17.1 — Introduction to std::array

ALEX DECEMBER 28, 2023

In lesson 16.1 -- Introduction to containers and arrays (https://www.learncpp.com/cpp-tutorial/introduction-to-containers-and-arrays/), we introduced containers and arrays. To summarize:

- Containers provide storage for a collection of unnamed objects (called elements).
- Arrays allocate their elements contiguously in memory, and allow fast, direct access to any element via subscripting.
- C++ has three different array types that are commonly used: `std::vector`, `std::array`, and C-style arrays.

In lesson 16.10 -- std::vector resizing and capacity (https://www.learncpp.com/cpp-tutorial/stdvector-resizing-and-capacity/), we mentioned that arrays fall into two categories:

- Fixed-size arrays (also called fixed-length arrays) require that the length of the array be known at the point of instantiation, and that length cannot be changed afterward. C-style arrays and `std::array` are both fixed-size arrays.
- Dynamic arrays can be resized at runtime. `std::vector` is a dynamic array.

In the previous chapter, we focused on `std::vector`, as it is fast, comparatively easy to use, and versatile. This makes it our go-to type when we need an array container.

So why not use dynamic arrays for everything?

Dynamic arrays are powerful and convenient, but like everything in life, they make some tradeoffs for the benefits they offer.

- `std::vector` is slightly less performant than the fixed-size arrays. In most cases you probably won't notice the difference (unless you're writing sloppy code that causes lots of inadvertent reallocations).
- `std::vector` only supports `constexpr` in very limited contexts.

In modern C++, it is really this latter point that's significant. Constexpr arrays offer the ability to write code that is more robust, and can also be optimized more highly by the compiler. Whenever we can use a constexpr array, we should -- and if we need a constexpr array, `std::array` is the container class we should be using.

Best practice

Use `std::array` for constexpr arrays, and `std::vector` for non-constexpr arrays.

Defining a `std::array`

`std::array` is defined in the <array> header. It is designed to work similarly to `std::vector`, and as you'll see, there are more similarities than differences between the two.

One difference is in how we declare a `std::array`:

```
#include <array>  // for std::array
#include <vector> // for std::vector

int main()
{
    std::array<int, 5> a {};  // a std::array of 5 ints

    std::vector<int> b(5);    // a std::vector of 5 ints (for comparison)

    return 0;
}
```

Our `std::array` declaration has two template arguments. The first (`int`) is a type template argument defining the type of the array element. The second (`5`) is an integral non-type template argument defining the array length.

Related content

We cover non-type template parameters in lesson 11.9 -- Non-type template parameters (https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/).

The length of a `std::array` must be a constant expression

Unlike a `std::vector`, which can be resized at runtime, the length of a `std::array` must be a constant expression. Most often, the value provided for the length will be an integer literal, constexpr variable, or an unscoped enumerator.

```
#include <array>

int main()
{
    std::array<int, 7> a {}; // Using a literal constant

    constexpr int len { 8 };
    std::array<int, len> b {}; // Using a constexpr variable

    enum Colors
    {
        red,
        green,
        blue,
        max_colors
    };

    std::array<int, max_colors> c {}; // Using an enumerator

#define DAYS_PER_WEEK 7
    std::array<int, DAYS_PER_WEEK> d {}; // Using a macro (don't do this, use a constexpr variable instead)

    return 0;
}
```

Note that non-const variables and runtime constants cannot be used for the length:

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

void foo(const int length) // length is a runtime constant
{
    std::array<int, length> e {}; // error: length is not a constant expression
}

int main()
{
    // using a non-const variable
    int numStudents{};
    std::cin >> numStudents; // numStudents is non-constant

    std::array<int, numStudents> {}; // error: numStudents is not a constant expression

    foo(7);

    return 0;
}
```

Aggregate initialization of a `std::array`

Perhaps surprisingly, `std::array` is an aggregate. This means it has no constructors, and instead is initialized using aggregate initialization. As a quick recap, aggregate initialization allows us to directly initialize the members of aggregates. To do this, we provide an initializer list, which is a brace-enclosed list of comma-separated initialization values.

Related content

We covered aggregate initialization for structs in lesson 13.6 -- Struct aggregate initialization (https://www.learncpp.com/cpp-tutorial/struct-aggregate-initialization/).

```
#include <array>

int main()
{
    std::array<int, 6> fibonnaci = { 0, 1, 1, 2, 3, 5 }; // copy-list initialization using braced list
    std::array<int, 5> prime { 2, 3, 5, 7, 11 };         // list initialization using braced list (preferred)

    return 0;
}
```

Each of these initialization forms initializes the array members in sequence, starting with element 0.

If a `std::array` is defined without an initializer, the elements will be default initialized. In most cases, this will result in elements being left uninitialized.

Because we generally want our elements to be initialized, `std::array` should be value initialized (using empty braces) when defined with no initializers.

