16.7 — Arrays, loops, and sign challenge solutions

In lesson 4.5 -- Unsigned integers, and why to avoid them, we noted how we generally prefer to use signed values to hold quantities, because unsigned values can act in surprising ways. However, in lesson 16.3 -- std::vector and the unsigned length and subscript problem, we discussed how `std::vector` (and other container classes) uses unsigned integral type `std::size_t` for length and indices.

This can lead to problems such as this one:

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

template <typename T>
void printReverse(const std::vector<T>& arr)
{
    for (std::size_t index{ arr.size() - 1 }; index >= 0; --index) // index is
unsigned
    {
        std::cout << arr[index] << ' ';
    }

    std::cout << '\n';
}

int main()
{
    std::vector arr{ 4, 6, 7, 3, 8, 2, 1, 9 };

    printReverse(arr);

    return 0;
}
```

This code begins by printing the array in reverse:

```
9 1 2 8 3 7 6 4
```

And then exhibits undefined behavior. It might print garbage values, or crash the application.

There are two problems here. First, our loop executes as long as `index >= 0` (or in other words, as long as `index` is positive), which is always true when `index` is unsigned. Therefore, the loop never terminates.

Second, when we decrement `index` when it has value `0`, it will wrap around to a large positive value, which we then use to index the array on the next iteration. This is an out-of-bounds index, and will cause undefined behavior. We run into the same problem if our vector is empty.

And while there are plenty of ways to work around these specific issues, these kinds of issues are magnets for bugs.

Using a signed type for a loop variable more easily avoids such problems, but has its own challenges. Here's a version of the above problem that uses a signed index:

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

template <typename T>
void printReverse(const std::vector<T>& arr)
{
    for (int index{ static_cast<int>(arr.size()) - 1}; index >= 0; --index) // index
is signed
    {
        std::cout << arr[static_cast<std::size_t>(index)] << ' ';
    }

    std::cout << '\n';
}

int main()
{
    std::vector arr{ 4, 6, 7, 3, 8, 2, 1, 9 };

    printReverse(arr);

    return 0;
}
```

While this version functions as intended, the code is also a cluttered due to the addition two static casts. `arr[static_cast<std::size_t>(index)]` is particularly hard to read. In this case, we've improved safety at a significant cost to readability.

Here's another example of using a signed index:

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

// Function template to calculate the average value in a std::vector
template <typename T>
T calculateAverage(const std::vector<T>& arr)
{
    int length{ static_cast<int>(arr.size()) };

    T average{ 0 };
    for (int index{ 0 }; index < length; ++index)
        average += arr[static_cast<std::size_t>(index)];
    average /= length;

    return average;
}

int main()
{
    std::vector testScore1 { 84, 92, 76, 81, 56 };
    std::cout << "The class 1 average is: " << calculateAverage(testScore1) << '\n';

    return 0;
}
```

The cluttering of our code with static casts is pretty terrible.

So what should we do? This is an area where there is no ideal solution.

There are many viable options here, which we'll present in order from what we believe is worst to best. You will likely encounter all of these in code written by others.

Author's note

Although we'll be discussing this in the context of `std::vector`, all of the standard library containers (e.g. `std::array`) work similarly and have the same challenges. The discussion that follows is applicable to any of them.

Leave signed/unsigned conversion warnings off

If you were wondering why signed/unsigned conversion warnings are often disabled by default, this topic is one of the key reasons. Every time we subscript a standard library container using a signed index, a sign conversion warning will be generated. This will quickly fill up your compilation log with spurious warnings, drowning out warnings that may actually be legitimate.

So one way to avoid having to deal with lots of signed/unsigned conversion warnings is to simply leave those warnings turned off.

This is the simplest solution, but not one we recommend, as this will also suppress generation of legitimate sign conversion warnings that may cause bugs if not addressed.

Using an unsigned loop variable

Many developers believe that since the standard library array types were designed to use unsigned indices, then we should use unsigned indices! This is a completely reasonable position. We just need to be extra careful that we do not run into signed/unsigned mismatches when doing so. If possible, avoid using the index loop variable for anything but indexing.

If we decide to use this approach, which unsigned type should we actually use?

In lesson 16.3 -- std::vector and the unsigned length and subscript problem, we noted that the standard library container classes define nested typedef `size_type`, which is an unsigned integral type used for array lengths and indices. The `size()` member function returns `size_type`, and `operator[]` uses `size_type` as an index, so using `size_type` as the type of your index is technically the most consistent and safe unsigned type to use (as it will work in all cases, even in the extremely rare case where `size_type` is something other than `size_t`.). For example:

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

int main()
{
	std::vector arr { 1, 2, 3, 4, 5 };

	for (std::vector<int>::size_type index { 0 }; index < arr.size(); ++index)
		std::cout << arr[index] << ' ';

	return 0;
}
```

However, using `size_type` has a major downside: because it is a nested type, to use it we have to explicitly prefix the name with the fully templated name of the container (meaning we have to type `std::vector<int>::size_type` rather than just `std::size_type`). This requires a lot of typing, is hard to read, and varies depending on the container and element type.

