13.2 — Unscoped enumerations

learncpp.com/cpp-tutorial/unscoped-enumerations/

C++ contains many useful fundamental and compound data types (which we introduced in lessons 4.1 -- Introduction to fundamental data types and 12.1 -- Introduction to compound data types). But these types aren't always sufficient for the kinds of things we want to do.

For example, let's say you're writing a program that needs to keep track of whether an apple is red, yellow, or green, or what color a shirt is (from a preset list of colors). If only fundamental types were available, how might you do this?

You might store the color as an integer value, using some kind of implicit mapping (0 = red , 1 = green, 2 = blue):

```
int main()
{
    int appleColor{ 0 }; // my apple is red
    int shirtColor{ 1 }; // my shirt is green

    return 0;
}
```

But this isn't at all intuitive, and we've already discussed why magic numbers are bad (5.2 -- Literals). We can get rid of the magic numbers by using symbolic constants:

```
constexpr int red{ 0 };
constexpr int green{ 1 };
constexpr int blue{ 2 };

int main()
{
    int appleColor{ red };
    int shirtColor{ green };

    return 0;
}
```

While this is a bit better for reading, the programmer is still left to deduce that `appleColor` and `shirtColor` (which are of type `int`) are meant to hold one of the values defined in the set of color symbolic constants (which are likely defined elsewhere, probably in a separate file).

We can make this program a little more clear by using a type alias:

```
using Color = int; // define a type alias named Color

// The following color values should be used for a Color
constexpr Color red{ 0 };
constexpr Color green{ 1 };
constexpr Color blue{ 2 };

int main()
{
    Color appleColor{ red };
    Color shirtColor{ green };

    return 0;
}
```

We're getting closer. Someone reading this code still has to understand that these color symbolic constants are meant to be used with variables of type `Color`, but at least the type has a unique name now so someone searching for `Color` would be able to find the set of associated symbolic constants.

However, because `Color` is just an alias for an `int`, we still have the problem that nothing enforces proper usage of these color symbolic constants. We can still do something like this:

```
Color eyeColor{ 8 }; // syntactically valid, semantically meaningless
```

Also, if we debug any of these variables in our debugger, we'll only see the integer value of the color (e.g. `0`), not the symbolic meaning (`red`), which can make it harder to tell if our program is correct.

Fortunately, we can do better.

As inspiration, consider the `bool` type. What makes `bool` particularly interesting is that it only has two defined values: `true` and `false`. We can use `true` or `false` directly (as literals), or we can instantiate a `bool` object and have it hold either one of those values. Additionally, the compiler is able to differentiate `bool` from other types. This means we can overload functions, and customize how those functions behave when passed a `bool` value.

If we had the ability to define our own custom types, where *we* could define the set of named values associated with that type, then we would have the perfect tool to elegantly solve the challenge above…

Enumerations

An **enumeration** (also called an **enumerated type** or an **enum**) is a compound data type whose values are restricted to a set of named symbolic constants (called **enumerators**).

C++ supports two kinds of enumerations: unscoped enumerations (which we'll cover now) and scoped enumerations (which we'll cover later in this chapter).

Because enumerations are program-defined types 13.1 -- Introduction to program-defined (user-defined) types, each enumeration needs to be fully defined before we can use it (a forward declaration is not sufficient).

Unscoped enumerations

Unscoped enumerations are defined via the `enum` keyword.

Enumerated types are best taught by example, so let's define an unscoped enumeration that can hold some color values. We'll explain how it all works below.

```
// Define a new unscoped enumeration named Color
enum Color
{
    // Here are the enumerators
    // These symbolic constants define all the possible values this type can hold
    // Each enumerator is separated by a comma, not a semicolon
    red,
    green,
    blue, // trailing comma optional but recommended
}; // the enum definition must end with a semicolon

int main()
{
    // Define a few variables of enumerated type Color
    Color apple { red };   // my apple is red
    Color shirt { green }; // my shirt is green
    Color cup { blue };    // my cup is blue

    Color socks { white }; // error: white is not an enumerator of Color
    Color hat { 2 };       // error: 2 is not an enumerator of Color

    return 0;
}
```

We start our example by using the `enum` keyword to tell the compiler that we are defining an unscoped enumeration, which we've named `Color`.

Inside a pair of curly braces, we define the enumerators for the `Color` type: `red`, `green`, and `blue`. These enumerators define the specific values that type `Color` is restricted to. Each enumerator must be separated by a comma (not a semicolon) -- a trailing comma after the last enumerator is optional but recommended for consistency.

It is most common to define one enumerator per line, but in simple cases (where there are a small number of enumerators and no comments are needed), they may all be defined on a single line.

The type definition for `Color` ends with a semicolon. We've now fully defined what enumerated type `Color` is!

Inside `main()`, we instantiate three variables of type `Color`: `apple` is initialized with the color `red`, `shirt` is initialized with the color `green`, and `cup` is initialized with the color `blue`. Memory is allocated for each of these objects. Note that the initializer for an enumerated type must be one of the defined enumerators for that type. The variables `socks` and `hat` cause compile errors because the initializers `white` and `2` are not enumerators of `Color`.

