16.2 — Introduction to std::vector and list constructors

In the previous lesson 16.1 -- Introduction to containers and arrays, we introduced both containers and arrays. In this lesson, we'll introduce the array type that we'll be focused on for the rest of the chapter: `std::vector`. We'll also solve one part of the scalability challenge we introduced last lesson.

Introduction to `std::vector`

`std::vector` is one of the container classes in the C++ standard containers library that implements an array. `std::vector` is defined in the <vector> header as a class template, with a template type parameter that defines the type of the elements. Thus, `std::vector<int>` declares a `std::vector` whose elements are of type `int`.

Instantiating a `std::vector` object is straightforward:

```
#include <vector>

int main()
{
	// Value initialization (uses default constructor)
	std::vector<int> empty{}; // vector containing 0 int elements

	return 0;
}
```

Variable `empty` is defined as a `std::vector` whose elements have type `int`. Because we've used value initialization here, our vector will start empty (that is, with no elements).

A vector with no elements may not seem useful now, but we'll encounter this again in future lessons (particularly 16.11 -- std::vector and stack behavior).

Initializing a `std::vector` with a list of values

Since the goal of a container is to manage a set of related values, most often we will want to initialize our container with those values. We can do this by using list initialization with the specific initialization values we want. For example:

```
#include <vector>

int main()
{
	// List construction (uses list constructor)
	std::vector<int> primes{ 2, 3, 5, 7 };          // vector containing 4 int elements with values 2, 3, 5, and 7
	std::vector vowels { 'a', 'e', 'i', 'o', 'u' }; // vector containing 5 char elements with values 'a', 'e', 'i', 'o', and 'u'.  Uses CTAD (C++17) to deduce element type char (preferred).

	return 0;
}
```

With `primes`, we're explicitly specifying that we want a `std::vector` whose elements have type `int`. Because we've supplied 4 initialization values, `primes` will contain 4 elements whose values are `2`, `3`, `5`, and `7`.

With `vowels`, we haven't explicitly specified an element type. Instead, we're using C++17's CTAD (class template argument deduction) to have the compiler deduce the element type from the initializers. Because we've supplied 5 initialization values, `vowels` will contain 5 elements whose values are `'a'`, `'e'`, `'i'`, `'o'`, and `'u'`.

List constructors and initializer lists

Let's talk about how the above works in a little more detail.

In lesson 13.8 -- Struct aggregate initialization, we defined an initializer list as a braced list of comma-separated values (e.g. `{ 1, 2, 3 }`).

Containers typically have a special constructor called a **list constructor** that allows us to construct an instance of the container using an initializer list. The list constructor does three things:

- Ensures the container has enough storage to hold all the initialization values (if needed).
- Sets the length of the container to the number of elements in the initializer list (if needed).
- Initializes the elements to the values in the initializer list (in sequential order).

Thus, when we provide a container with an initializer list of values, the list constructor is called, and the container is constructed using that list of values!

Best practice

Use list initialization with an initializer list of values to construct a container with those element values.

Related content

We discuss adding list constructors to your own program-defined classes in lesson 23.7 -- std::initializer_list.

Accessing array elements using the subscript operator (operator[])

So now that we've created an array of elements… how do we access them?

Let's use an analogy for a moment. Consider a set of identical mailboxes, side by side. To make it easier to identify the mailboxes, each mailbox has a number painted on the front. The first mailbox has number 0, the second has number 1, etc… So if you were told to put something in mailbox number 0, you'd know that meant the first mailbox.

In C++, the most common way to access array elements is by using the name of the array along with the subscript operator (`operator[]`). To select a specific element, inside the square brackets of the subscript operator, we provide an integral value that identifies which element we want to select. This integral value is called a **subscript** (or informally, an **index**). Much like our mailboxes, the first element is accessed using index 0, the second is accessed using index 1, etc…

For example, `primes[0]` will return the element with index `0` (the first element) from the `prime` array. The subscript operator returns a reference to the actual element, not a copy. Once we've accessed an array element, we can use it just like a normal object (e.g. assign a value to it, output it, etc…)

Because the indexing starts with 0 rather than 1, we say arrays in C++ are **zero-based**. This can be confusing because we're used to counting objects starting from 1. This also can cause some ambiguity, because when we talk about array element 1, it may not be clear whether we're talking about the first array element (with index 0) or the second array element (with index 1).

