13.4 — Converting an enumeration to and from a string

In the prior lesson (13.3 -- Unscoped enumerator integral conversions), we showed an example like this:

```
#include <iostream>

enum Color
{
    black, // 0
    red,   // 1
    blue,  // 2
};

int main()
{
    Color shirt{ blue };

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

    return 0;
}
```

This prints:

```
Your shirt is 2
```

Because `operator<<` doesn't know how to print a `Color`, the compiler will implicitly convert `Color` into an integral value and print that instead.

Most of the time, printing an enumeration as an integral value (such as `2`) isn't what we want. Instead, we typically want to print the name of whatever the enumerator represents (e.g. `blue`). C++ doesn't come with an out-of-the-box way to do this, so we'll have to find a solution ourselves. Fortunately, that's not very difficult.

Getting the name of an enumerator

The typical way to get the name of an enumerator is to write a function that allows us to pass in an enumerator and returns the enumerator's name as a string. But that requires some way to determine which string should be returned for a given enumerator.

There are two common ways to do this.

In lesson 8.5 -- Switch statement basics, we noted that a switch statement can switch on either an integral value or an enumerated value. In the following example, we use a switch statement to select an enumerator and return the appropriate color string literal for that enumerator:

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

enum Color
{
    black,
    red,
    blue,
};

constexpr std::string_view getColorName(Color color)
{
    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 " << getColorName(shirt) << '\n';

    return 0;
}
```

This prints:

```
Your shirt is blue
```

In the above example, we switch on `color`, which holds the enumerator we passed in. Inside the switch, we have a case-label for each enumerator of `Color`. Each case returns the name of the appropriate color as a C-style string literal. This C-style string literal gets implicitly converted into a `std::string_view`, which is returned to the caller. We also have a default case which returns `"???"`, in case the user passes in something we didn't expect.

A reminder

Because C-style string literals exist for the entire program, it's okay to return a `std::string_view` that is viewing a C-style string literal. When the `std::string_view` is copied back to the caller, the C-style string literal being viewed will still exist.

The function is constexpr so that we can use the color's name in a constant expression.

Related content

Constexpr functions are covered in lesson 5.8 -- Constexpr and consteval functions.

While this lets us get the name of an enumerator, if we want to print that name to the console, having to do `std::cout << getColorName(shirt)` isn't quite as nice as `std::cout << shirt`. We'll teach `std::cout` how to print an enumeration in upcoming lesson 13.5 -- Introduction to overloading the I/O operators.

The second way to solve the program of mapping enumerators to strings is to use an array. We cover this in lesson 17.6 -- std::array and enumerations.

Unscoped enumerator input

Now let's take a look at an input case. In the following example, we define a `Pet` enumeration. 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,   // 0
    dog,   // 1
    pig,   // 2
    whale, // 3
};

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

    return 0;
}
```

One simple way to work around this is to 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,   // 0
    dog,   // 1
    pig,   // 2
    whale, // 3
};

constexpr std::string_view getPetName(Pet pet)
{
    switch (pet)
    {
    case cat:   return "cat";
    case dog:   return "dog";
    case pig:   return "pig";
    case whale: return "whale";
    default:    return "???";
    }
}

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

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

    if (input < 0 || input > 3)
        std::cout << "You entered an invalid pet\n";
    else
    {
        Pet pet{ static_cast<Pet>(input) }; // static_cast our integer to a Pet
        std::cout << "You entered: " << getPetName(pet) << '\n';
    }

    return 0;
}
```

While this works, it’s a bit awkward. Also note that we should only `static_cast<Pet>(input)` once we know `input` is in range of the enumerator.

Getting an enumeration from a string

Instead of inputting a number, it would be nicer if the user could type in a string representing an enumerator (e.g. “pig”), and we could convert that string into the appropriate `Pet` enumerator. However, doing this requires us to solve a couple of challenges.

First, we can’t switch on a string, so we need to use something else to match the string the user passed in. The simplest approach here is to use a series of if-statements.

Second, what `Pet` enumerator should we return if the user passes in an invalid string? One option would be to add an enumerator to represent “none/invalid”, and return that. However, a better option is to use `std::optional` here.

Related content

We cover `std::optional` in lesson 12.15 -- std::optional.

```
#include <iostream>
#include <optional> // for std::optional

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

constexpr std::string_view getPetName(Pet pet)
{
    switch (pet)
    {
    case cat:   return "cat";
    case dog:   return "dog";
    case pig:   return "pig";
    case whale: return "whale";
    default:    return "???";
    }
}

constexpr std::optional<Pet> getPetFromString(std::string_view sv)
{
    if (sv == "cat")   return cat;
    if (sv == "dog")   return dog;
    if (sv == "pig")   return pig;
    if (sv == "whale") return whale;

    return {};
}

int main()
{
    std::cout << "Enter a pet: cat, dog, pig, or whale: ";
    std::string s{};
    std::cin >> s;

    std::optional<Pet> pet { getPetFromString(s) };

    if (!pet)
        std::cout << "You entered an invalid pet\n";
    else
        std::cout << "You entered: " << getPetName(*pet) << '\n';

    return 0;
}
```

In the above solution, we use a series of if-else statements to do string comparisons. If the user’s input string matches an enumerator string, we return the appropriate enumerator. If none of the strings match, we return `{}`, which means “no value”.

For advanced readers

Note that the above solution only matches lower case letters. If you want to match any letter case, you can use the following function to convert the user's input to lower case:

```
#include <algorithm> // for std::transform
#include <cctype>    // for std::tolower

// This function returns a std::string that is the lower-case version of the std::string_view passed in.
// Only 1:1 character mapping can be performed by this function
constexpr std::string toASCIILowerCase(std::string_view sv)
{
    std::string upper{};
    std::transform(sv.begin(), sv.end(), std::back_inserter(upper),
        [](unsigned char c){ return std::tolower(c); });
    return upper;
}
```

This function steps through each character in `std::string_view sv`, converts it to a lower case character using `std::tolower()` (with the help of a lambda), and then appends that lower-case character to `upper`.

We cover lambdas in lesson 20.6 -- Introduction to lambdas (anonymous functions).

Similar to the output case, it would be better if we could just `std::cin >> pet`. We'll cover this in upcoming lesson 13.5 -- Introduction to overloading the I/O operators.