6.16 — An introduction to std::vector

BY ALEX ON SEPTEMBER 28TH, 2015 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In the previous lesson, we introduced `std::array`, which provides the functionality of C++'s built-in fixed arrays in a safer and more usable form.

Analogously, the C++ standard library provides functionality that makes working with dynamic arrays safer and easier. This functionality is named `std::vector`.

Unlike `std::array`, which closely follows the basic functionality of fixed arrays, `std::vector` comes with some additional tricks up its sleeves. These help make `std::vector` one of the most useful and versatile tools to have in your C++ toolkit.

An introduction to std::vector

Introduced in C++03, `std::vector` provides dynamic array functionality that handles its own memory management. This means you can create arrays that have their length set at run-time, without having to explicitly allocate and deallocate memory using `new` and `delete`. `std::vector` lives in the <vector> header.

Declaring a `std::vector` is simple:

```
#include <vector>

// no need to specify length at initialization
std::vector<int> array;
std::vector<int> array2 = { 9, 7, 5, 3, 1 }; // use initializer list to initialize array
std::vector<int> array3 { 9, 7, 5, 3, 1 }; // use uniform initialization to initialize array
 (C++11 onward)

// as with std::array, the type can be omitted since C++17
std::vector array4 { 9, 7, 5, 3, 1 }; // deduced to std::vector<int>

```

Note that in both the uninitialized and initialized case, you do not need to include the array length at compile time. This is because `std::vector` will dynamically allocate memory for its contents as requested.

Just like `std::array`, accessing array elements can be done via the `[]` operator (which does no bounds checking) or the `at()` function (which does bounds checking):

```
array[6] = 2; // no bounds checking
array.at(7) = 3; // does bounds checking
```

In either case, if you request an element that is off the end of the array, the vector will *not* automatically resize.

As of C++11, you can also assign values to a `std::vector` using an initializer-list:

```
array = { 0, 1, 2, 3, 4 }; // okay, array length is now 5
array = { 9, 8, 7 }; // okay, array length is now 3
```

In this case, the vector will self-resize to match the number of elements provided.

Self-cleanup prevents memory leaks

When a vector variable goes out of scope, it automatically deallocates the memory it controls (if necessary). This is not only handy (as you don't have to do it yourself), it also helps prevent memory leaks. Consider the following snippet:

```
void doSomething(bool earlyExit)
{
    int *array{ new int[5] { 9, 7, 5, 3, 1 } };

    if (earlyExit)
        return;

    // do stuff here

    delete[] array; // never called
}
```

If earlyExit is set to true, array will never be deallocated, and the memory will be leaked.

However, if `array` is a `std::vector`, this won't happen, because the memory will be deallocated as soon as `array` goes out of scope (regardless of whether the function exits early or not). This makes `std::vector` much safer to use than doing your own memory allocation.

Vectors remember their length

Unlike built-in dynamic arrays, which don't know the length of the array they are pointing to, std::vector keeps track of its length. We can ask for the vector's length via the `size()` function:

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

void printLength(const std::vector<int>& array)
{
    std::cout << "The length is: " << array.size() << '\n';
}

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

    return 0;
}
```

The above example prints:

```
The length is: 5
```

Just like with `std::array`, `size()` returns a value of nested type `size_type` (full type in the above example would be `std::vector<int>::size_type`), which is an unsigned integer.

Resizing a vector

Resizing a built-in dynamically allocated array is complicated. Resizing a `std::vector` is as simple as calling the `resize()` function:

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

int main()
{
    std::vector array { 0, 1, 2 };
    array.resize(5); // set size to 5

    std::cout << "The length is: " << array.size() << '\n';
```

```
10
11      for (int i : array)
12          std::cout << i << ' ';
13
14      std::cout << '\n';
15
16      return 0;
17  }
```

This prints:

```
The length is: 5
0 1 2 0 0
```

There are two things to note here. First, when we resized the vector, the existing element values were preserved! Second, new elements are initialized to the default value for the type (which is 0 for integers).

