13.6 — Scoped enumerations (enum classes)

learncpp.com/cpp-tutorial/scoped-enumerations-enum-classes/

Although unscoped enumerations 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
    {
        red,
        blue,
    };

    enum Fruit
    {
        banana,
        apple,
    };

    Color color { red };
    Fruit fruit { banana };

    if (color == fruit) // The compiler will compare color and fruit 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;
}
```

This prints:

```
color and fruit are equal
```

When `color` and `fruit` are compared, the compiler will look to see if it knows how to compare a `Color` and a `Fruit`. It doesn't. Next, it will try converting `Color` and/or `Fruit` to integers to see if it can find a match. Eventually the compiler will determine that if it converts both to integers, it can do the comparison. Since `color` and `fruit` are both set to enumerators that convert to integer value `0`, `color` will equal `fruit`.

This doesn't make sense semantically since `color` and `fruit` are from different enumerations and are not intended to be comparable. With standard enumerators, there's no easy way to prevent this.

Because of such challenges, as well as the namespace pollution problem (unscoped enumerations defined in the global scope put their enumerators in the global namespace), the C++ designers determined that a cleaner solution for enumerations would be of use.

Scoped enumerations

That solution is the **scoped enumeration** (often called an **enum class** in C++ for reasons that will become obvious shortly).

Scoped enumerations work similarly to unscoped enumerations (13.2 -- Unscoped enumerations), but have two primary differences: They won't implicitly convert to integers, and the enumerators are *only* placed into the scope region of the enumeration (not into the scope region where the enumeration is defined).

To make a scoped enumeration, we use the keywords `enum class`. The rest of the scoped enumeration definition is the same as an unscoped enumeration definition. Here's an example:

```
#include <iostream>
int main()
{
    enum class Color // "enum class" defines this as a scoped enumeration rather than
an unscoped enumeration
    {
        red, // red is considered part of Color's scope region
        blue,
    };

    enum class Fruit
    {
        banana, // banana is considered part of Fruit's scope region
        apple,
    };

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

    if (color == fruit) // compile error: 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;
}
```

This program produces a compile error on line 19, since the scoped enumeration won't convert to any type that can be compared with another type.

As an aside…

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 scoped enumerations use the `class` keyword, they aren't considered to be a "class type" (which is reserved for structs, classes, and unions).

`enum struct` also works in this context, and behaves identically to `enum class`. However, use of `enum struct` is non-idiomatic, so avoid its use.

Scoped enumerations define their own scope regions

Unlike unscoped enumerations, which place their enumerators in the same scope as the enumeration itself, scoped enumerations place their enumerators *only* in the scope region of the enumeration. In other words, scoped enumerations act like a namespace for their enumerators. This built-in namespacing helps reduce global namespace pollution and the potential for name conflicts when scoped enumerations are used in the global scope.

To access a scoped enumerator, we do so just as if it was in a namespace having the same name as the scoped enumeration:

```
#include <iostream>

int main()
{
    enum class Color // "enum class" defines this as a scoped enum rather than an unscoped enum
    {
        red, // red is considered part of Color's scope region
        blue,
    };

    std::cout << red << '\n';        // compile error: red not defined in this scope region
    std::cout << Color::red << '\n'; // compile error: std::cout doesn't know how to print this (will not implicitly convert to int)

    Color color { Color::blue }; // okay

    return 0;
}
```

Because scoped enumerations offer their own implicit namespacing for enumerators, there's no need to put scoped enumerations inside another scope region (such as a namespace), unless there's some other compelling reason to do so, as it would be redundant.

Scoped enumerations don’t implicitly convert to integers

Unlike non-scoped enumerators, scoped enumerators won’t implicitly convert to integers. In most cases, this is a good thing because it rarely makes sense to do so, and it helps prevent semantic errors, such as comparing enumerators from different enumerations, or expressions such as `red + 5`.

Note that you can still compare enumerators from within the same scoped enumeration (since they are of the same type):

```
#include <iostream>
int main()
{
    enum class Color
    {
        red,
        blue,
    };

    Color shirt { Color::red };

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

    return 0;
}
```

There are occasionally cases where it is useful to be able to treat a scoped enumerator as an integral value. In these cases, you can explicitly convert a scoped enumerator to an integer by using a `static_cast`. A better choice in C++23 is to use `std::to_underlying()` (defined in the <utility> header), which converts an enumerator to a value of the underlying type of the enumeration.

