17.2 — std::array length and indexing

learncpp.com/cpp-tutorial/stdarray-length-and-indexing/

In lesson 16.3 -- std::vector and the unsigned length and subscript problem, we discussed the unfortunate decision to make the standard library container classes use unsigned values for lengths and indices. Because `std::array` is a standard library container class, it is subject to the same issues.

In this lesson, we'll recap ways to index and get the length of a `std::array`. Because `std::vector` and `std::array` have similar interfaces, this will parallel the what we covered for `std::vector`. But since only `std::array` has full support for constexpr, we'll focus a little more on that.

Before proceeding, now would be a good time to refresh your memory on "sign conversions are narrowing conversions, except when constexpr" (see 16.3 -- std::vector and the unsigned length and subscript problem).

The length of a `std::array` has type `std::size_t`

`std::array` is implemented as a template struct whose declaration looks like this:

```
template<typename T, std::size_t N> // N is a non-type template parameter
struct array;
```

As you can see, the non-type template parameter representing the array length (`N`) has type `std::size_t`. And as you're probably aware by now, `std::size_t` is a large unsigned integral type.

Related content

We cover class templates (which includes struct templates) in lesson 13.13 -- Class templates and non-type template parameters in lesson 11.9 -- Non-type template parameters.

Thus, when we define a `std::array`, the length non-type template argument must either have type `std::size_t`, or be convertible to a value of type `std::size_t`. Because this value must be constexpr, we don't run into sign conversion issues when we use a signed integral value, as the compiler will happily convert a signed integral value to a `std::size_t` at compile-time without it being considered a narrowing conversion.

As an aside…

Prior to C++23, C++ didn't even have a literal suffix for `std::size_t`, as the implicit compile-time conversion from `int` to `std::size_t` typically suffices for cases where we need a constexpr `std::size_t`.

The suffix was added primarily for type deduction purposes, as `constexpr auto x { 0 }` will give you an `int` rather than a `std::size_t`. In such cases, being able to differentiate `0` (`int`) from `0UZ` (`std::size_t`) without having to use an explicit `static_cast` is useful.

The length and indices of `std::array` have type `size_type`, which is always `std::size_t`

Just like a `std::vector`, `std::array` defines a nested typedef member named `size_type`, which is an alias for the type used for the length (and indices, if supported) of the container. In the case of `std::array`, `size_type` is *always* an alias for `std::size_t`.

Note that the non-type template parameter defining the length of the `std::array` is explicitly defined as `std::size_t` rather than `size_type`. This is because `size_type` is a member of `std::array`, and isn't defined at that point. This is the only place that uses `std::size_t` explicitly -- everywhere else uses `size_type`.

Getting the length of a `std::array`

There are three common ways to get the length of a `std::array` object.

First, we can ask a `std::array` object for its length using the `size()` member function (which returns the length as unsigned `size_type`):

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

int main()
{
    constexpr std::array arr { 9.0, 7.2, 5.4, 3.6, 1.8 };
    std::cout << "length: " << arr.size() << '\n'; // returns length as type
`size_type` (alias for `std::size_t`)
    return 0;
}
```

This prints:

```
length: 5
```

Unlike `std::string` and `std::string_view`, which have both a `length()` and a `size()` member function (that do the same thing), `std::array` (and most other container types in C++) only have `size()`.

Second, in C++17, we can use the `std::size()` non-member function (which for `std::array` just calls the `size()` member function, thus returning the length as unsigned `size_type`).

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

int main()
{
    constexpr std::array arr{ 9, 7, 5, 3, 1 };
    std::cout << "length: " << std::size(arr); // C++17, returns length as type
`size_type` (alias for `std::size_t`)

    return 0;
}
```

Finally, in C++20, we can use the `std::ssize()` non-member function, which returns the length as a large *signed* integral type (usually`std::ptrdiff_t`):

