10.22 — An introduction to std::array

ALEX AUGUST 30, 2021

In previous lessons, we've talked at length about fixed and dynamic arrays. Although both are built right into the C++ language, they both have downsides: Fixed arrays decay into pointers, losing the array length information when they do, and dynamic arrays have messy deallocation issues and are challenging to resize without error.

To address these issues, the C++ standard library includes functionality that makes array management easier, `std::array` and `std::vector`. We'll examine `std::array` in this lesson, and `std::vector` in the next.

An introduction to std::array

`std::array` provides fixed array functionality that won't decay when passed into a function. `std::array` is defined in the `<array>` header, inside the `std` namespace.

Declaring a `std::array` variable is easy:

```
#include <array>

std::array<int, 3> myArray; // declare an integer array with length 3
```

Just like the native implementation of fixed arrays, the length of a `std::array` must be known at compile time.

`std::array` can be initialized using initializer lists or list initialization:

```
std::array<int, 5> myArray = { 9, 7, 5, 3, 1 }; // initializer list
std::array<int, 5> myArray2 { 9, 7, 5, 3, 1 }; // list
```

Unlike built-in fixed arrays, with std::array you can not omit the array length when providing an initializer:

```
std::array<int, > myArray { 9, 7, 5, 3, 1 }; // illegal, array length must be provided
std::array<int> myArray { 9, 7, 5, 3, 1 }; // illegal, array length must be provided
```

However, since C++17, it is allowed to omit the type and size. They can only be omitted together, but not one or the other, and only if the array is explicitly initialized.

```
std::array myArray { 9, 7, 5, 3, 1 }; // The type is deduced to std::array<int, 5>
std::array myArray { 9.7, 7.31 }; // The type is deduced to std::array<double, 2>
```

We favor this syntax rather than typing out the type and size at the declaration. If your compiler is not C++17 capable, you need to use the explicit syntax instead.

```
// std::array myArray { 9, 7, 5, 3, 1 }; // Since C++17
std::array<int, 5> myArray { 9, 7, 5, 3, 1 }; // Before C++17

// std::array myArray { 9.7, 7.31 }; // Since C++17
std::array<double, 2> myArray { 9.7, 7.31 }; // Before C++17
```

Since C++20, it is possible to specify the element type but omit the array length. This makes creation o `std::array` a little more like creation of C-style arrays. To create an array with a specific type and deduced size, we use the `std::to_array` function:

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

Unfortunately, `std::to_array` is more expensive than creating a `std::array` directly, because it actually copies all elements from a C-style array to a `std::array`. For this reason, `std::to_array` should be avoided when the array is created many times (e.g. in a loop).

You can also assign values to the array using an initializer list

```
std::array<int, 5> myArray;
myArray = { 0, 1, 2, 3, 4 }; // okay
myArray = { 9, 8, 7 }; // okay, elements 3 and 4 are set to zero!
myArray = { 0, 1, 2, 3, 4, 5 }; // not allowed, too many elements in initializer list!
```

Accessing `std::array` values using the subscript operator works just like you would expect:

```
std::cout << myArray[1] << '\n';
myArray[2] = 6;
```

Just like built-in fixed arrays, the subscript operator does not do any bounds-checking. If an invalid index is provided, bad things will probably happen.

`std::array` supports a second form of array element access (the `at()` function) that does bounds checking:

```
std::array myArray { 9, 7, 5, 3, 1 };
myArray.at(1) = 6; // array element 1 is valid, sets array element 1 to value 6
myArray.at(9) = 10; // array element 9 is invalid, will throw a runtime error
```

In the above example, the call to `myArray.at(1)` checks to ensure the index 1 is valid, and because it is, it returns a reference to array element 1. We then assign the value of 6 to this. However, the call to `myArray.at(9)` fails because array element 9 is out of bounds for the array. Instead of returning a reference, the `at()` function throws an error that terminates the program (note: It's actually throwing an exception of type `std::out_of_range` -- we cover exceptions in chapter 14). Because it does bounds checking, `at()` is slower (but safer) than `operator[]`.