Here's an example:

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

int main()
{
    std::vector primes { 2, 3, 5, 7, 11 }; // hold the first 5 prime numbers (as int)

    std::cout << "The first prime number is: " << primes[0] << '\n';
    std::cout << "The second prime number is: " << primes[1] << '\n';
    std::cout << "The sum of the first 5 primes is: " << primes[0] + primes[1] +
primes[2] + primes[3] + primes[4] << '\n';

    return 0;
}
```

This prints:

```
The first prime number is: 2
The second prime number is: 3
The sum of the first 5 primes is: 28
```

By using arrays, we no longer have to define 5 differently-named variables to hold our 5 prime values. Instead, we can define a single array (`primes`) with 5 elements, and just change the value of the index to access different elements!

We'll talk more about `operator[]` and some other methods for accessing array elements in the next lesson 16.3 -- std::vector and the unsigned length and subscript problem.

Subscript out of bounds

When indexing an array, the index provided must select a valid element of the array. That is, for an array of length N, the subscript must be a value between 0 and N-1 (inclusive).

`operator[]` does not do any kind of **bounds checking**, meaning it does not check to see whether the index is within the bounds of 0 to N-1 (inclusive). Passing an invalid index to `operator[]` will return in undefined behavior.

It is fairly easy to remember not to use negative subscripts. It is less easy to remember that there is no element with index N! The last element of the array has index N-1, so using index N will cause the compiler to try to access an element that is one-past the end of the array.

Tip

In an array with N elements, the first element has index 0, the second has index 1, and the last element has index N-1. There is no element with index N!

Using N as a subscript will cause undefined behavior (as this is actually attempting to access the N+1th element, which isn't part of the array).

Tip

Some compilers (like Visual Studio) provide a runtime assert that the index is valid. In such cases, if an invalid index is provided in debug mode, the program will assert out. In release mode, the assert is compiled out so there is no performance penalty.

Arrays are contiguous in memory

One of the defining characteristics of arrays is that the elements are always allocated contiguously in memory, meaning the elements are all adjacent in memory (with no gaps between them).

As an illustration of this:

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

int main()
{
    std::vector primes { 2, 3, 5, 7, 11 }; // hold the first 5 prime numbers (as int)

    std::cout << "An int is " << sizeof(int) << " bytes\n";
    std::cout << &(primes[0]) << '\n';
    std::cout << &(primes[1]) << '\n';
    std::cout << &(primes[2]) << '\n';

    return 0;
}
```

On the author's machine, one run of the above program produced the following result:

```
An int is 4 bytes
00DBF720
00DBF724
00DBF728
```

You'll note that the memory addresses for these int elements are 4 bytes apart, the same as the size of an `int` on the author's machine.

This means arrays do not have any per-element overhead. It also allows the compiler to quickly calculate the address of any element in the array.

Related content

We'll talk about the math behind subscripting in lesson 17.9 -- Pointer arithmetic and subscripting

Arrays are one of the few container types that allow **random access**, meaning every element in the container can be accessed directly and with equal speed, regardless of the number of elements in the container. The ability to directly access any element quickly is the primary reason arrays are the container of choice.

Constructing a `std::vector` of a specific length

Consider the case where we want the user to input 10 values that we'll store in a `std::vector`. In this case, we need a `std::vector` of length 10 before we have any values to put in the `std::vector`. How do we address this?

We could create a `std::vector` and initialize it with an initializer list with 10 placeholder values:

```
std::vector<int> data { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }; // vector containing 10 int values
```

But that's bad for a lot of reasons. It requires a lot of typing. It's not easy to see how many initializers there are. And it's not easy to update if we decide we want a different number of values later.

Fortunately, `std::vector` has an explicit constructor (`explicit std::vector<T>(int)`) that takes a single int value defining the length of the `std::vector` to construct:

```
std::vector<int> data( 10 ); // vector containing 10 int elements, value-initialized to 0
```

Each of the created elements are value-initialized, which for `int` does zero-initialization (and for class types calls the default constructor).

However, there is one non-obvious thing about using this constructor: it must be called using direct initialization.

List constructors take precedence over other constructors

To understand why the previous constructor must be called using direct initialization, consider this definition:

```
std::vector<int> data{ 10 }; // what does this do?
```

There are two different constructors that match this initialization:

- `{ 10 }` can be interpreted as an initializer list, and matched with the list constructor to construct a vector of length 1 with value 10.
- `{ 10 }` can be interpreted as a single braced initialization value, and matched with the `std::vector<T>(int)` constructor to construct a vector of length 10 with elements value-initialized to 0.

Normally when a class type definition matches more than one constructor, the match is considered ambiguous and a compilation error results. However, C++ has a special rule for this case: A matching list constructor will be selected over other matching constructors. Without this rule, a list constructor would result in an ambiguous match with any constructor that took arguments of a single type.

Since `{ 10 }` can be interpreted as an initializer list and `std::vector` has a list constructor, the list constructor takes precedence in this case.

Key insight

When constructing a class type object, a matching list constructor is selected over other matching constructors.