When used inside a function template, we can use `T` for the template arguments. But we also need to prefix the type with the `typename` keyword:

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

template <typename T>
void printArray(const std::vector<T>& arr)
{
        // typename keyword prefix required for dependent type
        for (typename std::vector<T>::size_type index { 0 }; index < arr.size();
++index)
                std::cout << arr[index] << ' ';
}

int main()
{
        std::vector arr { 9, 7, 5, 3, 1 };

        printArray(arr);

        return 0;
}
```

If you forget the `typename` keyword, your compiler will probably remind you to add it.

For advanced readers

Any name that depends on a type containing a template parameter is called a **dependent name**. Dependent names must be prefixed with the keyword `typename` in order to be used as a type.

In the above example, `std::vector<T>` is a type with a template parameter, so nested type `std::vector<T>::size_type` is a dependent name, and must be prefixed with `typename` to be used as a type.

You may occasionally see the array type aliased to make the loop easier to read:

```
using arrayi = std::vector<int>;
for (arrayi::size_type index { 0 }; index < arr.size(); ++index)
```

A more general solution is to have the compiler fetch the type of the array type object for us, so that we don't have to explicitly specify the container type or template arguments. To do so, we can use the **decltype** keyword, which returns the type of its parameter.

```
// arr is some non-reference type
for (decltype(arr)::size_type index { 0 }; index < arr.size(); ++index) //
decltype(arr) resolves to std::vector<int>
```

However, if `arr` is a reference type (e.g. an array passed by reference), the above doesn't work. We need to first remove the reference from `arr`:

```
template <typename T>
void printArray(const std::vector<T>& arr)
{
	// arr can be a reference or non-reference type
	for (typename std::remove_reference_t<decltype(arr)>::size_type index { 0 };
index < arr.size(); ++index)
		std::cout << arr[index] << ' ';
}
```

Unfortunately, this is no longer very concise or easy to remember.

Because `size_type` is almost always a typedef for `size_t`, many programmers just skip using `size_type` altogether and use the easier to remember and type `std::size_t` directly:

```
for (std::size_t index { 0 }; index < arr.size(); ++index)
```

Unless you're using custom allocators (and you probably aren't), we believe this is a reasonable approach.

Using a signed loop variable

Although it makes working with the standard library container types a bit more difficult, using a signed loop variable is consistent with the best practices employed in the rest of our code (to favor signed values for quantities). And the more we can consistently apply our best practices, the fewer errors we will have overall.

If we are going to use signed loop variables, there are three issues we need to address:

- What signed type should we use?
- Getting the length of the array as a signed value
- Converting the signed loop variable to an unsigned index

What signed type should we use?

There are three (sometimes four) good options here.

1. Unless you are working with a very large array, using `int` should be fine (particularly on architectures where int is 4 bytes). `int` is the default signed integral type we use for everything when we don't really care about the type otherwise, and there's little reason to do otherwise here.
2. If you are dealing with very large arrays, or if you want to be a bit more defensive, you can use the strangely named `std::ptrdiff_t`. This typedef is often used as the signed counterpart to `std::size_t`.
3. Because `std::ptrdiff_t` has a weird name, another good approach is to define your own type alias for indices:

```
using Index = std::ptrdiff_t;

// Sample loop using index
for (Index index{ 0 }; index < static_cast<Index>(arr.size()); ++index)
```

We'll show a full example of this in the next section.

Defining your own type alias also has a potential future benefit: if the C++ standard library ever releases a type designed to be used as a signed index, it will be easy to either modify `Index` to alias that type, or to find/replace `Index` with whatever that type is named.

4. In cases where you can derive the type of your loop variable from the initializer, you can use `auto` to have the compiler deduce the type:

```
for (auto index{ static_cast<std::ptrdiff_t>(arr.size())-1 }; index >= 0; --index)
```

In C++23, the `Z` suffix can be used to define a literal of the type that is the signed counterpart to `std::size_t` (probably `std::ptrdiff_t`):

```
for (auto index{ 0Z }; index < static_cast<std::ptrdiff_t>(arr.size()); ++index)
```

Getting the length of an array as a signed value

1. Pre-C++20, the best option is to `static_cast` the return value of the `size()` member function or `std::size()` to a signed type:

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

using Index = std::ptrdiff_t;

int main()
{
    std::vector arr{ 9, 7, 5, 3, 1 };

    for (auto index{ static_cast<Index>(arr.size())-1 }; index >= 0; --index)
        std::cout << arr[static_cast<std::size_t>(index)] << ' ';

    return 0;
}
```

That way, the unsigned value returned by `arr.size()` will be converted to a signed type, so our comparison operator will have two signed operands. And because signed indices won't overflow when they go negative, we don't have the wrap-around problem we ran into when using unsigned indices.

