13.3 — Unscoped enumeration input and output

ALEX SEPTEMBER 16, 2023

In the prior lesson (13.2 -- Unscoped enumerations (https://www.learncpp.com/cpp-tutorial/unscoped-enumerations/)), we mentioned that enumerators are symbolic constants. What we didn't tell you then is that enumerators are integral symbolic constants. As a result, enumerated types actually hold an integral value.

This is similar to the case with chars (4.11 -- Chars (https://www.learncpp.com/cpp-tutorial/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 enumerator, each enumerator is automatically assigned an integer value based on its position in the enumerator list. By default, the first enumerator is assigned the integral value `0`, and each subsequent enumerator has a value one greater than the previous enumerator:

```
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 }; // This actually stores the 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,
    dog,         // assigned -2
    pig,         // assigned -1
    horse = 5,
    giraffe = 5, // shares same value as horse
    chicken,     // assigned 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.

Best practice

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

Unscoped enumerations will implicitly convert to integral values

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.

If the compiler can't find a match, the compiler will then implicitly convert an unscoped enumeration or enumerator to its corresponding integer value. Because `std::cout` does know how to print an integral value, the value in `shirt` gets converted to an integer and printed as integer value `2`.

Printing enumerator names

Most of the time, printing an enumeration as an integral value (such as `2`) isn't what we want. Instead, we typically will want to print the name of whatever the enumerator represents (`blue`). But to do that, we need some way to convert the integral value of the enumeration (`2`) into a string matching the enumerator name (`"blue"`).

As of C++20, C++ doesn't come with any easy way to do this, so we'll have to find a solution ourselves. Fortunately, that's not very difficult. The typical way to do this is to write a function that takes an enumerated type as a parameter and then outputs the corresponding string (or returns the string to the caller).

The typical way to do this is to test our enumeration against every possible enumerator:

```
// Using if-else for this is inefficient
void printColor(Color color)
{
    if (color == black) std::cout << "black";
    else if (color == red) std::cout << "red";
    else if (color == blue) std::cout << "blue";
    else std::cout << "???";
}
```

However, using a series of if-else statements for this is inefficient, as it requires multiple comparisons before a match is found. A more efficient way to do the same thing is to use a switch statement. In the following example, we will also return our `Color` as a `std::string`, to give the caller more flexibility to do whatever they want with the name (including print it):

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

enum Color
{
    black,
    red,
    blue,
};


// We'll show a better version of this for C++17 below
std::string getColor(Color color)
{
    switch (color)
    {
    case black: return "black";
    case red:   return "red";
    case blue:  return "blue";
    default:    return "???";
    }
}

int main()
{
    Color shirt { blue };

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

    return 0;
}
```

This prints:

```
Your shirt is blue
```

This likely performs better than the if-else chain (switch statements tend to be more efficient than if-else chains), and it's easier to read too. However, this version is still inefficient, because we need to create and return a `std::string` (which is expensive) every time the function is called.

In C++17, a more efficient option is to replace `std::string` with `std::string_view`. `std::string_view` allows us to return string literals in a way that is much less expensive to copy.

```
#include <iostream>
#include <string_view> // C++17

enum Color
{
    black,
    red,
    blue,
};

constexpr std::string_view getColor(Color color) // C++17
{
    switch (color)
    {
    case black: return "black";
    case red:   return "red";
    case blue:  return "blue";
    default:    return "???";
    }
}

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

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

    return 0;
}
```

Related content

Constexpr return types are covered in lesson 5.8 -- Constexpr and consteval functions (https://www.learncpp.com/cpp-tutorial/constexpr-and-consteval-functions/).