`std::array` will clean up after itself when it goes out of scope, so there's no need to do any kind of manual cleanup.

Size and sorting

The `size()` function can be used to retrieve the length of the `std::array`:

```
std::array myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };
std::cout << "length: " << myArray.size() << '\n';
```

This prints:

```
length: 5
```

Because `std::array` doesn't decay to a pointer when passed to a function, the `size()` function will work even if you call it from within a function:

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

void printLength(const std::array<double, 5>& myArray)
{
    std::cout << "length: " << myArray.size() << '\n';
}

int main()
{
    std::array myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };

    printLength(myArray);

    return 0;
}
```

This also prints:

```
length: 5
```

Note that the standard library uses the term "size" to mean the array length — do not get this confused with the results of `sizeof()` on a native fixed array, which returns the actual size of the array in memory (the size of an element multiplied by the array length). Yes, this nomenclature is inconsistent.

Also note that we passed `std::array` by (`const`) reference. This is to prevent the compiler from making a copy of the `std::array` when the `std::array` was passed to the function (for performance reasons).

Best practice

Always pass `std::array` by reference or `const` reference

Because the length is always known, range-based for-loops work with `std::array`:

```
std::array myArray{ 9, 7, 5, 3, 1
};

for (int element : myArray)
    std::cout << element << ' ';
```

You can sort `std::array` using `std::sort`, which lives in the `<algorithm>` header:

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

int main()
{
    std::array myArray { 7, 3, 1, 9, 5 };
    std::sort(myArray.begin(), myArray.end()); // sort the array forwards
//  std::sort(myArray.rbegin(), myArray.rend()); // sort the array
backwards

    for (int element : myArray)
        std::cout << element << ' ';

    std::cout << '\n';

    return 0;
}
```

This prints:

```
1 3 5 7 9
```

The sorting function uses iterators, which is a concept we haven't covered yet, so for now you can treat the parameters to `std::sort()` as a bit of magic. We'll explain them later.

Passing std::array of different lengths to a function

With a std::array, the element type and array length are part of the type information. Therefore, when we use a std::array as a function parameter, we have to specify the element type and array length:

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

void printArray(const std::array<int, 5>&
myArray)
{
    for (auto element : myArray)
        std::cout << element << ' ';
    std::cout << '\n';
}

int main()
{
    std::array myArray5{ 9.0, 7.2, 5.4, 3.6, 1.8
};
    printArray(myArray5);

    return 0;
}
```

The downside is that this limits our function to only handling arrays of this specific type and length. But what if we want to have our function handle arrays of different element types or lengths? We'd have to create a copy of the function for each different element type and/or array length we want to use. That's a lot of duplication.

Fortunately, we can have C++ do this for us, using templates. We can create a template function that parameterizes part or all of the type information, and then C++ will use that template to create "real" functions (with actual types) as needed.

```cpp
#include <array>
#include <cstddef>
#include <iostream>

// printArray is a template function
template <typename T, std::size_t size> // parameterize the element type and
size
void printArray(const std::array<T, size>& myArray)
{
    for (auto element : myArray)
        std::cout << element << ' ';
    std::cout << '\n';
}

int main()
{
    std::array myArray5{ 9.0, 7.2, 5.4, 3.6, 1.8 };
    printArray(myArray5);

    std::array myArray7{ 9.0, 7.2, 5.4, 3.6, 1.8, 1.2, 0.7 };
    printArray(myArray7);

    return 0;
}
```

Related content

We cover function templates in lesson8.13 -- Function templates.

Manually indexing std::array via size_type

Pop quiz: What's wrong with the following code?

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

int main()
{
    std::array myArray { 7, 3, 1, 9, 5 };

    // Iterate through the array and print the value of the elements
    for (int i{ 0 }; i < myArray.size(); ++i)
        std::cout << myArray[i] << ' ';

    std::cout << '\n';

    return 0;
}
```

The answer is that there's a likely signed/unsigned mismatch in this code! Due to a curious decision, the `size()` function and array index parameter to `operator[]` use a type called `size_type`, which is defined by the C++ standard as an *unsigned* integral type. Our loop counter/index (variable `i`) is a `signed int`. Therefore both the comparison `i < myArray.size()` and the array index `myArray[i]` have type mismatches.

