6.15 — An introduction to std::array

BY ALEX ON SEPTEMBER 14TH, 2015 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 24TH, 2020

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

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

// std::array myArray { 9.7, 7.31 };
std::array myArray<double, 2> { 9.7, 7.31 };
```

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 valid, sets array element 1 to value 6
myArray.at(9) = 10; // array element 9 is invalid, will throw an 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 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).

Rule

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.

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:

```cpp
#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. A type alias can help a bit:

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

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

    using index_t = std::array<int, 5>::size_type;
    for (index_t i{ 0 }; i < myArray.size(); ++i)
        std::cout << myArray[i] << ' ';

    std::cout << '\n';

    return 0;
}
```

This is a bit better, and this solution probably has the best combination of technical correctness and readability.

In all implementations of `std::array`, `size_type` is an alias for `std::size_t`. So it's somewhat common to see developers use `std::size_t` instead for brevity:

```cpp
#include <array>
#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.

```
#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`, its internal array gets initialized as if it were a member 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
{
```

```cpp
    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 Houses, we get an error.

```cpp
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:

```cpp
// 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:

```cpp
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 easy to read.

```
#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.

6.16 -- An introduction to std::vector

Index

6.14 -- Pointers to pointers and dynamic multidimensional arrays

C++ TUTORIAL | PRINT THIS POST

194 comments to 6.15 — An introduction to std::array

« Older Comments 1 2 3

Deepak Budha
December 23, 2019 at 1:34 am · Reply

When I run any of the given programs on Code::Blocks,I get the two errors:
1.error: missing template arguments before 'array'
2.error: 'array' was not declared in this scope

Why is it happening?

nascardriver
December 23, 2019 at 7:34 am · Reply

You didn't enable C++17 in your project settings. If your compile doesn't support C++17, you should update your compiler. If you can't/don't want to do that, explicitly specify the array type and size as shown in the beginning of this lesson.

Deepak Budha
December 24, 2019 at 4:44 am · Reply

Got it. Thanks!

Luiz Carlos
November 15, 2019 at 10:59 am · Reply

Can't I just cast myArray.size() to int instead of using an index of a different type?

nascardriver
November 15, 2019 at 11:09 am · Reply

You can. Depending on the situation, that might be more work than changing the index type.

Luiz Carlos
November 15, 2019 at 11:14 am · Reply

Thanks, I also just realized that it could be problematic for really large arrays, (which likely wouldn't be fixed anyways) since unsigned ints can have greater positive value than signed ones (half of it).

BP
September 23, 2019 at 11:14 am · Reply

Hi!
I'm currently to busy to really keep on learning Cpp right now and will continue when I have some time free, but out of intrest I decided to use Cpp to quickly do something. I need to learn all the amino acids for biochemistry and I thought it would be usefull to have some code that randomly chooses a type(one letter name, three letter name, full name, and image(through a number refering to a piece of paper with me)). and then in a random sequence prints of that type every animo acid. So if the oneLetterName was chosen, then 20 letter would have been print.

I thought as well that I might as well post the code here to see if there's room voor improvent(which always exists).
So if somebody has some free time please look this through, and yes it does involve std::array.

```
struct RandomChosenStuff
{
    int startingType{};            //one letter code, three letter code, full name, picture
    //startingType:0
    std::array<std::string, 20> oneLetterCode{ "G", "A", "P", "V","L","M","I","F","Y","W","
                                                "C","T","N","Q","K", "R", "H", "D", "E" };
    //startingType:1
    std::array<std::string, 20> threeLetterCode{ "Gly", "Ala","Pro","Val","Leu","Met","Ile"
                                                    "Ser","Cys","Thr","Asn","Gln", "Lys","A
    //startingType:2
    std::array<std::string, 20> fullName{ "Glycine","Alanine","Proline","Valine ","Leucine"
```

```
                                                   "Tryptophan","Serine ","Cysteine","Threonine","
                                                   "Histidine","Aspartate", "Glutamate" };

    //startingType:3
    std::array<int, 20> imageLink{ 1,2,3,4,5,6,7,8,9,10,12,13,14,15,16,17,18,19,20 };