Vectors may be resized to be smaller:

```
1   #include <vector>
2   #include <iostream>
3
4   int main()
5   {
6       std::vector array { 0, 1, 2, 3, 4 };
7       array.resize(3); // set length to 3
8
9       std::cout << "The length is: " << array.size() << '\n';
10
11      for (int i : array)
12          std::cout << i << ' ';
13
14      std::cout << '\n';
15
16      return 0;
17  }
```

This prints:

```
The length is: 3
0 1 2
```

Resizing a vector is computationally expensive, so you should strive to minimize the number of times you do so. If you need a vector with a specific number of elements but don't know the values of the elements at the point of declaration, you can create a vector with default elements like so:

```
1   #include <iostream>
2   #include <vector>
3
4   int main()
5   {
6       // Using direct initialization, we can create a vector with 5 elements,
        // each element is a 0. If we use brace initialization, the vector would
7       // have 1 element, a 5.
        std::vector<int> array(5);
8
9       std::cout << "The length is: " << array.size() << '\n';
10
11      for (int i : array)
12          std::cout << i << ' ';
```

```
13
14      std::cout << '\n';
15
16      return 0;
17  }
18
19
```

This prints:

```
5
0 0 0 0 0
```

We'll talk about why direct and brace-initialization are treated differently in a later chapter. A rule of thumb is, if the type is some kind of list and you don't want to initialize it with a list, use direct initialization.

Compacting bools

`std::vector` has another cool trick up its sleeves. There is a special implementation for `std::vector` of type bool that will compact 8 booleans into a byte! This happens behind the scenes, and doesn't change how you use the std::vector.

```
1  #include <vector>
2  #include <iostream>
3
4  int main()
5  {
6      std::vector<bool> array { true, false, false, true, true };
7      std::cout << "The length is: " << array.size() << '\n';
8
9      for (int i : array)
10         std::cout << i << ' ';
11
12     std::cout << '\n';
13
14     return 0;
15 }
```

This prints:

```
The length is: 5
1 0 0 1 1
```

More to come

Note that this is an introduction article intended to introduce the basics of `std::vector`. In lesson 7.10, we'll cover some additional capabilities of `std::vector`, including the difference between a vector's length and capacity, and take a deeper look into how `std::vector` handles memory allocation.

Conclusion

Because variables of type `std::vector` handle their own memory management (which helps prevent memory leaks), remember their length, and can be easily resized, we recommend using `std::vector` in most cases where dynamic arrays are needed.


6.17 -- Introduction to iterators


Index


6.15 -- An introduction to std::array

C++ TUTORIAL | PRINT THIS POST

193 comments to 6.16 — An introduction to std::vector

« Older Comments 1 2 3

Sid22
January 28, 2020 at 10:13 am · Reply

Recently learned about brace brace initialization and vectors so playing around with both. This code is not printing the float values of the

```
athlete
```

objects. Can't understand why?

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

class athlete
{
public:
    std::string f_name;
    int age;
    float speed;
};

//overloaded output stream operator
 std::ostream& operator<< ( std::ostream& print_that, const athlete& ind)
{
    print_that << "\nFirst name: " << ind.f_name << "\nAge: " << ind.age << "\nSpeed: " << ind.age << "\n";
    return print_that;
}

int main()
{
    std::vector<athlete> aray;
    aray.push_back({"David",22,50.1});   // may be I am doing something wrong here?
    aray.push_back({"Goliath",16,100.23}); // or here?

    //printing out
    for (athlete player: aray)
```

```
    {
        std::cout << player;
    }


}


```

nascardriver
January 29, 2020 at 4:21 am · Reply

You're not printing the speed, you're printing age twice.

Sid22
January 30, 2020 at 9:25 pm · Reply

Oh my God. How embarrassing. Thank you for answering my stupid questions. I can't even imagine the patience it takes.

Sriram M R
December 18, 2019 at 6:03 pm · Reply

Under Resizing a vector there is a error in the code in line 7 of

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

int main()
{
    // Using direct initialization, we can create a vector with 5 elements,
    // each element is a 0. If we use brace initialization, the vector would
    // have 1 element, a 5.
    std::vector<int> vec(5);

    std::cout << "The length is: " << array.size() << '\n';

    for (int i : vec)
        std::cout << i << ' ';

    std::cout << '\n';

    return 0;
}


```

Instead of array.size() shouldn't we use vec.size()

nascardriver
December 19, 2019 at 2:24 am · Reply

Renamed `vec` to "array" to match the rest of the lesson, thanks!