Interestingly enough, `size_type` isn't a global type (like `int` or `std::size_t`). Rather, it's defined inside the definition of `std::array` (C++ allows nested types). This means when we want to use `size_type`, we have to prefix it with the full array type (think of `std::array` acting as a namespace in this regard). In our above example, the fully-prefixed type of "size_type" is `std::array<int, 5>::size_type`!

Therefore, the correct way to write the above code is as follows:

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

int main()
{
    std::array myArray { 7, 3, 1, 9, 5 };

    // std::array<int, 5>::size_type is the return type of size()!
    for (std::array<int, 5>::size_type i{ 0 }; i < myArray.size(); ++i)
        std::cout << myArray[i] << ' ';

    std::cout << '\n';

    return 0;
}

```

That's not very readable. Fortunately, `std::array::size_type` is just an alias for `std::size_t`, so we can use that instead.

```cpp
#include <array>
#include <cstddef> // std::size_t
#include <iostream>

int main()
{
    std::array myArray { 7, 3, 1, 9, 5 };

    for (std::size_t i{ 0 }; i < myArray.size(); ++i)
        std::cout << myArray[i] << ' ';

    std::cout << '\n';

    return 0;
}
```

A better solution is to avoid manual indexing of `std::array` in the first place. Instead, use range-based for-loops (or iterators) if possible.

Keep in mind that unsigned integers wrap around when you reach their limits. A common mistake is to decrement an index that is 0 already, causing a wrap-around to the maximum value. You saw this in the lesson about for-loops, but let's repeat.

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

int main()
{
    std::array myArray { 7, 3, 1, 9, 5 };

    // Print the array in reverse order.
    // We can use auto, because we're not initializing i with 0.
    // Bad:
    for (auto i{ myArray.size() - 1 }; i >= 0; --i)
        std::cout << myArray[i] << ' ';

    std::cout << '\n';

    return 0;
}
```

This is an infinite loop, producing undefined behavior once `i` wraps around. There are two issues here. If `myArray` is empty, ie. `size()` returns 0 (which is possible with `std::array`), `myArray.size() - 1` wraps around. The other issue occurs no matter how many elements there are. `i >= 0` is always true, because unsigned integers cannot be less than 0.

A working reverse for-loop for unsigned integers takes an odd shape:

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

int main()
{
    std::array myArray { 7, 3, 1, 9, 5 };

    // Print the array in reverse order.
    for (auto i{ myArray.size() }; i-- > 0; )
        std::cout << myArray[i] << ' ';

    std::cout << '\n';

    return 0;
}
```

Suddenly we decrement the index in the condition, and we use the postfix `--` operator. The condition runs before every iteration, including the first. In the first iteration, `i` is `myArray.size() - 1`, because `i` was decremented in the condition. When `i` is 0 and about to wrap around, the condition is no longer `true` and the loop stops. `i` actually wraps around when we do `i--` for the last time, but it's not used afterwards.

Array of struct

Of course `std::array` isn't limited to numbers as elements. Every type that can be used in a regular array can be used in a `std::array`.

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

struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};

int main()
{
    std::array<House, 3> houses{};

    houses[0] = { 13, 4, 30 };
    houses[1] = { 14, 3, 10 };
    houses[2] = { 15, 3, 40 };

    for (const auto& house : houses)
    {
        std::cout << "House number " << house.number
                  << " has " << (house.stories *
house.roomsPerStory)
                  << " rooms\n";
    }

    return 0;
}
```

Output

```
House number 13 has 120 rooms
House number 14 has 30 rooms
House number 15 has 120 rooms
```

However, things get a little weird when we try to initialize the array.

```
// Doesn't work.
std::array<House, 3>
houses{
    { 13, 4, 30 },
    { 14, 3, 10 },
    { 15, 3, 40 }
};
```

Although we can initialize `std::array` like this if its elements are simple types, like `int` or `std::string`, it doesn't work with types that need multiple values to be created. Let's have a look at why this is the case.

