13.3 — Unscoped enumerator integral conversions

In the prior lesson (13.2 -- Unscoped enumerations), we mentioned that enumerators are symbolic constants. What we didn't tell you then is that these enumerators have values that are of an integral type.

This is similar to the case with chars (4.11 -- Chars). Consider:

```
char ch { 'A' };
```

A char is really just a 1-byte integral value, and the character `'A'` gets converted to an integral value (in this case, `65`) and stored.

When we define an enumeration, each enumerator is automatically associated with an integer value based on its position in the enumerator list. By default, the first enumerator is given the integral value `0`, and each subsequent enumerator has a value one greater than the previous enumerator:

```
enum Color
{
    black,   // 0
    red,     // 1
    blue,    // 2
    green,   // 3
    white,   // 4
    cyan,    // 5
    yellow,  // 6
    magenta, // 7
};

int main()
{
    Color shirt{ blue }; // shirt actually stores integral value 2

    return 0;
}
```

It is possible to explicitly define the value of enumerators. These integral values can be positive or negative, and can share the same value as other enumerators. Any non-defined enumerators are given a value one greater than the previous enumerator.

```
enum Animal
{
    cat = -3,    // values can be negative
    dog,         // -2
    pig,         // -1
    horse = 5,
    giraffe = 5, // shares same value as horse
    chicken,     // 6
};
```

Note in this case, `horse` and `giraffe` have been given the same value. When this happens, the enumerators become non-distinct -- essentially, `horse` and `giraffe` are interchangeable. Although C++ allows it, assigning the same value to two enumerators in the same enumeration should generally be avoided.

Most of the time, the default values for enumerators will be exactly what you want, so do not provide your own values unless you have a specific reason to do so.

Best practice

Avoid assigning explicit values to your enumerators unless you have a compelling reason to do so.

Value-initializing an enumeration

If an enumeration is zero-initialized (which happens when we use value-initialization), the enumeration will be given value `0`, even if there is no corresponding enumerator with that value.

```
#include <iostream>

enum Animal
{
    cat = -3,    // -3
    dog,         // -2
    pig,         // -1
    // note: no enumerator with value 0 in this list
    horse = 5,   // 5
    giraffe = 5, // 5
    chicken,     // 6
};

int main()
{
    Animal a {}; // value-initialization zero-initializes a to value 0
    std::cout << a; // prints 0

    return 0;
}
```

This has two semantic consequences:

> If there is an enumerator with value 0, value-initialization defaults the enumeration to the meaning of that enumerator. For example, using the prior `enum Color` example, a value-initialized `Color` will default to `black`). For this reason, it is a good idea to consider making the enumerator with value 0 the one that represents the best default meaning for your enumeration.

Something like this is likely to cause problems:

```
enum UniverseResult
{
    destroyUniverse, // default value (0)
    saveUniverse
};
```

> If there is no enumerator with value 0, value-initialization makes it easy to create a semantically invalid enumeration. In such cases, we recommend adding an “invalid” or “unknown” enumerator with value 0 so that you have documentation for the meaning of that state, and a name for that state that you can explicitly handle.

```
enum Winner
{
    winnerUnknown, // default value (0)
    player1,
    player2,
};

// somewhere later in your code
if (w == winnerUnknown) // handle case appropriately
```

Best practice

Make the enumerator representing 0 the one that is the best default meaning for your enumeration. If no good default meaning exists, consider adding an “invalid” or “unknown” enumerator that has value 0, so that state is explicitly documented and can be explicitly handled where appropriate.

Unscoped enumerations will implicitly convert to integral values

Even though enumerations store integral values, they are not considered to be an integral type (they are a compound type). However, an unscoped enumeration will implicitly convert to an integral value. Because enumerators are compile-time constants, this is a constexpr conversion (we cover these in lesson 10.4 -- Narrowing conversions, list initialization, and constexpr initializers).

Consider the following program:

```
#include <iostream>

enum Color
{
    black, // assigned 0
    red, // assigned 1
    blue, // assigned 2
    green, // assigned 3
    white, // assigned 4
    cyan, // assigned 5
    yellow, // assigned 6
    magenta, // assigned 7
};

int main()
{
    Color shirt{ blue };

    std::cout << "Your shirt is " << shirt << '\n'; // what does this do?

    return 0;
}
```

Since enumerated types hold integral values, as you might expect, this prints:

```
Your shirt is 2
```

When an enumerated type is used in a function call or with an operator, the compiler will first try to find a function or operator that matches the enumerated type. For example, when the compiler tries to compile `std::cout << shirt`, the compiler will first look to see if `operator<<` knows how to print an object of type `Color` (because `shirt` is of type `Color`) to `std::cout`. It doesn't.

Since the compiler can't find a match, it will then then check if `operator<<` knows how to print an object of the integral type that the unscoped enumeration converts to. Since it does, the value in `shirt` gets converted to an integral value and printed as integral value `2`.

