the core language.

We've made much use of these fundamental types in our programs so far, especially the `int` data type. And while these fundamental types are extremely useful for straightforward uses, they don't cover our full range of needs as we begin to do more complicated things.

For example, imagine you were writing a math program to multiply two fractions. How would you represent a fraction in your program? You might use a pair of integers (one for the numerator, one for the denominator), like this:

```
#include <iostream>

int main()
{
    // Our first fraction
    int num1 {};
    int den1 {};

    // Our second fraction
    int num2 {};
    int den2 {};

    // Used to eat (remove) the slash between the numerator and denominator
    char ignore {};

    std::cout << "Enter a fraction: ";
    std::cin >> num1 >> ignore >> den1;

    std::cout << "Enter a fraction: ";
    std::cin >> num2 >> ignore >> den2;

    std::cout << "The two fractions multiplied: "
        << num1 * num2 << '/' << den1 * den2 << '\n';

    return 0;
}
```

And a run of this program:

```
Enter a fraction: 1/2
Enter a fraction: 3/4
The two fractions multiplied: 3/8
```

While this program works, it introduces a couple of challenges for us to improve upon. First, each pair of integers is only loosely linked -- outside of comments and the context of how they are used in the code, there's little to suggest that each numerator and denominator pair are related. Second, following the DRY (don't repeat yourself) principle, we should create a function to handle the user inputting a fraction (along with some error handling). However, functions can only return a single value, so how would we return the numerator and denominator back to the caller?

Now imagine another case where you're writing a program that needs to keep a list of employee IDs. How might you do so? You might try something like this:

```
int main()
{
    int id1 { 42 };
    int id2 { 57 };
    int id3 { 162 };
    // and so on
}
```

But what if you had 100 employees? First, you'd need to type in 100 variable names. And what if you needed to print them all? Or pass them to a function? We'd be in for a lot of typing. This simply doesn't scale.

Clearly fundamental types will only carry us so far.

Compound data types

Fortunately, C++ supports a second set of data types called `compound data types`. **Compound data types** (also sometimes called **composite data types**) are data types that can be constructed from fundamental data types (or other compound data types). Each compound data type has its own unique properties as well.

As we'll show in this chapter and future chapters, we can use compound data types to elegantly solve all of the challenges we presented above.

C++ supports the following compound types:

- Functions
- Arrays
- Pointer types:
    - Pointer to object
    - Pointer to function
- Pointer to member types:
    - Pointer to data member
    - Pointer to member function

- Reference types:
  - L-value references
  - R-value references
- Enumerated types:
  - Unscoped enumerations
  - Scoped enumerations
- Class types:
  - Structs
  - Classes
  - Unions

You've already been using one compound type regularly: functions. For example, consider this function:

```
void doSomething(int x, double y)
{
}
```

The type of this function is `void(int, double)`. Note that this type is composed of fundamental types, making it a compound type. Of course, functions also have their own special behaviors as well (e.g. being callable).

Because there's a lot of material to cover here, we'll do it over multiple chapters. In this chapter, we'll cover some of the more straightforward compound types, including `l-value references`, and `pointers`. Next chapter, we'll cover `unscoped enumerations`, `scoped enumerations`, and basic `structs`. Then, in the chapters beyond that, we'll introduce class types and dig into some of the more useful `array` types. This includes `std::string` (introduced in lesson 5.9 -- Introduction to std::string), which is actually a class type!

Got your game face on? Let's go!