`std::array` is an aggregate type, just like `House`. There is no special function for the creation of a `std::array`. Rather, its internal array gets initialized like any other member variable of a `struct`. To make this easier to understand, we'll implement a simple array type ourselves.

As of now, we can't do this without having to access the `value` member. You'll learn how to get around that later. This doesn't affect the issue we're observing.

```
struct Array
{
  int
value[3]{};
};

int main()
{
    Array
array{
        11,
        12,
        13
    };

    return 0;
}
```

As expected, this works. So does `std::array` if we use it with `int` elements. When we instantiate a `struct`, we can initialize all of its members. If we try to create an `Array` of `House`s, we get an error.

```
struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};

struct Array
{
    // This is now an array of House
    House value[3]{};
};

int main()
{
    // If we try to initialize the array, we get an
error.
    Array houses{
        { 13, 4, 30 },
        { 14, 3, 10 },
        { 15, 3, 40 }
    };

    return 0;
}
```

When we use braces inside of the initialization, the compiler will try to initialize one member of the `struct` for each pair of braces. Rather than initializing the `Array` like this:

```
// This is wrong
Array houses{
    { 13, 4, 30 }, // value[0]
    { 14, 3, 10 }, // value[1]
    { 15, 3, 40 }  // value[2]
};
```

The compiler tries to initialize the `Array` like this:

```
Array houses{
    { 13, 4, 30 }, // value
    { 14, 3, 10 }, // ???
    { 15, 3, 40 }  // ???
};
```

The first pair of inner braces initializes `value`, because `value` is the first member of `Array`. Without the other two pairs of braces, there would be one house with number 13, 4 stories, and 30 rooms per story.

A reminder

Braces can be omitted during aggregate initialization:

```
struct S
{
  int arr[3]{};
  int i{};
};

// These two do the same
S s1{ { 1, 2, 3 }, 4 };
S s2{ 1, 2, 3, 4 };
```

To initialize all houses, we need to do so in the first pair of braces.

```
Array houses{
    { 13, 4, 30, 14, 3, 10, 15, 3, 40 }, // value
};
```

This works, but it's very confusing. So confusing that your compiler might even warn you about it. If we add braces around each element of the array, the initialization is a lot easier to read.

```cpp
#include <iostream>

struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};

struct Array
{
    House value[3]{};
};

int main()
{
    // With braces, this works.
    Array houses{
        { { 13, 4, 30 }, { 14, 3, 10 }, { 15, 3, 40 } }
    };

    for (const auto& house : houses.value)
    {
        std::cout << "House number " << house.number
                  << " has " << (house.stories *
house.roomsPerStory)
                  << " rooms\n";
    }

    return 0;
}
```

This is why you'll see an extra pair of braces in initializations of `std::array`.

Summary

`std::array` is a great replacement for built-in fixed arrays. It's efficient, in that it doesn't use any more memory than built-in fixed arrays. The only real downside of a `std::array` over a built-in fixed array is a slightly more awkward syntax, that you have to explicitly specify the array length (the compiler won't calculate it for you from the initializer, unless you also omit the type, which isn't always possible), and the signed/unsigned issues with size and indexing. But those are comparatively minor quibbles — we recommend using `std::array` over built-in fixed arrays for any non-trivial array use.

Next lesson

10.23 An introduction to std::vector

Back to table of contents

Previous lesson

10.21 Pointers to pointers and dynamic multidimensional arrays

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

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

Notify me about replies: POST COMMENT

DP N N FOUT

Newest