};

int selectType()
{
    int randomInt{ getRandomNumber(0, 3) };
    return randomInt;
}

int main()
{
    RandomChosenStuff test;
    test.startingType = selectType();
    const int amountElements{ 20 };
    //char test{};

    std::array<int, 10> numbers{ 0,1,2,3,4,5,6,7,8,9 };

    std::array<unsigned char, 29> seed_data;
    std::random_device rd;
    std::generate_n(seed_data.data(), seed_data.size(), std::ref(rd));
    std::seed_seq seq(std::begin(seed_data), std::end(seed_data));

    std::mt19937 g(seq);

    std::shuffle(test.oneLetterCode.begin(), test.oneLetterCode.end(), g);
    std::shuffle(test.threeLetterCode.begin(), test.threeLetterCode.end(), g);
    std::shuffle(test.fullName.begin(), test.fullName.end(), g);
    std::shuffle(test.imageLink.begin(), test.imageLink.end(), g);

    switch (test.startingType)
    {
    case 0:

        for (int count{}; count < amountElements; ++count)
        {
            std::cout << "#" << count+1 << "\t " << test.oneLetterCode.at(count) << " \t";
            system("pause");
        }
        break;
    case 1:

        for (int count{}; count < amountElements; ++count)
        {
            std::cout << "#" << count + 1 << "\t " << test.threeLetterCode.at(count) << " \
            system("pause");
        }
        break;
    case 2:

        for (int count{}; count < amountElements; ++count)
        {
            std::cout << "#" << count + 1 << "\t " << test.fullName.at(count) << "\t ";
            system("pause");
        };
        break;
    case 3:
        for (int count{}; count < amountElements; ++count)
        {
            std::cout << "#" << count + 1 << "\t " << test.imageLink.at(count) << "\t ";
            system("pause");
        }
        break;
    default:
        std::cout << "Just how!?\n";
    }




    //localMain();
    //menu();
    return 0;
}
```

And in another file:

```
namespace MyRandom
{
    std::mt19937 mersenne{ static_cast<std::mt19937::result_type>(std::time(nullptr)) };
}

int getRandomNumber(int min, int max)
{
    std::uniform_int_distribution<> randomInteger{ min, max };
    return randomInteger(MyRandom::mersenne);
}
```

I recon this is spaghetti code...
Thanks eitherway!
BP

edit:
the code I used to shuffle the array, I copied from the internet. It gives a couple of warnings but works.
I don't really understand how it works, as I believe I seeded mersenne twice?

nascardriver
September 23, 2019 at 11:23 pm · Reply

Hi!

- `startingType` should be of an `enum class` type.
- You should be using a struct to store the names

```
struct SAminoAcid
{
  std::string_view strFullName;
  std::string_view strThreeLetterName;
  std::string_view strOneLetterName;

  int iImageLink;
};

std::array<SAminoAcid, 20> arrAminoAcids{
  // ...
};
```

- Use `constexpr` for compile-time constants.
- Don't use `std::system`, it work work on other platforms.
- Line 42-45: You're losing the ability to find the other properties of a given entry based on index. Eg. if you generate the three letter code "Gly", your program doesn't know the full name, one letter name, or image index for "Gly" anymore. By using a struct, everything stays together and you could print the solution.
- Line 42-45: You're shuffling everything, even the arrays you don't use anymore.
- Explicitly initialize variables which you read from before writing to them.

> I believe I seeded mersenne twice
You have two distinct mersenne twisters. You're seeding them from different sources, so they should be different. It'd be better to use a single twister tough.
You're using `std::random_device`. It might not be supported on all systems and shouldn't be used without checking its entropy (ie. check if it works).

> through a number refering to a piece of paper with me
So simple, I like it :)

p1
September 1, 2019 at 8:04 am · Reply

So what is the reason to use this over normal arrays?

nascardriver
September 1, 2019 at 8:12 am · Reply

Convenient member functions (https://en.cppreference.com/w/cpp/container/array#Member_functions).

Alex
September 1, 2019 at 2:38 pm · Reply

Beyond Nascardriver's answer, it also doesn't decay to a pointer, which can help prevent errors.

Hai Linh
August 25, 2019 at 5:42 am · Reply

Why doesn't operator[] have bounds checking? After all, one can overload operator[] to add bounds checking, so why wouldn't std::array's operator[]?

Edit: We learn exceptions in chapter 14, not chapter 15 (chapter 15 is about smart pointers)

nascardriver
August 25, 2019 at 5:51 am · Reply

Bounds checking takes up processing power. If you're careful about your indexes, you're wasting performance. If you want bounds checking, use `std::array::at`.

Samira Ferdi
August 22, 2019 at 5:05 pm · Reply

Hi, Alex and Nascardriver!

How to delete array's element?

nascardriver
August 23, 2019 at 2:05 am · Reply