Related content

We show how to convert an enumeration into a string in lesson 13.4 -- Converting an enumeration to and from a string.
We teach `std::cout` how to print an enumerator in lesson 13.5 -- Introduction to overloading the I/O operators.

Enumeration size and underlying type (base)

Enumerators have values that are of an integral type. But what integral type? The specific integral type used to represent the value of enumerators is called the enumeration's **underlying type** (or **base**).

For unscoped emumerations, the C++ standard does not specify which specific integral type should be used as the underlying type, so the choice is implementation-defined. Most compilers will use `int` as the underlying type (meaning an unscoped enum will be the same size as an `int`), unless a larger type is required to store the enumerator values. But you shouldn't assume this will hold true for every compiler or platform.

It is possible to explicitly specify an underlying type for an enumeration. For example, if you are working in some bandwidth-sensitive context (e.g. sending data over a network) you may want to specify a smaller type for your enumeration:

```
#include <cstdint>  // for std::int8_t
#include <iostream>

// Use an 8-bit integer as the enum underlying type
enum Color : std::int8_t
{
    black,
    red,
    blue,
};

int main()
{
    Color c{ black };
    std::cout << sizeof(c) << '\n'; // prints 1 (byte)

    return 0;
}
```

Best practice

Specify the base type of an enumeration only when necessary.

Warning

Because `std::int8_t` and `std::uint8_t` are usually type aliases for char types, using either of these types as the enum base will most likely cause the enumerators to print as char values rather than int values.

Integer to unscoped enumerator conversion

While the compiler will implicitly convert an unscoped enumeration to an integer, it will *not* implicitly convert an integer to an unscoped enumeration. The following will produce a compiler error:

```
enum Pet // no specified base
{
    cat, // assigned 0
    dog, // assigned 1
    pig, // assigned 2
    whale, // assigned 3
};

int main()
{
    Pet pet { 2 }; // compile error: integer value 2 won't implicitly convert to a
Pet
    pet = 3;       // compile error: integer value 3 won't implicitly convert to a
Pet

    return 0;
}
```

There are two ways to work around this.

First, you can explicitly convert an integer to an unscoped enumerator using `static_cast`:

```
enum Pet // no specified base
{
    cat, // assigned 0
    dog, // assigned 1
    pig, // assigned 2
    whale, // assigned 3
};

int main()
{
    Pet pet { static_cast<Pet>(2) }; // convert integer 2 to a Pet
    pet = static_cast<Pet>(3);       // our pig evolved into a whale!

    return 0;
}
```

We'll see an example in lesson 13.4 -- Converting an enumeration to and from a string where we make use of this.

It is safe to static_cast any integral value that is represented by an enumerator of the target enumeration. Since our `Pet` enumeration has enumerators with values `0`, `1`, `2`, and `3`, static_casting integral values `0`, `1`, `2`, and `3` to a `Pet` is valid.

It is also safe to static_cast any integral value that is in range of the target enumeration's underlying type, even if there are no enumerators representing that value. Static casting a value outside the range of the underlying type will result in undefined behavior.

For advanced readers

If the enumeration has an explicitly defined underlying type, the range of the enumeration is identical to the range of the underlying type.

If the enumeration does not have an explicit underlying type, things are a bit more complicated. In this case, the compiler gets to pick the underlying type, and it can pick any signed or unsigned type so long as the value of all enumerators fit in that type. Given this, it is only safe to static_cast integral values that fit in the range of the smallest number of bits that can hold the value of all enumerators.

Let's do two examples to illustrate this:

- With enumerators that have values 2, 9, and 12, these enumerators could minimally fit in an unsigned 4-bit integral type with range 0 to 15. Therefore, it is only safe to static_cast integral values 0 through 15 to this enumerated type.
- With enumerators that have values -28, 2, and 6, these enumerators could minimally fit in a signed 6-bit integral type with range -32 to 31. Therefore, it is only safe to static_cast integral values -32 through 31 to this enumerated type.

Second, as of C++17, if an unscoped enumeration has an explicitly specified base, then the compiler will allow you to list initialize an unscoped enumeration using an integral value:

```
enum Pet: int // we've specified a base
{
    cat, // assigned 0
    dog, // assigned 1
    pig, // assigned 2
    whale, // assigned 3
};

int main()
{
    Pet pet1 { 2 }; // ok: can brace initialize unscoped enumeration with specified
base with integer (C++17)
    Pet pet2 (2);   // compile error: cannot direct initialize with integer
    Pet pet3 = 2;   // compile error: cannot copy initialize with integer

    pet1 = 3;       // compile error: cannot assign with integer

    return 0;
}
```

Quiz time

Question #1

True or false. Enumerators can be:

> Given an integer value

Show Solution

Given no explicit value

Show Solution

Given a floating point value

Show Solution

Given a negative value

Show Solution

Given a non-unique value

Show Solution

Given the value of a prior enumerator (e.g. magenta = red)

Show Solution

Given a non-constexpr value

Show Solution