16.3 — std::vector and the unsigned length and subscript problem

In the prior lesson 16.2 -- Introduction to std::vector and list constructors, we introduced `operator[]`, which can be used to subscript an array to access an element.

In this lesson, we'll look at other ways to access array elements, as well as a few different ways to get the length of an container class (the number of elements currently in the container class).

But before we can do that, we need to discuss one major mistake that the designers of C++ made, and how it affects all the container classes in the C++ standard library.

The container length sign problem

Let's start with an assertion: the data type used for subscripting an array should match the data type used for storing the length of the array. This is so that all elements in the longest possible array can be indexed, and no more.

As Bjarne Stroustrup recalls, when the container classes in the C++ standard library was being designed (circa 1997), the designers had to choose whether to make the length (and array subscripts) signed or unsigned. They chose to make them unsigned.

The reasons given for this: the subscripts of the standard library array types can't be negative, using an unsigned type allows arrays of greater length due to the extra bit (something that was important in the 16-bit days), and range-checking the subscript requires one conditional check instead of two (since no check was needed to ensure the index was less than 0).

In retrospect, this is generally regarded as having been the wrong choice. We now understand that using unsigned values to try to enforce non-negativity doesn't work due to the implicit conversion rules (since a negative signed integer will just implicitly convert to a large unsigned integer, producing a garbage result), the extra bit of range typically isn't needed on 32-bit or 64-bit systems (since you probably aren't creating arrays with more than 2 million elements), and the commonly used `operator[]` doesn't do range-checking anyway.

In lesson 4.5 -- Unsigned integers, and why to avoid them, we discussed the reasons why we prefer to use signed values to hold quantities. We also noted that mixing signed and unsigned values is a recipe for unexpected behavior. So the fact that the standard library container classes use unsigned values for the length (and indices) is problematic, as it makes it impossible to avoid unsigned values when using these types.

For now, we are stuck with this choice and the unnecessary complexity it causes.

A review: sign conversions are narrowing conversions, except when constexpr

Before we proceed further, let's quickly recap some material from lesson 10.4 -- Narrowing conversions, list initialization, and constexpr initializers regarding sign conversions (integral conversions from signed to unsigned or vice-versa) because we'll be talking about these a lot in this chapter.

Sign conversions are considered to be narrowing conversions because a signed or unsigned type cannot hold all the values contained in the range of the opposing type. When such a conversion is attempted at runtime, the compiler will issue an error in contexts where narrowing conversions are disallowed (such as in list initialization), and may or may not issue a warning in other contexts where such a conversion is required.

For example:

```
#include <iostream>

void foo(unsigned int)
{
}

int main()
{
    int s { 5 };

    [[maybe_unused]] unsigned int u { s }; // compile error: list initialization
disallows narrowing conversion
    foo(s);                                // possible warning: copy initialization
allows narrowing conversion

    return 0;
}
```

In the above example, the initialization of variable `u` produces a compilation error because narrowing conversions are disallowed when doing list initialization. The call to `foo()` performs copy initialization, which does allow narrowing conversions, and which may or may not produce a warning depending on how aggressive the compiler is in producing sign conversion warnings. For example, both GCC and Clang will produce warnings in this case when compiler flag `-Wsign-conversion` is used.

However, if the value to be sign converted is constexpr and can be converted to an equivalent value in the opposing type, the sign conversion is *not* considered to be narrowing. This is because the compiler can guarantee that the conversion is safe, or halt the compilation process.

```
#include <iostream>

void foo(unsigned int)
{
}

int main()
{
    constexpr int s { 5 };                  // now constexpr
    [[maybe_unused]] unsigned int u { s }; // ok: s is constexpr and can be converted
safely, not a narrowing conversion
    foo(s);                                 // ok: s is constexpr and can be converted
safely, not a narrowing conversion

    return 0;
}
```

In this case, since `s` is constexpr and the value to be converted (`5`) can be represented as an unsigned value, the conversion is not considered to be narrowing and can be performed implicitly without issue.

This non-narrowing constexpr conversion (from `constexpr int` to `constexpr std::size_t`) will be something we make use of a lot.

The length and indices of `std::vector` have type `size_type`

In lesson 10.7 -- Typedefs and type aliases, we mentioned that typedefs and type aliases are often used in cases where we need a name for a type that may vary (e.g. because it is implementation-defined). For example `std::size_t` is a typedef for some large unsigned integral type, usually `unsigned long` or `unsigned long long`.