```
#include <iostream>
#include <utility> // for std::to_underlying() (C++23)

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

    Color color { Color::blue };

    std::cout << color << '\n'; // won't work, because there's no implicit conversion
to int
    std::cout << static_cast<int>(color) << '\n';   // explicit conversion to int,
will print 1
    std::cout << std::to_underlying(color) << '\n'; // convert to underlying type,
will print 1 (C++23)

    return 0;
}
```

Conversely, you can also `static_cast` an integer to a scoped enumerator, which can be useful when doing input from users:

```
#include <iostream>

int main()
{
    enum class Pet
    {
        cat, // assigned 0
        dog, // assigned 1
        pig, // assigned 2
        whale, // assigned 3
    };

    std::cout << "Enter a pet (0=cat, 1=dog, 2=pig, 3=whale): ";

    int input{};
    std::cin >> input; // input an integer

    Pet pet{ static_cast<Pet>(input) }; // static_cast our integer to a Pet

    return 0;
}
```

As of C++17, you can list initialize a scoped enumeration using an integral value without the static_cast (and unlike an unscoped enumeration, you don’t need to specify a base).

Best practice

Favor scoped enumerations over unscoped enumerations unless there's a compelling reason to do otherwise.

Despite the benefits that scoped enumerations offer, unscoped enumerations are still commonly used in C++ because there are situations where we desire the implicit conversion to int (doing lots of static_casting get annoying) and we don't need the extra namespacing.

Easing the conversion of scoped enumerators to integers (advanced)

Scoped enumerations are great, but the lack of implicit conversion to integers can sometimes be a pain point. If we need to convert a scoped enumeration to integers often (e.g. cases where we want to use scoped enumerators as array indices), having to use static_cast every time we want a conversion can clutter our code significantly.

If you find yourself in the situation where it would be useful to make conversion of scoped enumerators to integers easier, a useful hack is to overload the unary `operator+` to perform this conversion. We haven't explained how this works yet, so consider it magic for now:

```
#include <iostream>
#include <type_traits> // for std::underlying_type_t

enum class Animals
{
    chicken, // 0
    dog, // 1
    cat, // 2
    elephant, // 3
    duck, // 4
    snake, // 5

    maxAnimals,
};

// Overload the unary + operator to convert Animals to the underlying type
// adapted from https://stackoverflow.com/a/42198760, thanks to Pixelchemist for the
idea
constexpr auto operator+(Animals a) noexcept
{
    return static_cast<std::underlying_type_t<Animals>>(a);
}

int main()
{
    std::cout << +Animals::elephant << '\n'; // convert Animals::elephant to an
integer using unary operator+

    return 0;
}
```

This prints:

```
3
```

This method prevents unintended implicit conversions to an integral type, but provides a convenient way to explicitly request such conversions as needed.

`using enum` statements C++20

Introduced in C++20, a `using enum` statement imports all of the enumerators from an enum into the current scope. When used with an enum class type, this allows us to access the enum class enumerators without having to prefix each with the name of the enum class.

This can be useful in cases where we would otherwise have many identical, repeated prefixes, such as within a switch statement:

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

enum class Color
{
    black,
    red,
    blue,
};

constexpr std::string_view getColor(Color color)
{
    using enum Color; // bring all Color enumerators into current scope (C++20)
    // We can now access the enumerators of Color without using a Color:: prefix

    switch (color)
    {
    case black: return "black"; // note: black instead of Color::black
    case red:   return "red";
    case blue:  return "blue";
    default:    return "???";
    }
}

int main()
{
    Color shirt{ Color::blue };

    std::cout << "Your shirt is " << getColor(shirt) << '\n';

    return 0;
}
```

In the above example, `Color` is an enum class, so we normally would access the enumerators using a fully qualified name (e.g. `Color::blue`). However, within function `getColor()`, we've added the statement `using enum Color;`, which allows us to access those enumerators without the `Color::` prefix.

This saves us from having multiple, redundant, obvious prefixes inside the switch statement.

Quiz time

Question #1

Define an enum class named Animal that contains the following animals: pig, chicken, goat, cat, dog, duck. Write a function named getAnimalName() that takes an Animal parameter and uses a switch statement to return the name for that animal as a std::string_view (or std::string if you're using C++14). Write another function named printNumberOfLegs() that

uses a switch statement to print the number of legs each animal walks on. Make sure both functions have a default case that prints an error message. Call printNumberOfLegs() from main() with a cat and a chicken. Your output should look like this:

```
A cat has 4 legs.
A chicken has 2 legs.
```

Show Solution