13.11 — Struct miscellany

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)).

Structs that are owners should have data members that are owners

In lesson 5.11 -- std::string_view (part 2), we introduced the dual concepts of owners and viewers. Owners manage their own data, and control when it is destroyed. Viewers view someone else's data, and do not control when it is altered or destroyed.

In most cases, we want our structs (and classes) to be owners of the data they contain. This provides a few useful benefits:

- The data members will be valid for as long as the struct (or class) is.
- The value of those data members won't change unexpectedly.

The easiest way to make a struct (or class) an owner is to give each data member a type that is an owner (e.g. not a viewer, pointer, or reference). If a struct or class has data members that are all owners, then the struct or class itself is automatically an owner.

If a struct (or class) has a data member that is a viewer, it is possible that the object being viewed by that member will be destroyed before the data member that is viewing it. If this happens, the struct will be left with a dangling member, and accessing that member will lead to undefined behavior.

Best practice

In most cases, we want our structs (and classes) to be owners. The easiest way to enable this is to ensure each data member has an owning type (e.g. not a viewer, pointer, or reference).

Author’s note

Practice safe structs. Don’t let your member dangle.

This is why string data members are almost always have type `std::string` (which is an owner), and not `std::string_view` (which is a viewer). The following example illustrates a case where this matters:

```
#include <iostream>
#include <string>
#include <string_view>

struct Owner
{
    std::string name{}; // std::string is an owner
};

struct Viewer
{
    std::string_view name {}; // std::string_view is a viewer
};

// getName() returns the user-entered string as a temporary std::string
// This temporary std::string will be destroyed at the end of the full expression
// containing the function call.
std::string getName()
{
    std::cout << "Enter a name: ";
    std::string name{};
    std::cin >> name;
    return name;
}

int main()
{
    Owner o { getName() };  // The return value of getName() is destroyed just after
initialization
    std::cout << "The owners name is " << o.name << '\n';  // ok

    Viewer v { getName() }; // The return value of getName() is destroyed just after
initialization
    std::cout << "The viewers name is " << v.name << '\n'; // undefined behavior

    return 0;
}
```

The `getName()` function returns the name the user entered as a temporary `std::string`. This temporary return value is destroyed at the end of the full expression in which the function is called.

In the case of `o`, this temporary `std::string` is used to initialize `o.name`. Since `o.name` is a `std::string`, `o.name` makes a copy of the temporary `std::string`. The temporary `std::string` then dies, but `o.name` is not affected since it's a copy. When we print `o.name` in the subsequent statement, it works as we expect.

In the case of `v`, this temporary `std::string` is used to initialize `v.name`. Since `v.name` is a `std::string_view`, `v.name` is just a view of the temporary `std::string`, not a copy. The temporary `std::string_view` then dies, leaving `v.name` dangling. When we print `v.name` in the subsequent statement, we get undefined behavior.

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 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.

Tip

You can minimize padding by defining your members in decreasing order of size.

The C++ compiler is not allowed to reorder members, so this has to be done manually.