```
#include <array>
#include <vector>

int main()
{
    std::array<int, 5> a;   // Members default initialized (int elements are left uninitialized)
    std::array<int, 5> b{}; // Members value initialized (int elements are zero initialized) (preferred)

    std::vector<int> v(5);  // Members value initialized (int elements are zero initialized) (for comparison)

    return 0;
}
```

If more initializers are provided in an initializer list than the defined array length, the compiler will error. If fewer initializers are provided in an initializer list than the defined array length, the remaining elements without initializers are value initialized:

```
#include <array>

int main()
{
    std::array<int, 4> a { 1, 2, 3, 4, 5 }; // compile error: too many initializers
    std::array<int, 4> b { 1, 2 };          // arr[2] and arr[3] are value initialized

    return 0;
}
```

Const and constexpr `std::array`

A `std::array` can be const:

```
#include <array>

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

    return 0;
}
```

Even though the elements of a `const std::array` are not explicitly marked as const, they are still treated as const (because the whole array is const).

`std::array` also has full support for constexpr:

```
#include <array>

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

    return 0;
}
```

This support for constexpr is the key reason to use `std::array`.

Best practice

Define your `std::array` as constexpr whenever possible. If your `std::array` is not constexpr, consider using a `std::vector` instead.

Class template argument deduction (CTAD) for `std::array` C++17

Using CTAD (class template argument deduction) in C++17, we can have the compiler deduce both the element type and the array length of a `std::array` from a list of initializers:

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

int main()
{
    constexpr std::array a1 { 9, 7, 5, 3, 1 }; // The type is deduced to std::array<int, 5>
    constexpr std::array a2 { 9.7, 7.31 };     // The type is deduced to std::array<double, 2>

    return 0;
}
```

We favor this syntax whenever practical. If your compiler is not C++17 capable, you'll need to explicitly provide the type and length template arguments.

Best practice

Use class template argument deduction (CTAD) to have the compiler deduce the type and length of a `std::array` from its initializers.

C++ does not support partial omission of template parameters (as of C++20), so there is no way to use a core language feature to omit just the length or just the type of a `std::array`:

```
#include <iostream>

int main()
{
    constexpr std::array<int> a2 { 9, 7, 5, 3, 1 };     // error: too few template arguments (length missing)
    constexpr std::array<5> a2 { 9, 7, 5, 3, 1 };       // error: too few template arguments (type missing)

    return 0;
}
```

Omitting just the array length using `std::to_array` C++20

Since C++20, it is possible to omit the array length of a `std::array` by using the `std::to_array` helper function:

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

int main()
{
    constexpr auto myArray1 { std::to_array<int, 5>({ 9, 7, 5, 3, 1 }) }; // Specify type and size
    constexpr auto myArray2 { std::to_array<int>({ 9, 7, 5, 3, 1 }) };    // Specify type only, deduce size
    constexpr auto myArray3 { std::to_array({ 9, 7, 5, 3, 1 }) };         // Deduce type and size

    return 0;
}
```

Unfortunately, using `std::to_array` is more expensive than creating a `std::array` directly, because it involves creation of a temporary `std::array` that is then used to copy initialize our desired `std::array`. For this reason, `std::to_array` should only be used in cases where the type can't be effectively determined from the initializers, and should be avoided when an array is created many times (e.g. inside a loop).

Accessing array elements using `operator[]`

Just like a `std::vector`, the most common way to access elements of a `std::array` is by using the subscript operator (`operator[]`):

```
#include <array> // for std::array
#include <iostream>

int main()
{
    constexpr std::array<int, 5> 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;
}
```

As a reminder, `operator[]` does not do bounds checking. If an invalid index is provided, undefined behavior will result.

We'll discuss a few other ways to index a `std::array` in the next lesson.

Quiz time

Question #1

What type of initialization does `std::array` use?

Show Solution (javascript:void(0))

Why should you explicitly value-initialize a `std::array` if you are not providing initialization values?

Show Solution (javascript:void(0))

Question #2

Define a `std::array` that will hold the high temperature for each day of the year (to the nearest tenth of a degree).

Show Solution (javascript:void(0))

Question #3

Initialize a `std::array` with the following values: 'h', 'e', 'l', 'l', 'o'. Print the value of the element with index 1.

Show Solution (javascript:void(0))

Next lesson
17.2 std::array length and indexing

Back to table of contents

Previous lesson
16.x Chapter 16 summary and quiz

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/ are connected to your provided email address.

489 COMMENTS

Newest