9.3 — Enum classes

ALEX JUNE 17, 2021

Although enumerated types are distinct types in C++, they are not type safe, and in some cases will allow you to do things that don’t make sense. Consider the following case:

```
#include <iostream>

int main()
{
    enum Color
    {
        color_red, // color_red is placed in the same scope as Color
        color_blue
    };

    enum Fruit
    {
        fruit_banana, // fruit_banana is placed in the same scope as Fruit
        fruit_apple
    };

    Color color{ color_red }; // Color and color_red can be accessed in the same scope (no prefix needed)
    Fruit fruit{ fruit_banana }; // Fruit and fruit_banana can be accessed in the same scope (no prefix needed)

    if (color == fruit) // The compiler will compare a and b as integers
        std::cout << "color and fruit are equal\n"; // and find they are equal!
    else
        std::cout << "color and fruit are not equal\n";

    return 0;
}
```

When C++ compares color and fruit, it implicitly converts color and fruit to integers, and compares the integers. Since color and fruit have both been set to enumerators that evaluate to value 0, this means that in the above example, color will equal fruit. This is definitely not as desired since color and fruit are from different enumerations and are not intended to be comparable! With standard enumerators, there’s no way to prevent comparing enumerators from different enumerations.

C++11 defines a new concept, the enum class (also called a scoped enumeration), which makes enumerations both strongly typed and strongly scoped. To make an enum class, we use the keyword class after the enum keyword. Here’s an example:

```cpp
#include <iostream>

int main()
{
    enum class Color // "enum class" defines this as a scoped enumeration instead of a standard
enumeration
    {
        red, // red is inside the scope of Color
        blue
    };

    enum class Fruit
    {
        banana, // banana is inside the scope of Fruit
        apple
    };

    Color color{ Color::red }; // note: red is not directly accessible any more, we have to use
Color::red
    Fruit fruit{ Fruit::banana }; // note: banana is not directly accessible any more, we have to use
Fruit::banana

    if (color == fruit) // compile error here, as the compiler doesn't know how to compare different
types Color and Fruit
        std::cout << "color and fruit are equal\n";
    else
        std::cout << "color and fruit are not equal\n";

    return 0;
}
```

With normal enumerations, enumerators are placed in the same scope as the enumeration itself, so you can typically access enumerators directly (e.g. red). However, with enum classes, the strong scoping rules mean that all enumerators are considered part of the enumeration, so you have to use a scope qualifier to access the enumerator (e.g. Color::red). This helps keep name pollution and the potential for name conflicts down.

Because the enumerators are part of the enum class, there's no need to prefix the enumerator names (e.g. it's okay to name them "red" instead of "color_red", since Color::color_red is redundant).

The strong typing rules means that each enum class is considered a unique type. This means that the compiler will *not* implicitly compare enumerators from different enumerations. If you try to do so, the compiler will throw an error, as shown in the example above.

However, note that you can still compare enumerators from within the same enum class (since they are of the same type):

```cpp
#include <iostream>

int main()
{
    enum class Color
    {
        red,
        blue
    };

    Color color{ Color::red };

    if (color == Color::red) // this is okay
        std::cout << "The color is red!\n";
    else if (color == Color::blue)
        std::cout << "The color is blue!\n";

    return 0;
}
```

With enum classes, the compiler will no longer implicitly convert enumerator values to integers. This is mostly a good thing. However, there are occasionally cases where it is useful to be able to do so. In these cases, you can explicitly convert an enum class enumerator to an integer by using a static_cast to int:

```cpp
#include <iostream>

int main()
{
    enum class Color
    {
        red,
        blue
    };

    Color color{ Color::blue };

    std::cout << color; // won't work, because there's no implicit conversion to int
    std::cout << static_cast<int>(color); // will print 1

    return 0;
}
```

There's little reason to use normal enumerated types instead of enum classes.

Note that the class keyword, along with the static keyword, is one of the most overloaded keywords in the C++ language, and can have different meanings depending on context. Although enum classes use the class keyword, they aren't considered "classes" in the traditional C++ sense. We'll cover actual classes later.

Also, just in case you ever run into it, “enum struct” is equivalent to “enum class”. But this usage is not recommended and is not commonly used.

Next lesson
9.4 Structs

Back to table of contents

Previous lesson
9.2 Enumerated types

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X