# 6.15 — An introduction to std::array

BY ALEX ON SEPTEMBER 14TH, 2015 | LAST MODIFIED BY ALEX ON FEBRUARY 15TH, 2018

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 in C++11**

Introduced in C++11, 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 set at compile time.

std::array can be initialized using an initializer lists or uniform initialization:

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

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
```

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

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

In the above example, the call to array.at(1) checks to ensure array element 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 array.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 15). 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 cleanup.

**Size and sorting**

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

```
std::array<double, 5> myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };
std::cout << "length: " << myArray.size();
```

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 <iostream>
#include <array>

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

int main()
{
    std::array<double, 5> 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 array when the array was passed to the function (for performance reasons).

*Rule: Always pass std::array by reference or const reference*

Because the length is always known, for-each (ranged for) loops work with std::array:

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

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

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

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

int main()
{
    std::array<int, 5> 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 (const auto &element : myArray)
        std::cout << element << ' ';

    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 in the lesson on iterators.

**Summary**

std::array is a great replacement for build-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, and that you have to explicitly specify the array length (the compiler won't calculate it for you from the initializer). But those are minor quibbles — we recommend using std::array over built-in fixed arrays for any non-trivial use.



**6.16 -- An introduction to std::vector**


**Index**


**6.14 -- Pointers to pointers and dynamic multidimensional arrays**


## Share this:




C++ TUTORIAL | PRINT THIS POST


## 112 comments to 6.15 — An introduction to std::array


« Older Comments 1 2


Joe
April 19, 2018 at 8:52 am · Reply

Hi guys,

I am having a bit of an issue.
I am attempting to pass a structured array to a void function to allow modification of its elements and I have run into an error I do not understand.

When I hover over the issue it displays
"
a reference of type "std::array<testStructure, 6U> &" (not const-qualified) cannot be inititalized with a value of type "std::array<testStructure, 6U>"
"

my code is

```
1  void replaceArray(std::array<testStructure, 6> &projectile, int x)
2  {
3
4      projectile.at[x].alpha =
5
6      std::cout << '\t' << '\t' << "Enter a replacement text: " << '\n' << '\n';
```