Enumerators are implicitly constexpr.

A reminder

To quickly recap on nomenclature:

- An *enumeration* or *enumerated type* is the program-defined type itself (e.g. `Color`).
- An *enumerator* is a specific named value belonging to the enumeration (e.g. `red`).

Naming enumerations and enumerators

By convention, the names of enumerated types start with a capital letter (as do all program-defined types).

Warning

Enumerations don't have to be named, but unnamed enumerations should be avoided in modern C++.

Enumerators must be given names. Unfortunately, there is no common naming convention for enumerator names. Common choices include starting with lower case (e.g. red), starting with caps (Red), all caps (RED), all caps with a prefix (COLOR_RED), or prefixed with a "k" and intercapped (kColorRed).

Modern C++ guidelines typically recommend avoiding the all caps naming conventions, as all caps is typically used for preprocessor macros and may conflict. We recommend also avoiding the conventions starting with a capital letter, as names beginning with a capital letter are typically reserved for program-defined types.

Best practice

Name your enumerated types starting with a capital letter. Name your enumerators starting with a lower case letter.

Enumerated types are distinct types

Each enumerated type you create is considered to be a **distinct type**, meaning the compiler can distinguish it from other types (unlike typedefs or type aliases, which are considered non-distinct from the types they are aliasing).

Because enumerated types are distinct, enumerators defined as part of one enumerated type can't be used with objects of another enumerated type:

```
enum Pet
{
    cat,
    dog,
    pig,
    whale,
};

enum Color
{
    black,
    red,
    blue,
};

int main()
{
    Pet myPet { black }; // compile error: black is not an enumerator of Pet
    Color shirt { pig }; // compile error: pig is not an enumerator of Color

    return 0;
}
```

You probably didn't want a pig shirt anyway.

Putting enumerations to use

Because enumerators are descriptive, they are useful for enhancing code documentation and readability. Enumerated types are best used when you have a smallish set of related constants, and objects only need to hold one of those values at a time.

Commonly defined enumerations include days of the week, the cardinal directions, and the suits in a deck of cards:

```
enum DaysOfWeek
{
    sunday,
    monday,
    tuesday,
    wednesday,
    thursday,
    friday,
    saturday,
};

enum CardinalDirections
{
    north,
    east,
    south,
    west,
};

enum CardSuits
{
    clubs,
    diamonds,
    hearts,
    spades,
};
```

Sometimes functions will return a status code to the caller to indicate whether the function executed successfully or encountered an error. Traditionally, small negative numbers were used to represent different possible error codes. For example:

```
int readFileContents()
{
    if (!openFile())
        return -1;
    if (!readFile())
        return -2;
    if (!parseFile())
        return -3;

    return 0; // success
}
```

However, using magic numbers like this isn't very descriptive. A better method would be to use an enumerated type:

```
enum FileReadResult
{
    readResultSuccess,
    readResultErrorFileOpen,
    readResultErrorFileRead,
    readResultErrorFileParse,
};

FileReadResult readFileContents()
{
    if (!openFile())
        return readResultErrorFileOpen;
    if (!readFile())
        return readResultErrorFileRead;
    if (!parseFile())
        return readResultErrorFileParse;

    return readResultSuccess;
}
```

Then the caller can test the function's return value against the appropriate enumerator, which is easier to understand than testing the return result for a specific integer value.

```
if (readFileContents() == readResultSuccess)
{
    // do something
}
else
{
    // print error message
}
```

Enumerated types can also be put to good use in games, to identify different types of items, or monsters, or terrain. Basically, anything that is a small set of related objects.

For example:

```
enum ItemType
{
        sword,
        torch,
        potion,
};

int main()
{
        ItemType holding{ torch };

        return 0;
}
```

Enumerated types can also make for useful function parameters when the user needs to make a choice between two or more options:

```
enum SortOrder
{
    alphabetical,
    alphabeticalReverse,
    numerical,
};

void sortData(SortOrder order)
{
    switch (order)
    {
        case alphabetical:
            // sort data in forwards alphabetical order
            break;
        case alphabeticalReverse:
            // sort data in backwards alphabetical order
            break;
        case numerical:
            // sort data numerically
            break;
    }
}
```

Many languages use enumerations to define Booleans -- after all, a Boolean is essentially just an enumeration with 2 enumerators: `false` and `true`! However, in C++, `true` and `false` are defined as keywords instead of enumerators.

Because enumerations are small and inexpensive to copy, it is fine to pass (and return) them by value.