To help clarify what happens in various initialization cases further, let's look at similar cases using copy, direct, and list initialization:

```
// Copy init
std::vector<int> v1 = 10;     // 10 not an initializer list, copy init won't match
explicit constructor: compilation error

// Direct init
std::vector<int> v2(10);      // 10 not an initializer list, matches explicit single-
argument constructor

// List init
std::vector<int> v3{ 10 };    // { 10 } interpreted as initializer list, matches list
constructor

// Copy list init
std::vector<int> v4 = { 10 }; // { 10 } interpreted as initializer list, matches list
constructor
std::vector<int> v5({ 10 });  // { 10 } interpreted as initializer list, matches list
constructor
```

In case `v1`, the initialization value of `10` is not an initializer list, so the list constructor isn't a match. The single-argument constructor `explicit std::vector<T>(int)` won't match either because copy initialization won't match explicit constructors. Since no constructors match, this is a compilation error.

In case `v2`, the initialization value of `10` is not an initializer list, so the list constructor isn't a match. The single-argument constructor `explicit std::vector<T>(int)` is a match, so the single-argument constructor is selected.

In case `v3` (list initialization), `{ 10 }` can be matched with the list constructor or `explicit std::vector<T>(int)`. The list constructor takes precedence over other matching constructors and is selected.

In case `v4` (copy list initialization), `{ 10 }` an be matched with the list constructor (which is a non-explicit constructor, so can be used with copy initialization). The list constructor is selected.

Case `v5` surprisingly is an alternate syntax for copy list initialization (not direct initialization), and is the same as `v4`.

This is one of the warts of C++ initialization: `{ 10 }` will match a list constructor if one exists, or a single-argument constructor if a list constructor doesn't exist. This means which behavior you get depends on whether a list constructor exists! You can generally assume containers have list constructors.

To summarize, list initializers are generally designed to allow us to initialize a container with a list of element values, and should be used for that purpose. That is what we want the majority of the time anyway. Therefore, `{ 10 }` is appropriate if `10` is meant to be an element value. If `10` is meant to be an argument to a non-list constructor of a container, use direct initialization.

Best practice

When constructing a container (or any type that has a list constructor) with initializers that are not element values, use direct initialization.

Tip

When a `std::vector` is a member of a class type, it is not obvious how to provide a default initializer that sets the length of a `std::vector` to some initial value:

```
#include <vector>

struct Foo
{
    std::vector<int> v1(8); // compile error: direct initialization not allowed for
member default initializers
};
```

When providing a default initializer for a member of a class type:

- We must use either copy initialization or list initialization.
- CTAD is not allowed (so we must explicitly specify the element type).

The answer is as follows:

```
struct Foo
{
    std::vector<int> v{ std::vector<int>(8) }; // ok
};
```

This creates a `std::vector` with a capacity of 8, and then uses that as the initializer for `v`.

Const and constexpr `std::vector`

Objects of type `std::vector` can be made `const`:

```
#include <vector>

int main()
{
    const std::vector<int> prime { 2, 3, 5, 7, 11 }; // prime and its elements cannot
be modified

    return 0;
}
```

A `const std::vector` must be initialized, and then cannot be modified. The elements of such a vector are treated as if they were const.

The elements of a non-const `std::vector` must be non-const. Thus, the following is not permitted:

```
#include <vector>

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

One of the biggest downsides of `std::vector` is that it cannot be made `constexpr`. If you need a `constexpr` array, use `std::array`.

Related content

We cover `std::array` in lesson 17.1 -- Introduction to std::array.

Why is it called a vector?

When people use the term “vector” in conversation, they typically mean a geometric vector, which is an object with a magnitude and direction. So how did `std::vector` get its name when it’s not a geometric vector?

In the book “From Mathematics to Generic Programming”, Alexander Stepanov wrote, “The name vector in STL was taken from the earlier programming languages Scheme and Common Lisp. Unfortunately, this was inconsistent with the much older meaning of the term in mathematics… this data structure should have been called array. Sadly, if you make a mistake and violate these principles, the result might stay around for a long time.”

So, basically, `std::vector` is misnamed, but it's too late to change it now.

Quiz time

Question #1

Define a `std::vector` using CTAD and initialize it with the first 5 positive square numbers (1, 4, 9, 16, and 25).

Show Solution

Question #2

What's the behavioral difference between these two definitions?

```
std::vector<int> v1 { 5 };
std::vector<int> v2 ( 5 );
```

Show Solution

Question #3

Define a `std::vector` (using an explicit template type argument) to hold the high temperature (to the nearest tenth of a degree) for each day of a year (assume 365 days in a year).

Show Solution

Question #4

Using a `std::vector`, write a program that asks the user to enter 3 integral values. Print the sum and product of those values.

The output should match the following:

```
Enter 3 integers: 3 4 5
The sum is: 12
The product is: 60
```

Show Solution