15.3 — Nested types (member types)

learncpp.com/cpp-tutorial/nested-types-member-types/

Consider the following short program:

```
#include <iostream>

enum class FruitType
{
        apple,
        banana,
        cherry
};

class Fruit
{
private:
        FruitType m_type { };
        int m_percentageEaten { 0 };

public:
        Fruit(FruitType type) :
                m_type { type }
        {
        }

        FruitType getType() { return m_type; }
        int getPercentageEaten() { return m_percentageEaten; }

        bool isCherry() { return m_type == FruitType::cherry; }

};

int main()
{
        Fruit apple { FruitType::apple };

        if (apple.getType() == FruitType::apple)
                std::cout << "I am an apple";
        else
                std::cout << "I am not an apple";

        return 0;
}
```

There's nothing wrong with this program. But because `enum class FruitType` is meant to be used in conjunction with the `Fruit` class, having it exist independently of the class leaves us to infer how they are connected.

Nested types (member types)

So far, we’ve seen class types with two different kinds of members: data members and member functions. Our `Fruit` class in the example above has both of these.

Class types support another kind of member: **nested types** (also called **member types**). To create a nested type, you simply define the type inside the class, under the appropriate access specifier.

Here’s the same program as above, rewritten to use a nested type defined inside the `Fruit` class:

```
#include <iostream>

class Fruit
{
public:
	// FruitType has been moved inside the class, under the public access
specifier
	// We've also renamed it Type and made it an enum rather than an enum class
	enum Type
	{
		apple,
		banana,
		cherry
	};

private:
	Type m_type {};
	int m_percentageEaten { 0 };

public:
	Fruit(Type type) :
		m_type { type }
	{
	}

	Type getType() { return m_type;  }
	int getPercentageEaten() { return m_percentageEaten;  }

	bool isCherry() { return m_type == cherry; } // Inside members of Fruit, we
no longer need to prefix enumerators with FruitType::
};

int main()
{
	// Note: Outside the class, we access the enumerators via the Fruit:: prefix
now
	Fruit apple { Fruit::apple };

	if (apple.getType() == Fruit::apple)
		std::cout << "I am an apple";
	else
		std::cout << "I am not an apple";

	return 0;
}
```

There are a few things worth pointing out here.

First, note that `FruitType` is now defined inside the class, where it has been renamed `Type` for reasons that we will discuss shortly.

Second, nested type `Type` has been defined at the top of the class. Nested type names must be fully defined before they can be used, so they are usually defined first.

Best practice

Define any nested types at the top of your class type.

Third, nested types follow normal access rules. `Type` is defined under the `public` access specifier, so that the type name and enumerators can be directly accessed by the public.

Fourth, class types act as a scope region for names declared within, just as namespaces do. Therefore the fully qualified name of `Type` is `Fruit::Type`, and the fully qualified name of the `apple` enumerator is `Fruit::apple`.

Within the members of the class, we do not need to use the fully qualified name. For example, in member function `isCherry()` we access the `cherry` enumerator without the `Fruit::` scope qualifier.

Outside the class, we must use the fully qualified name (e.g. `Fruit::apple`). We renamed `FruitType` to `Type` so we can access it as `Fruit::Type` (rather than the more redundant `Fruit::FruitType`).

Finally, we changed our enumerated type from scoped to unscoped. Since the class itself is now acting as a scope region, it's somewhat redundant to use a scoped enumerator as well. Changing to an unscoped enum means we can access enumerators as `Fruit::apple` rather than the longer `Fruit::Type::apple` we'd have to use if the enumerator were scoped.

Nested typedefs and type aliases

Class types can also contain nested typedefs or type aliases:

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

class Employee
{
public:
    using IDType = int;

private:
    std::string m_name{};
    IDType m_id{};
    double m_wage{};

public:
    Employee(std::string_view name, IDType id, double wage)
        : m_name { name }
        , m_id { id }
        , m_wage { wage }
    {
    }

    const std::string& getName() { return m_name; }
    IDType getId() { return m_id; } // can use unqualified name within class
};

int main()
{
    Employee john { "John", 1, 45000 };
    Employee::IDType id { john.getId() }; // must use fully qualified name outside
class

    std::cout << john.getName() << " has id: " << id << '\n';

    return 0;
}
```

This prints:

```
John has id: 1
```

Note that inside the class we can just use `IDType`, but outside the class we must use the fully qualified name `Employee::IDType`.

We discuss the benefits of type aliases in lesson 10.7 -- Typedefs and type aliases, and they serve the same purpose here. It is very common for classes in the C++ standard library to make use of nested typedefs. As of the time of writing, `std::string` defines ten nested typedefs!

Nested classes and access to outer class members

It is fairly uncommon for classes to have other classes as a nested type, but it is possible. In C++, a nested class does not have access to the `this` pointer of the outer (containing) class, so nested classes can not directly access the members of the outer class. This is because a nested class can be instantiated independently of the outer class (and in such a case, there would be no outer class members to access!)

However, because nested classes are members of the outer class, they can access any private members of the outer class that are in scope.

Let’s illustrate with an example:

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

class Employee
{
public:
    using IDType = int;

    class Printer
    {
    public:
        void print(const Employee& e) const
        {
            // Printer can't access Employee's `this` pointer
            // so we can't print m_name and m_id directly
            // Instead, we have to pass in an Employee object to use
            // Because Printer is a member of Employee,
            // we can access private members e.m_name and e.m_id directly
            std::cout << e.m_name << " has id: " << e.m_id << '\n';
        }
    };

private:
    std::string m_name{};
    IDType m_id{};
    double m_wage{};

public:
    Employee(std::string_view name, IDType id, double wage)
        : m_name{ name }
        , m_id{ id }
        , m_wage{ wage }
    {
    }

    // removed the access functions in this example (since they aren't used)
};

int main()
{
    const Employee john{ "John", 1, 45000 };
    const Employee::Printer p{}; // instantiate an object of the inner class
    p.print(john);

    return 0;
}
```

This prints:

```
John has id: 1
```

There is one case where nested classes are more commonly used. In the standard library, most iterator classes are implemented as nested classes of the container they are designed to iterate over. For example, `std::string::iterator` is implemented as a nested class of `std::string`. We'll cover iterators in a future chapter.

Nested types can't be forward declared

There is one other limitation of nested types that is worth mentioning -- nested types can't be forward declared. This limitation may be lifted in a future version of C++.