13.9 — Struct miscellany

ALEX SEPTEMBER 11, 2023

Structs with program-defined members

In C++, structs (and classes) can have members that are other program-defined types. There are two ways to do this.

First, we can define one program-defined type (in the global scope) and then use it as a member of another program-defined type:

```
#include <iostream>

struct Employee
{
    int id {};
    int age {};
    double wage {};
};

struct Company
{
    int numberOfEmployees {};
    Employee CEO {}; // Employee is a struct within the Company struct
};

int main()
{
    Company myCompany{ 7, { 1, 32, 55000.0 } }; // Nested initialization list to initialize Employee
    std::cout << myCompany.CEO.wage << '\n'; // print the CEO's wage

    return 0;
}
```

In the above case, we've defined an `Employee` struct, and then used that as a member in a `Company` struct. When we initialize our `Company`, we can also initialize our `Employee` by using a nested initialization list. And if we want to know what the CEO's salary was, we simply use the member selection operator twice: `myCompany.CEO.wage;`

Second, types can also be nested inside other types, so if an Employee only existed as part of a Company, the Employee type could be nested inside the Company struct:

```
#include <iostream>

struct Company
{
    struct Employee // accessed via Company::Employee
    {
        int id{};
        int age{};
        double wage{};
    };

    int numberOfEmployees{};
    Employee CEO{}; // Employee is a struct within the Company struct
};

int main()
{
    Company myCompany{ 7, { 1, 32, 55000.0 } }; // Nested initialization list to initialize Employee
    std::cout << myCompany.CEO.wage << '\n'; // print the CEO's wage

    return 0;
}
```

This is more often done with classes, so we'll talk more about this in a future lesson (15.3 -- Nested types (member types) (https://www.learncpp.com/cpp-tutorial/nested-types-member-types/)).

Struct size and data structure alignment

Typically, the size of a struct is the sum of the size of all its members, but not always!

Consider the following program:

```
#include <iostream>

struct Foo
{
    short a {};
    int b {};
    double c {};
};

int main()
{
    std::cout << "The size of short is " << sizeof(short) << " bytes\n";
    std::cout << "The size of int is " << sizeof(int) << " bytes\n";
    std::cout << "The size of double is " << sizeof(double) << " bytes\n";

    std::cout << "The size of Foo is " << sizeof(Foo) << " bytes\n";

    return 0;
}
```

On the author’s machine, this printed:

```
The size of short is 2 bytes
The size of int is 4 bytes
The size of double is 8 bytes
The size of Foo is 16 bytes
```

Note that the size of `short` + `int` + `double` is 14 bytes, but the size of `Foo` is 16 bytes!

It turns out, we can only say that the size of a struct will be at least as large as the size of all the variables it contains. But it could be larger! For performance reasons, the compiler will sometimes add gaps into structures (this is called **padding**).

In the `Foo` struct above, the compiler is invisibly adding 2 bytes of padding after member `a`, making the size of the structure 16 bytes instead of 14.

For advanced readers

The reason compilers may add padding is beyond the scope of this tutorial, but readers who want to learn more can read about data structure alignment (https://en.wikipedia.org/wiki/Data_structure_alignment) on Wikipedia. This is optional reading and not required to understand structures or C++!

This can actually have a pretty significant impact on the size of the struct, as the following program demonstrates:

```
#include <iostream>

struct Foo1
{
    short a{}; // will have 2 bytes of padding after a
    int b{};
    short c{}; // will have 2 bytes of padding after c
};

struct Foo2
{
    int b{};
    short a{};
    short c{};
};

int main()
{
    std::cout << sizeof(Foo1) << '\n'; // prints 12
    std::cout << sizeof(Foo2) << '\n'; // prints 8

    return 0;
}
```

This program prints:

```
12
8
```

Note that `Foo1` and `Foo2` have the same members, the only difference being the declaration order. Yet `Foo1` is 50% larger due to the added padding.

Next lesson
13.10 Member selection with pointers and references

Back to table of contents

Previous lesson
13.8 Passing and returning structs

Leave a comment...