The downside of this approach is that it clutters up our loop, making it harder to read. We can address this by moving the length out of the loop:

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

using Index = std::ptrdiff_t;

int main()
{
    std::vector arr{ 9, 7, 5, 3, 1 };

    auto length{ static_cast<Index>(arr.size()) };
    for (auto index{ length-1 }; index >= 0; --index)
        std::cout << arr[static_cast<std::size_t>(index)] << ' ';

    return 0;
}
```

2. In C++20, use `std::ssize()`:

If you want more evidence that the designers of C++ now believe that signed indices are the way to go, consider the introduction of `std::ssize()` in C++20. This function returns the size of an array type as a signed type (likely `ptrdiff_t`).

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

int main()
{
    std::vector arr{ 9, 7, 5, 3, 1 };

    for (auto index{ std::ssize(arr)-1 }; index >= 0; --index) // std::ssize
introduced in C++20
        std::cout << arr[static_cast<std::size_t>(index)] << ' ';

    return 0;
}
```

Converting the signed loop variable to an unsigned index

Once we have a signed loop variable, we're going to run into implicit sign conversion warnings whenever we try to use that signed loop variable as an index. So we need some way to convert our signed loop variable to an unsigned value wherever we intend to use it as an index.

1. The obvious option is to static cast our signed loop variable into an unsigned index. We show this in the prior example. Unfortunately, we need to do this everywhere we subscript the array, and it makes our array indices hard to read.
2. Use a conversion function with a short name:

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

using Index = std::ptrdiff_t;

constexpr std::size_t toUZ(Index value)
{
    return static_cast<std::size_t>(value);
}

int main()
{
    std::vector arr{ 9, 7, 5, 3, 1 };

    auto length { static_cast<Index>(arr.size()) };  // in C++20, prefer std::ssize()
    for (auto index{ length-1 }; index >= 0; --index)
        std::cout << arr[toUZ(index)] << ' '; // use toUZ() to avoid sign conversion
warning

    return 0;
}
```

In the above example, we've created a function named `toUZ()` that is designed to convert values of type `Index` to values of type `std::size_t`. This allows us to index our array as `arr[toUZ(index)]`, which is pretty readable.

3. Use a custom view

In prior lessons, we discussed how `std::string` owns a string, whereas `std::string_view` is a view into a string that exists elsewhere. One of the neat things about `std::string_view` is how it can view different types of strings (C-style string literals, `std::string`, and other `std::string_view`) but keeps a consistent interface for us to use.

While we can't modify the standard library containers to accept a signed integral index, we can create our own custom view class to "view" a standard library container class. And in doing so, we can define our own interface to work however we want.

In the following example, we define a custom view class that can view any standard library container that supports indexing. Our interface will do two things:

- Allow us to access elements using `operator[]` with a signed integral type.
- Get the length of the container as a signed integral type (since `std::ssize()` is only available on C++20).

This uses operator overloading, a topic we haven't covered yet, in order to implement `operator[]`. You don't need to know how `SignedArrayView` is implemented in order to use it.

SignedArrayView.h:

```
#ifndef SIGNED_ARRAY_VIEW_H
#define SIGNED_ARRAY_VIEW_H

#include <cstddef> // for std::size_t and std::ptrdiff_t

// SignedArrayView provides a view into a container that supports indexing
// allowing us to work with these types using signed indices
template <typename T>
class SignedArrayView // C++17
{
private:
    T& m_array;

public:
    using Index = std::ptrdiff_t;

    SignedArrayView(T& array)
        : m_array{ array } {}

    // Overload operator[] to take a signed index
    constexpr auto& operator[](Index index) { return m_array[static_cast<typename
T::size_type>(index)]; }
    constexpr const auto& operator[](Index index) const { return
m_array[static_cast<typename T::size_type>(index)]; }
    constexpr auto ssize() const { return static_cast<Index>(m_array.size()); }
};

#endif
```

main.cpp:

```
#include <iostream>
#include <vector>
#include "SignedArrayView.h"

int main()
{
    std::vector arr{ 9, 7, 5, 3, 1 };
    SignedArrayView sarr{ arr }; // Create a signed view of our std::vector

    for (auto index{ sarr.ssize() - 1 }; index >= 0; --index)
        std::cout << sarr[index] << ' '; // index using a signed type

    return 0;
}
```

The only sane choice: avoid indexing altogether!

All of the options presented above have their own downsides, so it’s hard to recommend one approach over the other. However, there is a choice that is far more sane than the others: avoid indexing with integral values altogether.

C++ provides several other methods for traversing through arrays that do not use indices at all. And if we don't have indices, then we don't run into all of these signed/unsigned conversion issues.

Two common methods for array traversal without indices include range-based for loops, and iterators.

Related content

We cover ranged-for loops in the next lesson (16.8 -- Range-based for loops (for-each)). We cover iterators in upcoming lesson 18.2 -- Introduction to iterators.

If you're only using the index variable to traverse the array, then prefer a method that does not use indices.

Best practice

Avoid array indexing with integral values whenever possible.