Each of the standard library container classes defines a nested typedef member named `size_type` (sometimes written as `T::size_type`), which is an alias for the type used for the length (and indices, if supported) of the container.

You'll typically see `size_type` appear in documentation and in compiler warnings/error messages. For example, this documentation for the `size()` member function of `std::vector` indicates that `size()` returns a value of `size_type`.

Related content

We cover nested typedefs in lesson 15.3 -- Nested types (member types).

`size_type` is almost always an alias for `std::size_t`, but can be overridden (in rare cases) to use a different type.

Key insight

`size_type` is a nested typedef defined in standard library container classes, used as the type for the length (and indices, if supported) of the container class.

`size_type` defaults to `std::size_t`, and since this is almost never changed, we can reasonably assume `size_type` is an alias for `std::size_t`.

For advanced readers

All of the standard library containers except `std::array` use `std::allocator` to allocate memory. For these containers, `T::size_type` is derived from the `size_type` of the allocator used. Since `std::allocator` can allocate up to `std::size_t` bytes of memory, `std::allocator<T>::size_type` is defined as `std::size_t`. Therefore, `T::size_type` defaults to `std::size_t`.

Only in cases where a custom allocator whose `T::size_type` is defined as something other than `std::size_t` will a container's `T::size_type` be something other than `std::size_t`. This is rare and something done on a per-application basis, so it's generally safe to assume `T::size_type` will be `std::size_t` unless your application is using such a custom allocator (and you will know if that is the case).

When accessing the `size_type` member of a container class, we must scope qualify it with the fully templated name of the container class. For example, `std::vector<int>::size_type`.

Getting the length of a `std::vector` using the `size()` member function or `std::size()`

We can ask a container class object for its length using the `size()` member function (which returns the length as unsigned `size_type`):

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

int main()
{
    std::vector prime { 2, 3, 5, 7, 11 };
    std::cout << "length: " << prime.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::vector` (and most other container types in C++) only has `size()`. And now you understand why the length of a container is sometimes

ambiguously called its size.

In C++17, we can also use the `std::size()` non-member function (which for container classes just calls the `size()` member function).

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

int main()
{
    std::vector prime { 2, 3, 5, 7, 11 };
    std::cout << "length: " << std::size(prime); // C++17, returns length as type
`size_type` (alias for `std::size_t`)

    return 0;
}
```

For advanced readers

Because `std::size()` can also be used on non-decayed C-style arrays, this method is sometimes favored over the using the `size()` member function (particularly when writing function templates that can accept either a container class or non-decayed C-style array argument).

If we want to use either of the above methods to store the length in a variable with a signed type, this will likely result in a signed/unsigned conversion warning or error. The simplest thing to do here is static_cast the result to the desired type:

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

int main()
{
    std::vector prime { 2, 3, 5, 7, 11 };
    int length { static_cast<int>(prime.size()) }; // static_cast return value to int
    std::cout << "length: " << length ;

    return 0;
}
```

Getting the length of a `std::vector` using `std::ssize()` C++20

C++20 introduces the `std::ssize()` non-member function, which returns the length as a large *signed* integral type (usually `std::ptrdiff_t`, which is the type normally used as the signed counterpart to `std::size_t`):

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

int main()
{
    std::vector prime{ 2, 3, 5, 7, 11 };
    std::cout << "length: " << std::ssize(prime); // 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.

If you want to use this method to store the length in a variable with a signed type, you have a couple of options.

First, because the `int` type may be smaller than the signed type returned by `std::ssize()`, if you are going to assign the length to an `int` variable, you should static_cast the result to `int` to make any such conversion explicit (otherwise you might get a narrowing conversion warning or error):

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

int main()
{
    std::vector prime{ 2, 3, 5, 7, 11 };
    int length { static_cast<int>(std::ssize(prime)) }; // static_cast return value
to int
    std::cout << "length: " << length;

    return 0;
}
```

Alternatively, you can use `auto` to have the compiler deduce the correct signed type to use for the variable:

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

int main()
{
    std::vector prime{ 2, 3, 5, 7, 11 };
    auto length { std::ssize(prime) }; // use auto to deduce signed type, as returned
by std::ssize()
    std::cout << "length: " << length;

    return 0;
}
```

Accessing array elements using `operator[]` does no bounds checking

In the prior lesson, we introduced the subscript operator (`operator[]`):