```
delete array[i];
```

If you want to _remove_ an element, you'll have to wait for `std::vector`. `std::array` has a fixed size, no elements can be removed.

potterman28wxcv
August 15, 2019 at 10:33 am · Reply

Hello! Do you have any thoughts on using `auto` instead of `std::array<int, 5>::size_type` in the for loop?

nascardriver
August 16, 2019 at 12:12 am · Reply

The issue is that you need a 0 of type `std::array<int, 5>::size_type`, but you won't get that from the array.

```
// @i is an int
for (auto i{ 0 }; i < myArray.size(); ++i)
  std::cout << myArray[i] << ' ';
```

You might be tempted use `myArray.size() - 1` to initialize `i` and try to iterate in reverse.
But unsigned integers get you, this will cause an infinite loop.

The only way to get the start of the array is through `myArray.begin()`, but that gives you an iterator.
Since the whole point of a regular for-loop is that you have an index, this doesn't help.

`std::array` is guaranteed to use `std::size_t` as its index type (@Alex), so you can use that. This isn't guaranteed for all standard containers, check cppreference if you're unsure what your container's types are

std::vector with an implementation defined index
https://en.cppreference.com/w/cpp/container/vector#Member_types

std::array with a guaranteed index
https://en.cppreference.com/w/cpp/container/array#Member_types

Anastasia
August 4, 2019 at 2:49 pm · Reply

Hi!
Do we have to write a separate function for every std::array's length we eventually may want to pass to that function? I mean isn't it a bigger downside than the fact that the regular array decays to a pointer if we can just pass it's length along with it and the same function will work for (build-in) arrays of all sizes?

And a bit unrelated question, but in S.4.8 Alex wrote that `auto` keyword shouldn't work with function parameters. Yet, this snippet compiles just fine and without any warnings and works as expected. Has it been changed in C++17?

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

// std::array 1
void printStdArray(const std::array<auto, 5> &fancy_array) {
    std::cout << "std::array<auto, 5> :" << '\n';
    for (auto &value : fancy_array)
        std::cout << value << ' ';
    std::cout << '\n';
}

// std::array 2
void printStdArray(const std::array<auto, 3> &fancy_array) {
    std::cout << "std::array<auto, 3> :" << '\n';
    for (auto &value : fancy_array)
        std::cout << value << ' ';
    std::cout << '\n';
}

// std::array 3
void printStdArray(const std::array<auto, 4> &fancy_array) {
    std::cout << "std::array<auto, 4> :" << '\n';
    for (auto &value : fancy_array)
        std::cout << value << ' ';
    std::cout << '\n';
}

// build-in array 1
void printBuildInArray(const auto *just_array, const int length) {
    std::cout << "regular array :" << '\n';
    for (int count { 0 }; count < length; ++count)
        std::cout << just_array[count] << ' ';
    std::cout << '\n';
}

int main() {

    std::array<int, 5> stdArrayOf5Ints { 1, 2, 3, 4, 5 };
    std::array<double, 3> stdArrayOf3Doubles { 6.3, 7.1, 8.8 };
    std::array<char, 4> stdArrayOf4Chars { 'a', 'b', 'c', 'd' };

    int regArrayOf5Ints[] { 1, 2, 3, 4, 5 };
    double regArrayOf3Doubles[] { 6.3, 7.1, 8.8 };
    char regArrayOf4Chars[] { 'a', 'b', 'c', 'd' };

    printStdArray(stdArrayOf5Ints);
    printStdArray(stdArrayOf3Doubles);
    printStdArray(stdArrayOf4Chars);

    printBuildInArray(regArrayOf5Ints, std::size(regArrayOf5Ints));
    printBuildInArray(regArrayOf3Doubles, std::size(regArrayOf3Doubles));
    printBuildInArray(regArrayOf4Chars, std::size(regArrayOf4Chars));

    return 0;
}
```

nascardriver
August 5, 2019 at 4:21 am · Reply

Hi!