Ryan

December 6, 2019 at 12:43 pm · Reply

In very large programs, does bound checking increase compile time significantly or is it just a minor issue?

nascardriver
December 7, 2019 at 4:43 am · Reply

Compile time shouldn't bother you, the user of your software isn't affected by it.
Bounds checking via `at` doesn't increase compile time, it's performed at run-time, which can affect the user.
An out-of-bounds error should be caught during testing. Using `at` slows down the user without adding a safety benefit (Because we assume there are no out-of-bounds errors in user software).
You can use `at` while you learn programming, because you're likely to make mistakes, but I can't recommend using it in production code.

Ged
December 2, 2019 at 7:23 am · Reply

5. About std::array. We talked about size_type when we want to get the length of the array. We used r-value. But why don't we just use " const (datatype) length = 5(example) ". I wrote the "const" because it doesn't allow me to use length in the std::array without "const". In a situation if we write a lot of code and use a lot of std::array. Manually changing the length number would be very annoying. As well as we wouldn't need to convert size_type. I see a lot of pros in this situation or am I missing something?

5. I don't understand your suggestion, can you post a full example?

I'm writing a new post, because for some reason it only allows me to reply through a phone.

```
1 std::array<int,15> scores; // let's say we will use this 100 times
2
3 const int length{15};
4 std::array<int,length> scores;
```

In example number 1 if we use that line a lot in our code and it happens that we need to change the array from 15 to 25 it can take a long time replace 15 in every line. Isn't the second option better?

Extra question. This came up to me because a lot of the tasks that I'm doing online ask the user to input the length of an array.
Let's say I have 3 different files with the length of an array and its elements. How do I use std::vector to input all that? Cause usually I always used simple fixed arrays, then after finding out about pointers, I've started using them. But from the lesson with the vector, there weren't any examples of situations where the user needs to input the length.

nascardriver
December 2, 2019 at 7:43 am · Reply

You can use a type alias that you then use instead of the array type

```
1 using my_array_type = std::array<int, 15>;
2
3 my_array_type scores;
```

Now you only have to update the alias when you want to change the length.

If you know how many elements your vector will have beforehand, you can use either one of those:

```
1 std::vector<int> v(100); // 100 elements, all 0
```

```
2
3  // If you don't know the size at the definition:
4  std::vector<int> v{};
5  v.resize(100); // 100 elements, all 0
```

Ged
November 28, 2019 at 8:43 am · Reply

A few questions to make things a bit more clear for myself.
1. The only 2 things we should be using when working with arrays are std::vector and std::array, should we forget about pointer arrays and simple fixed arrays? (just know that they exist)?
2. std::vector is used when we need a dynamical array (we don't know what length it will be).
3. std::array is used when we need a fixed array (we know its length).
4. But what if we need to have a 2d or 3d array? If we write it int array[x][y] or int array[x][y][z]. It is the easiest to understand, but I feel it is bad for practice. Should we always go with the flatten (x*y) method as you specified in "6.14 topic" to avoid confusion?

5. About std::array. We talked about size_type when we want to get the length of the array. We used r-value. But why don't we just use " const (datatype) length = 5(example) ". I wrote the "const" because it doesn't allow me to use length in the std::array without "const". In a situation if we write a lot of code and use a lot of std::array. Manually changing the length number would be very annoying. As well as we wouldn't need to convert size_type. I see a lot of pros in this situation or am I missing something?

6. Last thing about the last code in this topic.

There is a special implementation for std::vector of type bool that will compact 8 booleans into a byte!

```
1  std::vector<bool> array { true, false, false, true, true };
```

How to understand this sentence "8 booleans into a byte". Can the array have a maximum of 255 length or I misunderstood something?

I do apologise if I asked too many questions, but I don't wanna leave things out that I don't understand. Thank you in advance.

nascardriver
November 29, 2019 at 2:22 am · Reply

1. This depends on your field of work. Low-level code will probably want to use a custom container for higher efficiency or better memory usage. Unless you have a reason to implement a container yourself, you should stick to those in the container library, which `vector` and `array` are a part of (https://en.cppreference.com/w/cpp/container).

2. Correct.

3. Correct.

4. One-dimensional arrays are easier to work with most of the time. If you really want a multidimensional array, you can do that with `std::array` too. Either by using a long type definition or by writing a custom type that creates the array for you.

5. I don't understand your suggestion, can you post a full example?

6. Without the specialization, each element of the vector will take up `sizeof(bool)` bytes. That's a lot of wasted space, because a bool is either false or true, so a bool fits into a single bit. 1 byte is 8 bits, so we can store 8 bools in 1 byte, which means we can store 8 times the amount of elements without using any more memory.

« Older Comments 1 2 3