17.6 — std::array and enumerations

In lesson 16.9 -- Array indexing and length using enumerators, we discussed arrays and enumerations.

Now that we have `constexpr std::array` in our toolkit, we're going to continue that discussion and show a few additional tricks.

Using static assert to ensure the proper number of array initializers

When initializing a `constexpr std::array` using CTAD, the compiler will deduce how long the array should be from the number of initializers. If less initializers are provided than there should be, the array will be shorter than expected, and indexing it can lead to undefined behavior.

For example:

```
#include <array>
#include <iostream>

enum StudentNames
{
    kenny, // 0
    kyle, // 1
    stan, // 2
    butters, // 3
    cartman, // 4
    max_students // 5
};

int main()
{
    constexpr std::array testScores { 78, 94, 66, 77 }; // oops, only 4 values

    std::cout << "Cartman got a score of " << testScores[StudentNames::cartman] <<
'\n'; // undefined behavior due to invalid index

    return 0;
}
```

Whenever the number of initializers in a `constexpr std::array` can be reasonably sanity checked, you can do so using a static assert:

```
#include <array>
#include <iostream>

enum StudentNames
{
    kenny, // 0
    kyle, // 1
    stan, // 2
    butters, // 3
    cartman, // 4
    max_students // 5
};

int main()
{
    constexpr std::array testScores { 78, 94, 66, 77 };

    // Ensure the number of test scores is the same as the number of students
    static_assert(std::size(testScores) == max_students); // compile error:
static_assert condition failed

    std::cout << "Cartman got a score of " << testScores[StudentNames::cartman] <<
'\n';

    return 0;
}
```

That way, if you add a new enumerator later but forget to add a corresponding initializer to `testScores`, the program will fail to compile.

You can also use a static assert to ensure two different `constexpr std::array` have the same length.

Using constexpr arrays for better enumeration input and output

In lesson 13.5 -- Introduction to overloading the I/O operators, we covered a few ways to output the names of enumerators, as well as a way to input an enumerated value as an integer.

Let's improve these functions a bit. In the following example, we use an array to hold the name of each enumerator as a `std::string_view`. We then use this array for two purposes:

- To easily output the name of enumerators as a string.
- To easily input the name of enumerators as a string.

```
#include <array>
#include <iostream>
#include <string>
#include <string_view>

namespace Color
{
    enum Type
    {
        black,
        red,
        blue,
        max_colors
    };

    // use sv suffix so std::array will infer type as std::string_view
    using namespace std::string_view_literals; // for sv suffix
    constexpr std::array colorName { "black"sv, "red"sv, "blue"sv };

    // Make sure we've defined strings for all our colors
    static_assert(std::size(colorName) == max_colors);
};

constexpr std::string_view getColorName(Color::Type color)
{
    return Color::colorName[color];
}

// Teach operator<< how to print a Color
// 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::Type color)
{
    return out << getColorName(color);
}

// Teach operator>> how to input a Color by name
// We pass color by non-const reference so we can have the function modify its value
std::istream& operator>> (std::istream& in, Color::Type& color)
{
    std::string input {};
    std::getline(in >> std::ws, input);

    // See if we can find a match
    for (std::size_t index=0; index < Color::colorName.size(); ++index)
    {
        if (input == Color::colorName[index])
        {
            color = static_cast<Color::Type>(index);
            return in;
        }
```

```
    }

    // We didn't find a match, so input must have been invalid
    // so we will set input stream to fail state
    in.setstate(std::ios_base::failbit);

    // On an extraction failure, operator>> zero-initializes fundamental types
    // Uncomment the following line to make this operator do the same thing
    // color = {};
    return in;
}

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

    std::cout << "Enter a new color: ";
    std::cin >> shirt;
    if (!std::cin)
        std::cout << "Invalid\n";
    else
        std::cout << "Your shirt is now " << shirt << '\n';

    return 0;
}
```

This prints:

```
Your shirt is blue
Enter a new color: red
Your shirt is now red
```

Range-based for-loops and enumerations

Occasionally we run across situations where it would be useful to iterate through the enumerators of an enumeration. While we can do this using a for-loop with an integer index, this is likely to require a lot of static casting of the integer index to our enumeration type.