()Teaching `operator<<` how to print an enumerator (#insertion)

Although the above example functions well, we still have to remember the name of the function we created to get the enumerator name. While this usually isn't too burdensome, it can become more problematic if you have lots of enumerations. Using operator overloading (a capability similar to function overloading), we can actually teach `operator<<` how to print the value of a program-defined enumeration! We haven't explained how this works yet, so consider it a bit of magic for now:

```
#include <iostream>

enum Color
{
    black,
    red,
    blue,
};

// Teach operator<< how to print a Color
// Consider this magic for now since we haven't explained any of the concepts it uses yet
// std::ostream is the type of std::cout
// The return type and parameter type are references (to prevent copies from being made)!
std::ostream& operator<<(std::ostream& out, Color color)
{
    switch (color)
    {
    case black: return out << "black";
    case red:   return out << "red";
    case blue:  return out << "blue";
    default:    return out << "???";
    }
}

int main()
{
    Color shirt{ blue };
    std::cout << "Your shirt is " << shirt << '\n'; // it works!

    return 0;
}
```

This prints:

```
Your shirt is blue
```

For advanced readers

For the curious, here's what the above code is actually doing. When we try to print `shirt` using `std::cout` and `operator<<`, the compiler will see that we've overloaded `operator<<` to work with objects of type `Color`. This overloaded `operator<<` function is then called with `std::cout` as the `out` parameter, and our `shirt` as parameter `color`. Since `out` is a reference to `std::cout`, a statement such as `out << "blue"` is really just printing `"blue"` to `std::cout`.

We cover overloading the I/O operators in lesson 21.4 -- Overloading the I/O operators (https://www.learncpp.com/cpp-tutorial/overloading-the-io-operators/). For now, you can copy this code and replace `Color` with your own enumerated type.

()Enumeration size and underlying type (base) (#base)

The enumerators of an enumeration are integral constants. The specific integral type used to represent enumerators is called the **underlying type** (or **base**).

For unscoped enumerators, the C++ standard does not specify which specific integral type should be used as the underlying type. Most compilers will use type `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.

It is possible to specify a different underlying type. 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:

```
#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 unscoped enumerators to an integer, it will not implicitly convert an integer to an unscoped enumerator. 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 force the compiler to 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 a moment where this can be useful.

Second, in C++17, if an unscoped enumeration has a 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 with integer
    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;
}
```

Unscoped enumerator input

Because `Pet` is a program-defined type, the language doesn't know how to input a Pet using `std::cin`:

```
#include <iostream>

enum Pet
{
    cat, // assigned 0
    dog, // assigned 1
    pig, // assigned 2
    whale, // assigned 3
};

int main()
{
    Pet pet { pig };
    std::cin >> pet; // compile error, std::cin doesn't know how to input a Pet

    return 0;
}
```

To work around this, we can read in an integer, and use `static_cast` to convert the integer to an enumerator of the appropriate enumerated type:

```
#include <iostream>

enum Pet
{
    cat, // assigned 0
    dog, // assigned 1
    pig, // assigned 2
    whale, // assigned 3
};

int main()
{
    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;
}
```

For advanced readers

Similar to how we were able to teach `operator<<` to output an enum type above, we can also teach `operator>>` how to input an enum type:

```
#include <iostream>

enum Pet
{
    cat, // assigned 0
    dog, // assigned 1
    pig, // assigned 2
    whale, // assigned 3
};

// Consider this magic for now
// We pass pet by reference so we can have the function modify its value
std::istream& operator>> (std::istream& in, Pet& pet)
{
    int input{};
    in >> input; // input an integer

    pet = static_cast<Pet>(input);
    return in;
}

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

    Pet pet{};
    std::cin >> pet; // input our pet using std::cin

    std::cout << pet << '\n'; // prove that it worked

    return 0;
}
```

Again, consider this a bit of magic for now (since we haven't explained the concepts behind it yet), but you might find it handy.

In lesson 17.6 -- std::array and enumerations (https://www.learncpp.com/cpp-tutorial/stdarray-and-enumerations/), we show an improved version of `operator>>` that allows input via text (rather than an int).

Quiz time

Question #1

True or false. Enumerators can be:

- Given an integer value

Show Solution (javascript:void(0))

- Given no explicit value

Show Solution (javascript:void(0))

- Given a floating point value

Show Solution (javascript:void(0))

- Given a negative value

Show Solution (javascript:void(0))

- Given a non-unique value

Show Solution (javascript:void(0))

- Initialized with the value of prior enumerators (e.g. magenta = red)

Show Solution (javascript:void(0))

Next lesson
13.4 Scoped enumerations (enum classes)

Back to table of contents

Previous lesson
13.2 Unscoped enumerations