`std::array` always has to know its size. You could let the compiler generate functions for different sizes by using templates.
If you want dynamically sized lists, `std::vector` is what you're looking for (Covered later in this chapter).

`auto` cannot be used for function parameters. Make sure you followed lesson 0.10 and disabled compiler extensions.

Anastasia
August 5, 2019 at 5:50 am · Reply

> `auto` cannot be used for function parameters. Make sure you followed lesson 0.10 and disabled compiler extensions.

You're right, I've added `-pedantic-errors` flag and it doesn't compile now. I'd never have thought it's just an extension.

I'll wait untill we learn about templates then. Thank you!

noobmaster
June 21, 2019 at 4:47 am · Reply

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

int main()
{
    int arr1[]{ 3, 4, 5 };
    std::array<int, 3> arr2{ 1, 2, 3 };
    std::cout << arr2.size(); // this line works fine
    std::cout << arr1.size(); // this line won't compile
}
```

Is that because arr1 decays to pointer when passed to .size() function?

nascardriver
June 21, 2019 at 5:22 am · Reply

C-style arrays and `std::array` are substantially different. They have the same purpose, but aren't drop-in replacements for one another.
You'll understand their differences after chapter 8, which talks about classes.

noobmaster
June 22, 2019 at 12:00 am · Reply

okay then. Thanks

Tamara
June 6, 2019 at 3:05 am · Reply

I'm a bit confused by this part: "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."
But if that's the case then haven't we've been having type mismatches in our code every time we would use for loops with built-in fixed arrays, for example like this:

```
1  int main()
2  {
3      int myArray[5] = { 7,3,1,9,5 };
4      for (int i{ 0 }; i < size(myArray); ++i)
5      {
6          std::cout << myArray[i] << "\n";
7      }
8      return 0;
9  }
```

Now, by using typeid(size(myArray)).name(), I can see that the return type of size() is, indeed, an unsigned int and I do get a warning for it. But when I remove that, I get no type mismatch warning for the array index myArray[i]. Why is that? And also, how could I see the type of the array index parameter using typeid? If I just used typeid(myArray[i]).name() that would return the type of the array element, int in this case.

Thanks in advance.

nascardriver
June 6, 2019 at 4:33 am · Reply

Line 4 should cause a warning, because you're comparing signed to unsigned. If it doesn't make sure you enabled sign comparison warnings in your compiler settings.
Line 6 doesn't cause a warning, because array indexes only have to be integral types. This is not the case for `std::array`. `std::array` wants its `size_type` type as an index.

Tamara
June 6, 2019 at 5:01 am · Reply

Oh, so it differs with std::array in that regard. But then one more thing. When I copied this entire piece of code from the lesson into VS:

```
1  #include <iostream>
2  #include <array>
3
4  int main()
5  {
6      std::array<int, 5> myArray{ 7, 3, 1, 9, 5 };
7
8      // Iterate through the array and print the value of the elements
9      for (int i{ 0 }; i < myArray.size(); ++i)
10         std::cout << myArray[i] << ' ';
11
12     return 0;
13 }
```

it gives me a warning for Line 9, for the "myArray.size()" as it should. But in the lesson it also says that the array index "myArray[i]" is a type mismatch and I'm not getting a warning back for that line. I'm assuming the sign comparison warnings are enabled since I'm getting a warning for Line 9.

nascardriver
June 6, 2019 at 5:09 am · Reply

Line 9 is a sign comparison,
Line 10 is a sign conversion.
Sorry, I don't know how to control warnings in VS, you'll have to look it up on your own.

Tamara
June 6, 2019 at 5:17 am · Reply

Ah, I get it now. Thanks for the help!

Yaroslav
June 28, 2019 at 11:58 am · Reply

can i ask why would we not just use

```
unsigned int i
```

or even better cast .size() to signed so 'i' will remain signed and we could use it down in comparison with signed by code