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

namespace Color
{
    enum Type
    {
        black,
        red,
        blue,
        max_colors
    };

    // use sv suffix so std::array will infer type as std::string_view
    using namespace std::string_view_literals; // for sv suffix
    constexpr std::array colorName { "black"sv, "red"sv, "blue"sv };

    // Make sure we've defined strings for all our colors
    static_assert(std::size(colorName) == max_colors);
};

constexpr std::string_view getColorName(Color::Type color)
{
    return Color::colorName[color];
}

// Teach operator<< how to print a Color
// 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::Type color)
{
    return out << getColorName(color);
}

int main()
{
    // Use a for loop to iterate through all our colors
    for (int i=0; i < Color::max_colors; ++i )
        std::cout << getColorName(static_cast<Color::Type>(i)) << '\n'; // ugly
static_cast here

    return 0;
}
```

Unfortunately, range-based for-loops won't allow you to iterate over the enumerators of an enumeration:

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

namespace Color
{
    enum Type
    {
        black,
        red,
        blue,
        max_colors
    };

    // use sv suffix so std::array will infer type as std::string_view
    using namespace std::string_view_literals; // for sv suffix
    constexpr std::array colorName { "black"sv, "red"sv, "blue"sv };

    // Make sure we've defined strings for all our colors
    static_assert(std::size(colorName) == max_colors);
};

constexpr std::string_view getColorName(Color::Type color)
{
    return Color::colorName[color];
}

// Teach operator<< how to print a Color
// 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::Type color)
{
    return out << getColorName(color);
}

int main()
{
    for (auto c: Color::Type) // compile error: can't traverse enumeration
        std::cout << c < '\n';

    return 0;
}
```

There are many creative solutions for this. Since we can use a range-based for-loop on an array, one of the most straightforward solutions is to create a `constexpr std::array` containing each of our enumerators, and then iterate over that. This method only works if the enumerators have sequential values starting at 0 (but most enumerations do).

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

namespace Color
{
    enum Type
    {
        black,     // 0
        red,       // 1
        blue,      // 2
        max_colors // 3
    };

    using namespace std::string_view_literals; // for sv suffix
    constexpr std::array colorName { "black"sv, "red"sv, "blue"sv };
    static_assert(std::size(colorName) == max_colors);

    constexpr std::array types { black, red, blue }; // A std::array containing all
our enumerators
    static_assert(std::size(types) == max_colors);
};

constexpr std::string_view getColorName(Color::Type color)
{
    return Color::colorName[color];
}

// Teach operator<< how to print a Color
// 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::Type color)
{
    return out << getColorName(color);
}

int main()
{
    for (auto c: Color::types) // ok: we can do a range-based for on a std::array
        std::cout << c << '\n';

    return 0;
}
```

In the above example, since the element type of `Color::types` is `Color::Type`, variable `c` will be deduced as a `Color::Type`, which is exactly what we want!

This prints:

```
black
red
blue
```

Quiz time

Define a namespace named `Animal`. Inside it, define an enum containing the following animals: chicken, dog, cat, elephant, duck, and snake. Also create a struct named `Data` to store each animal's name, number of legs, and the sound it makes. Create a `std::array` of Data and fill out a Data element for each animal.

Ask the user to enter the name of an animal. If the name does not match the name of one of our animals, tell them so. Otherwise, print the data for that animal. Then print the data for all of the other animals that didn't match their input.

For example:

```
Enter an animal: dog
A dog has 4 legs and says woof.

Here is the data for the rest of the animals:
A chicken has 2 legs and says cluck.
A cat has 4 legs and says meow.
A elephant has 4 legs and says pawoo.
A duck has 2 legs and says quack.
A snake has 0 legs and says hissss.

Enter an animal: frog
That animal couldn't be found.

Here is the data for the rest of the animals:
A chicken has 2 legs and says cluck.
A dog has 4 legs and says woof.
A cat has 4 legs and says meow.
A elephant has 4 legs and says pawoo.
A duck has 2 legs and says quack.
A snake has 0 legs and says hissss.
```

Question #1

Show Solution