16.9 — Array indexing and length using enumerators

learncpp.com/cpp-tutorial/array-indexing-and-length-using-enumerators/

One of the bigger documentation problems with arrays is that integer indices do not provide any information to the programmer about the meaning of the index.

Consider an array holding 5 test scores:

```
#include <vector>

int main()
{
    std::vector testScores { 78, 94, 66, 77, 14 };

    testScores[2] = 76; // who does this represent?
}
```

Who is the student represented by `testScores[2]`? It's not clear.

Using unscoped enumerators for indexing

In lesson 16.3 -- std::vector and the unsigned length and subscript problem, we spent a lot of time discussing how the index of `std::vector<T>::operator[]` (and the other C++ container classes that can be subscripted) has type `size_type`, which is generally an alias for `std::size_t`. Therefore, our indices either need to be of type `std::size_t`, or a type that converts to `std::size_t`.

Since unscoped enumerations will implicitly convert to a `std::size_t`, this means we can use unscoped enumerations as array indices to help document the meaning of the index:

```
#include <vector>

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

int main()
{
    std::vector testScores { 78, 94, 66, 77, 14 };

    testScores[Students::stan] = 76; // we are now updating the test score belonging
to stan

    return 0;
}
```

In this way, it's much clearer what each of the array elements represents.

Because enumerators are implicitly constexpr, conversion of an enumerator to an unsigned integral type is not considered a narrowing conversion, thus avoiding signed/unsigned indexing problems.

Using a non-constexpr unscoped enumeration for indexing

The underlying type of an unscoped enumeration is implementation defined (and thus, could be either a signed or unsigned integral type). Because enumerators are implicitly constexpr, as long as we stick to indexing with unscoped enumerators, we won't run into sign conversion issues.

However, if we define a non-constexpr variable of the enumeration type, and then try to index our `std::vector` with that, we may get sign conversion warnings on any platform that defaults unscoped enumerations to a signed type:

```
#include <vector>

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

int main()
{
    std::vector testScores { 78, 94, 66, 77, 14 };
    Students::Names name { Students::stan }; // non-constexpr

    testScores[name] = 76; // may trigger a sign conversion warning if Student::Names
defaults to a signed underlying type

    return 0;
}
```

While we could make `name` constexpr (so that the conversion from a constexpr signed integral type to `std::size_t` is non-narrowing), there's a better way. Instead, we can explicitly specify the underlying type of the enumeration to be an unsigned int:

```
#include <vector>

namespace Students
{
    enum Names : unsigned int // explicitly specifies the underlying type is unsigned
int
    {
        kenny, // 0
        kyle, // 1
        stan, // 2
        butters, // 3
        cartman, // 4
        max_students // 5
    };
}

int main()
{
    std::vector testScores { 78, 94, 66, 77, 14 };
    Students::Names name { Students::stan }; // non-constexpr

    testScores[name] = 76; // not a sign conversion since name is unsigned

    return 0;
}
```

In the example above, since `name` is now guaranteed to be an `unsigned int`, it can be converted to a `std::size_t` without sign conversion issues.

Using a count enumerator

Note that we have defined an extra enumerator at the end of the enumerator list named `max_students`. If all the prior enumerators are using default values (which is recommended) this enumerator will have a default value matching the count of the preceding enumerators. In the example above, `max_students` has value `5`, as there are 5 enumerators defined prior. Informally, we'll call this a **count enumerator**, as its value represents the count of previously defined enumerators.

This count enumerator can then be used anywhere we need a count of the prior enumerators. For example:

```
#include <iostream>
#include <vector>

namespace Students
{
    enum Names
    {
        kenny, // 0
        kyle, // 1
        stan, // 2
        butters, // 3
        cartman, // 4
        // add future enumerators here
        max_students // 5
    };
}

int main()
{
    std::vector<int> testScores(Students::max_students); // Create a vector with 5
elements

    testScores[Students::stan] = 76; // we are now updating the test score belonging
to stan

    std::cout << "The class has " << Students::max_students << " students\n";

    return 0;
}
```

We use `max_students` in two places: first, we create a `std::vector` with a length of `max_students`, so the vector will have one element per student. We also use `max_students` to print the number of students.