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

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

    std::cout << prime[3];  // print the value of element with index 3 (7)
    std::cout << prime[9]; // invalid index (undefined behavior)

    return 0;
}
```

`operator[]` does not do bounds checking. The index for operator[] can be non-const. We'll discuss this further in a later section.

Accessing array elements using the `at()` member function does runtime bounds checking

The array container classes support another method for accessing an array. The `at()` member function can be used to do array access with runtime bounds checking:

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

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

    std::cout << prime.at(3); // print the value of element with index 3
    std::cout << prime.at(9); // invalid index (throws exception)

    return 0;
}
```

In the above example, the call to `prime.at(3)` checks to ensure the index 3 is valid, and because it is, it returns a reference to array element 3. We can then print this value. However, the call to `prime.at(9)` fails (at runtime) because 9 is not a valid index for this array. Instead of returning a reference, the `at()` function generates an error that terminates the program.

For advanced readers

When the `at()` member function encounters an out-of-bounds index, it actually throws an exception of type `std::out_of_range`. If the exception is not handled, the program will be terminated. We cover exceptions and how to handle them in chapter 27.

Just like `operator[]`, the index passed to `at()` can be non-const.

Because it does runtime bounds checking on every call, `at()` is slower (but safer) than `operator[]`. Despite being less safe, `operator[]` is typically used over `at()`, primarily because it's better to do bounds checking before we call `at()`, so we don't call it with an invalid index in the first place.

Indexing `std::vector` with a constexpr signed int

When indexing a `std::vector` with a constexpr (signed) int, we can let the compiler implicitly convert this to a `std::size_t` without it being a narrowing conversion:

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

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

    std::cout << prime[3] << '\n';     // okay: 3 converted from int to std::size_t,
not a narrowing conversion

    constexpr int index { 3 };         // constexpr
    std::cout << prime[index] << '\n'; // okay: constexpr index implicitly converted
to std::size_t, not a narrowing conversion

    return 0;
}
```

Indexing `std::vector` with a non-constexpr value

The subscripts used to index an array can be non-const:

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

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

    std::size_t index { 3 };           // non-constexpr
    std::cout << prime[index] << '\n'; // operator[] expects an index of type
std::size_t, no conversion required

    return 0;
}
```

However, as per our best practices (4.5 -- Unsigned integers, and why to avoid them), we generally want to avoid using unsigned types to hold quantities.

When our subscript is a non-constexpr signed value value, we run into problems:

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

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

    int index { 3 };                    // non-constexpr
    std::cout << prime[index] << '\n'; // possible warning: index implicitly
converted to std::size_t, narrowing conversion

    return 0;
}
```

In this example, `index` is a non-constexpr signed int. The subscript of `operator[]` defined as part of `std::vector` has type `size_type` (an alias for `std::size_t`). Therefore, when we call `prime[index]`, our signed int must be converted to `std::size_t`.

Such a conversion should not be dangerous (because the index of a `std::vector` is expected to be non-negative, and a non-negative signed value will safely convert to an unsigned value). But when performed at runtime, this is considered to be a narrowing conversion, and your compiler should produce a warning about this being an unsafe conversion (if it doesn't, you should consider modifying your warnings so that it does).

Because array subscripting is common, and each such conversion will generate a warning, this can easily clutter up your compilation log with spurious warnings. Or, if you have "treat warning as errors" enabled, it will halt your compilation.

There are many possible ways to avoid this issue (e.g. `static_cast` your `int` to a `std::size_t` every time you index an array), but none that are convenient -- they all inevitably end up cluttering or complicating your code in some way.

The simplest thing to do in this case is use a variable of type `std::size_t` as your index, and do not use this variable for anything but indexing.

Author's note

We'll discuss additional options to address such indexing challenges in lesson 16.7 -- Arrays, loops, and sign challenge solutions.

Quiz time

Question #1

Initialize a `std::vector` with the following values: ‘h’, ‘e’, ‘l’, ‘l’, ‘o’. Then print the length of the array (use `std::size()`). Finally, print the value of the element with index 1 using the subscript operator and the `at()` member function.

The program should output the following:

```
The array has 5 elements.
ee
```

Show Solution

Question #2

a) What is `size_type` and what is it used for?

Show Solution

b) What type does `size_type` default to? Is it signed or unsigned?

Show Solution

c) Which functions to get the length of a container return `size_type`?

Show Solution