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

int main()
{
    constexpr std::array arr { 9, 7, 5, 3, 1 };
    std::cout << "length: " << std::ssize(arr); // C++20, returns length as a large
signed integral type

    return 0;
}
```

This is the only function of the three which returns the length as a signed type.

Getting the length of a `std::array` as a constexpr value

Because the length of a `std::array` is constexpr, each of the above functions will return the length of a `std::array` as a constexpr value (even when called on a non-constexpr `std::array` object)! This means we can use any of these functions in constant expressions, and the length returned can be implicitly converted to an `int` without it being a narrowing conversion:

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

int main()
{
    std::array arr { 9, 7, 5, 3, 1 }; // note: not constexpr for this example
    constexpr int length{ std::size(arr) }; // ok: return value is constexpr
std::size_t and can be converted to int, not a narrowing conversion

    std::cout << "length: " << length << '\n';

    return 0;
}
```

For Visual Studio users

Visual Studio incorrectly triggers warning C4365 for the above example. The issue has been reported to Microsoft.

Warning

Due to a language defect, the above functions will return a non-constexpr value when called on a `std::array` function parameter passed by (const) reference:

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

void foo(const std::array<int, 5> &arr)
{
    constexpr int length{ std::size(arr) }; // compile error!
    std::cout << "length: " << length << '\n';
}

int main()
{
    std::array arr { 9, 7, 5, 3, 1 };
    constexpr int length{ std::size(arr) }; // works just fine
    std::cout << "length: " << length << '\n';

    foo(arr);

    return 0;
}
```

This defect has been addressed in C++23 by P2280. At the time of writing, few compilers currently support this feature.

A workaround is to make `foo()` a function template where the array length is a non-type template parameter. This non-type template parameter can then be used inside the function. We discuss this further in lesson 17.3 -- Passing and returning std::array.

Subscripting `std::array` using `operator[]` or the `at()` member function

In the prior lesson 17.1 -- Introduction to std::array, we covered that the most common way to index a `std::array` is to use the subscript operator (`operator[]`). No bounds checking is done in this case, and passing in an invalid index will result in undefined behavior.

Just like `std::vector`, `std::array` also has an `at()` member function that does subscripting with runtime bounds checking. We recommend avoiding this function since we typically want to do bounds checking before indexing, or we want compile-time bounds checking.

Both of these functions expect the index to be of type `size_type` (`std::size_t`).

If either of these functions are called with a constexpr value, the compiler will do a constexpr conversion to `std::size_t`. This isn't considered to be a narrowing conversion, so you won't run into sign problems here.

However, if either of these functions are called with a non-constexpr signed integral value, the conversion to `std::size_t` is considered narrowing and your compiler may emit a warning. We discuss this case further (using `std::vector`) in lesson 16.3 -- std::vector and the unsigned length and subscript problem.

`std::get()` does compile-time bounds checking for constexpr indices

Since the length of a `std::array` is constexpr, if our index is also a constexpr value, then the compiler should be able to validate at compile-time that our constexpr index is within the bounds of the array (and stop compilation if the constexpr index is out of bounds).

However, `operator[]` does no bounds checking by definition, and the `at()` member function only does runtime bounds checking. And function parameters can't be constexpr (even for constexpr or consteval functions), so how do we even pass a constexpr index?

To get compile-time bounds checking when we have a constexpr index, we can use the `std::get()` function template, which takes the index as a non-type template argument:

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

int main()
{
    constexpr std::array prime{ 2, 3, 5, 7, 11 };

    std::cout << std::get<3>(prime); // print the value of element with index 3
    std::cout << std::get<9>(prime); // invalid index (compile error)

    return 0;
}
```

Inside the implementation of `std::get()`, there is a static_assert that checks to ensure that the non-type template argument is smaller than the array length. If it isn't, then the static_assert will halt the compilation process with compilation error.

Since template arguments must be constexpr, `std::get()` can only be called with constexpr indices.

Quiz time

Question #1

Initialize a `std::array` with the following values: 'h', 'e', 'l', 'l', 'o'. Print the length of the array, and then use `operator[]`, `at()` and `std::get()` to print the value of the element with index 1.

The program should print:

```
The length is 5
eee
```

Show Solution