6.16 — An introduction to std::vector

BY ALEX ON SEPTEMBER 28TH, 2015 | LAST MODIFIED BY ALEX ON NOVEMBER 16TH, 2017

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 runtime, 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 onwa
rd)
```

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 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 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 <vector>
#include <iostream>

int main()
{
    std::vector<int> array { 9, 7, 5, 3, 1 };
    std::cout << "The length is: " << array.size() << '\n';

    return 0;
}
```

The above example prints:

```
The length is: 5
```

Resizing an array

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

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

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

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

    for (auto const &element: array)
        std::cout << element << ' ';

    return 0;
}
```

This prints:

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

There are two things to note here. First, when we resized the array, 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:

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

int main()
{
    std::vector<int> array { 0, 1, 2, 3, 4 };
    array.resize(3); // set length to 3

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

```
10
11      for (auto const &element: array)
12          std::cout << element << ' ';
13
14      return 0;
15  }
```

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.

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 is largely transparent to you as a programmer.

```
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 (auto const &element: array)
10          std::cout << element << ' ';
11
12      return 0;
13  }
```

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 a 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.x -- Chapter 6 comprehensive quiz


Index


6.15 -- An introduction to std::array