This technique is also nice because if another enumerator is added later (just before `max_students`), then `max_students` will automatically become one larger, and all our arrays using `max_students` will update to use the new length without further modification.

```
#include <vector>
#include <iostream>

namespace Students
{
    enum Names
    {
        kenny, // 0
        kyle, // 1
        stan, // 2
        butters, // 3
        cartman, // 4
        wendy, // 5 (added)
        // add future enumerators here
        max_students // now 6
    };
}

int main()
{
    std::vector<int> testScores(Students::max_students); // will now allocate 6
elements

    testScores[Students::stan] = 76; // still works

    std::cout << "The class has " << Students::max_students << " students\n";

    return 0;
}
```

Asserting on array length with a count enumerator

More often, we’re creating an array using an initializer list of values, with the intent of indexing that array with enumerators. In such cases, it can be useful to assert that the size of the container equals our count enumerator. If this assert triggers, then either our enumerator list is incorrect somehow, or we have provided the wrong number of initializers. This can easily happen when a new enumerator is added to the enumeration, but a new initialization value is not added to the array.

For example:

```
#include <cassert>
#include <iostream>
#include <vector>

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

int main()
{
    std::vector testScores { 78, 94, 66, 77, 14 };

    // Ensure the number of test scores is the same as the number of students
    assert(std::size(testScores) == max_students);

    return 0;
}
```

Tip

If your array is constexpr, then you should `static_assert` instead. `std::vector` doesn't support constexpr, but `std::array` (and C-style arrays) do.

We discuss this further in lesson 17.3 -- Passing and returning std::array.

Best practice

Use a `static_assert` to ensure the length of your constexpr array matches your count enumerator.
Use an `assert` to ensure the length of your non-constexpr array matches your count enumerator.

Arrays and enum classes

Because unscoped enumerations pollute the namespace they are defined in with their enumerators, it is preferable to use enum classes in cases where the enum is not already contained in another scope region (e.g. a namespace or class).

However, because enum classes don't have an implicit conversion to integral types, we run into a problem when we try to use their enumerators as array indices:

```
#include <iostream>
#include <vector>

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

int main()
{
    // compile error: no conversion from StudentNames to std::size_t
    std::vector<int> testScores(StudentNames::max_students);

    // compile error: no conversion from StudentNames to std::size_t
    testScores[StudentNames::stan] = 76;

    // compile error: no conversion from StudentNames to any type that operator<< can
output
    std::cout << "The class has " << StudentNames::max_students << " students\n";

    return 0;
}
```

There are a couple of ways to address this. Most obviously, we can `static_cast` the enumerator to an integer:

```
#include <iostream>
#include <vector>

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

int main()
{
    std::vector<int> testScores(static_cast<int>(StudentNames::max_students));

    testScores[static_cast<int>(StudentNames::stan)] = 76;

    std::cout << "The class has " << static_cast<int>(StudentNames::max_students) <<
" students\n";

    return 0;
}
```

However, this is not only a pain to type, it also clutters up our code significantly.

A better option is to use the helper function that we introduced in lesson 13.6 -- Scoped enumerations (enum classes), which allows us to convert the enumerators of enum classes to integral values using unary `operator+`.

```
#include <iostream>
#include <type_traits> // for std::underlying_type_t
#include <vector>

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

// Overload the unary + operator to convert StudentNames to the underlying type
constexpr auto operator+(StudentNames a) noexcept
{
    return static_cast<std::underlying_type_t<StudentNames>>(a);
}

int main()
{
    std::vector<int> testScores(+StudentNames::max_students);

    testScores[+StudentNames::stan] = 76;

    std::cout << "The class has " << +StudentNames::max_students << " students\n";

    return 0;
}
```

However, if you're going to be doing a lot of enumerator to integral conversions, it's probably better to just use a standard enum inside a namespace (or class).

Quiz time

Question #1

Create a program-defined enum (inside a namespace) containing the names of the following animals: chicken, dog, cat, elephant, duck, and snake. Define an array with an element for each of these animals, and use an initializer list to initialize each element to hold the number of legs that animal has. Assert that the array has the correct number of initializers.

Write a main() function that prints the number of legs an elephant has, using the enumerator.

Show Solution