```
for (int i=0; i<static_cast<signed int>(arr.size()); ++i)
```

? thank you

nascardriver
June 28, 2019 at 12:14 pm · Reply

> why would we not just use `unsigned int i`
For one, we don't know that `arr.size()` returns an unsigned int. We could use `std::size_t`, but that's and unsigned integer type and looping over unsigned integers (Or anything involving unsigned integers, really) is prone to wrap arounds or unexpected signed -> unsigned promotions. It's easier to stay with signed integers as long as you don't need the high positive range of unsigned.
A cast is the preferred solution. Since C++20, you can replace `arr.size()` with `std::ssize(arr)`, which returns the size as a signed integer, making the cast unnecessary.

Yaroslav
June 28, 2019 at 12:43 pm · Reply

thank you @nascardriver.

darshan
March 6, 2019 at 12:28 am · Reply

Why does

```
typedef std::vector::size_type index_t;
```

give an error?

nascardriver

March 6, 2019 at 6:49 am · Reply

@std::vector is a template type, you need to specify it's contents.

```
// eg. if your vector stores int
using index_t = std::vector<int>::size_type;
```

Piyush Yadav
February 17, 2019 at 10:56 pm · Reply

```
1  #include<array>
2  std::array<std::string,9> keywords {"int","double","char","main","class","break","const","an
3  #include<iostream>
4  int main(){
5      std::string var;
6      std::cout<<"Enter the variable name :";
7      std::getline(std::cin,var);
8      return 0;
9  }
```

Why i am getting error in the above program?
But in this one.

```
1  #include<array>
2  std::array<int,9> keywords {1,2,3,4,5,6,7,8,9};
3  #include<iostream>
4  int main(){
5      std::string var;
6      std::cout<<"Enter the variable name :";
7      std::getline(std::cin,var);
8      return 0;
9  }
```

nascardriver
February 18, 2019 at 7:20 am · Reply

Snippet 1
* Line 5: Initialize your variables with brace initializers.
* Line 2: Limit your lines to 80 characters in length for better readability on small displays.

Snippet 2
* Line 5: Initialize your variables with brace initializers.

If you're getting errors, post the error message.
You're using @std::string without including <string>.

Piyush
February 18, 2019 at 8:17 am · Reply

Oh! I actually never included string library unless i had to do string operations as it worked all the time and this is the first time i used std::string in global scope. It worked when i included string library. But it also works when i initialise keywords array inside the main even without using string library.

```
#include<array>//does'nt works if not
```

```
included.
#include<string>
std::array<std::string,9> keywords {"int","double","char","main","class","break",
#include<iostream>
int main(){
    std::string var{};
    std::cout<<"Enter the variable name :";
    std::getline(std::cin,var);
    return 0;
}
```

```
#include<array>
#include<string>//works even when not included.
#include<iostream>
int main(){
    std::array<std::string,9> keywords {"int","double","char","main","class","bre
    std::string var{};
    std::cout<<"Enter the variable name :";
    std::getline(std::cin,var);
    return 0;
}
```

Why this happens ?
Thanks a lot for the suggestions. :)

nascardriver
February 18, 2019 at 8:19 am · Reply

Because you included <iostream>, which in your case, includes <string>. This is non-standard behavior. If you use @std::string, include <string>.

Piyush
February 18, 2019 at 8:23 am · Reply

Now i understood. Thanks a lot.

Shri
February 8, 2019 at 9:38 am · Reply

Hi Alex, in this chapter you have mentioned that one of the reasons to use std::array is that, when passing std:array to a function (where the function parameter is a reference to the argument), it does not decay into a pointer.

But in later chapters you show that even built-in arrays can be passed by reference (along with mentioning the length of array) such that the argument does not decay into a pointer.

```
void change_array (int (&array2) [5])
{
    for (auto &element : array2)
    {
            element = element + 1;
    }
}

Void main()
{
        int array1[5]  {0,1,2,3,4};
    change_array(array1);
}
```

Based on this, do you think this advantage (std:array not decaying into a pointer when passed by reference into a function) should be removed from this chapter?

Alex
February 9, 2019 at 8:56 am · Reply

No, it's still an advantage even if other methods share that advantage.

« Older Comments 1 2 3