In lesson O.1 -- Bit flags and bit manipulation via std::bitset, we discussed bit flags. Enums can also be used to define a collection of related bit flag positions for use with `std::bitset`:

```
#include <bitset>
#include <iostream>

namespace Flags
{
    enum State
    {
        isHungry,
        isSad,
        isMad,
        isHappy,
        isLaughing,
        isAsleep,
        isDead,
        isCrying,
    };
}

int main()
{
    std::bitset<8> me{};
    me.set(Flags::isHappy);
    me.set(Flags::isLaughing);

    std::cout << std::boolalpha; // print bool as true/false

    // Query a few states (we use the any() function to see if any bits remain set)
    std::cout << "I am happy? " << me.test(Flags::isHappy) << '\n';
    std::cout << "I am laughing? " << me.test(Flags::isLaughing) << '\n';

    return 0;
}
```

If you're wondering how we can use an enumerator where an integral value is expected, unscoped enumerators will implicitly convert to integral values. We will explore this further in the next lesson (13.3 -- Unscoped enumerator integral conversions).

The scope of unscoped enumerations

Unscoped enumerations are named such because they put their enumerator names into the same scope as the enumeration definition itself (as opposed to creating a new scope region like a namespace does).

For example, given this program:

```
enum Color // this enum is defined in the global namespace
{
    red, // so red is put into the global namespace
    green,
    blue,
};

int main()
{
    Color apple { red }; // my apple is red

    return 0;
}
```

The `Color` enumeration is defined in the global scope. Therefore, all the enumeration names (`red`, `green`, and `blue`) also go into the global scope. This pollutes the global scope and significantly raises the chance of naming collisions.

One consequence of this is that an enumerator name can't be used in multiple enumerations within the same scope:

```
enum Color
{
    red,
    green,
    blue, // blue is put into the global namespace
};

enum Feeling
{
    happy,
    tired,
    blue, // error: naming collision with the above blue
};

int main()
{
    Color apple { red }; // my apple is red
    Feeling me { happy }; // I'm happy right now (even though my program doesn't
compile)

    return 0;
}
```

In the above example, both unscoped enumerations (`Color` and `Feeling`) put enumerators with the same name `blue` into the global scope. This leads to a naming collision and subsequent compile error.

Unscoped enumerations also provide a named scope region for their enumerators (much like a namespace acts as a named scope region for the names declared within). This means we can access the enumerators of an unscoped enumeration as follows:

```
enum Color
{
    red,
    green,
    blue, // blue is put into the global namespace
};

int main()
{
    Color apple { red }; // okay, accessing enumerator from global namespace
    Color raspberry { Color::red }; // also okay, accessing enumerator from scope of
Color

    return 0;
}
```

Most often, unscoped enumerators are accessed without using the scope resolution operator.

Avoiding enumerator naming collisions

There are quite a few common ways to prevent unscoped enumerator naming collisions.

One option is to prefix each enumerator with the name of the enumeration itself:

```
enum Color
{
    color_red,
    color_blue,
    color_green,
};

enum Feeling
{
    feeling_happy,
    feeling_tired,
    feeling_blue, // no longer has a naming collision with color_blue
};

int main()
{
    Color paint { color_blue };
    Feeling me { feeling_blue };

    return 0;
}
```

This still pollutes the namespace but reduces the chance for naming collisions by making the names longer and more unique.

A better option is to put the enumerated type inside something that provides a separate scope region, such as a namespace:

```
namespace Color
{
    // The names Color, red, blue, and green are defined inside namespace Color
    enum Color
    {
        red,
        green,
        blue,
    };
}

namespace Feeling
{
    enum Feeling
    {
        happy,
        tired,
        blue, // Feeling::blue doesn't collide with Color::blue
    };
}

int main()
{
    Color::Color paint{ Color::blue };
    Feeling::Feeling me{ Feeling::blue };

    return 0;
}
```

This means we now have to prefix our enumeration and enumerator names with the name of the scoped region.

For advanced readers

Classes also provide a scope region, and it's common to put enumerated types related to a class inside the scope region of the class. We discuss this in lesson 15.3 -- Nested types (member types).

A related option is to use a scoped enumeration (which defines its own scope region). We'll discuss scoped enumerations shortly (13.6 -- Scoped enumerations (enum classes)).

Best practice

Prefer putting your enumerations inside a named scope region (such as a namespace or class) so the enumerators don't pollute the global namespace.

Alternatively, if an enumeration is only used within the body of a single function, the enumeration should be defined inside the function. This limits the scope of the enumeration and its enumerators to just that function. The enumerators of such an enumeration will shadow identically named enumerators defined in the global scope.

Comparing against enumerators

We can use the equality operators (`operator==` and `operator!=`) to test whether an enumeration has the value of a particular enumerator or not.

```
#include <iostream>

enum Color
{
    red,
    green,
    blue,
};

int main()
{
    Color shirt{ blue };

    if (shirt == blue) // if the shirt is blue
        std::cout << "Your shirt is blue!";
    else
        std::cout << "Your shirt is not blue!";

    return 0;
}
```

In the above example, we use an if-statement to test whether `shirt` is equal to the enumerator `blue`. This gives us a way to conditionalize our program's behavior based on what enumerator our enumeration is holding.

We'll make more use of this in the next lesson.

Quiz time

Question #1

Define an enumerated type named MonsterType to choose between the following monster races: orc, goblin, troll, ogre, and skeleton.

Show Solution

Question #2

Put the MonsterType enumeration inside a namespace. Then, create a main() function and instantiate a troll. The program should compile.

Show Solution