16.10 — std::vector resizing and capacity

In the previous lessons in this chapter, we introduced containers, arrays, and `std::vector`. We also discussed topics such as how to access array elements, get the length of an array, and how to traverse an array. While we used `std::vector` in our examples, the concepts that we have discussed are generally applicable to all of the array types.

In the remaining lessons in this chapter, we're going to focus on the one thing that makes `std::vector` significantly different than most of the other array types: the ability to resize itself after it has been instantiated.

Fixed-size arrays vs dynamic arrays

Most array types have a significant limitation: the length of the array must be known at the point of instantiation, and then cannot be changed. Such arrays are called **fixed-size arrays** or **fixed-length arrays**. Both `std::array` and `C-style arrays` are fixed-size array types. We'll discuss these further next chapter.

On the other hand, `std::vector` is a dynamic array. A **dynamic array** (also called a **resizable array**) is an array whose size can be changed after instantiation. This ability to be resized is what makes `std::vector` special.

Resizing a `std::vector` at runtime

A `std::vector` can be resized after instantiation by calling the `resize()` member function with the new desired length:

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

int main()
{
    std::vector v{ 0, 1, 2 }; // create vector with 3 elements
    std::cout << "The length is: " << v.size() << '\n';

    v.resize(5);              // resize to 5 elements
    std::cout << "The length is: " << v.size() << '\n';

    for (auto i : v)
        std::cout << i << ' ';

    std::cout << '\n';

    return 0;
}
```

This prints:

```
The length is: 3
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, the new elements are value-initialized (which performs default-initialization for class types, and zero-initialization for other types). Thus the two new elements of type `int` were zero-initialized to value `0`.

Vectors may also be resized to be smaller:

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

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

int main()
{
    std::vector v{ 0, 1, 2, 3, 4 }; // length is initially 5
    printLength(v);

    v.resize(3);                    // resize to 3 elements
    printLength(v);

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

    std::cout << '\n';

    return 0;
}
```

This prints:

```
The length is: 5
The length is: 3
0 1 2
```

The length vs capacity of a `std::vector`

Consider a row of 12 houses. We'd say that the count of houses (or the length of the row of houses) is 12. If we wanted to know which of those houses were currently being occupied… we'd have to determine that in some other way (e.g. ring the doorbell and see if anybody answered). When we only have a length, we only know how many things exist.

Now consider a carton of eggs that currently has 5 eggs in it. We'd say that the count of eggs is 5. But in this context, there's another dimension we care about: how many eggs the carton could hold if it were full. We'd say that the capacity of the carton of eggs is 12. The carton has room for 12 eggs, and only 5 are being used -- therefore, we could add 7 more eggs without overflowing the carton. When we have both a length and a capacity, we can differentiate how many things currently exist from how many things there is space for.

Up to this point, we've only talked about the length of a `std::vector`. But `std::vector` also has a capacity. In the context of a `std::vector`, **capacity** is how many elements the `std::vector` has allocated storage for, and **length** is how many elements are currently being used.

A `std::vector` with a capacity of 5 has allocated space for 5 elements. If the vector contains 2 elements in active use, the length (size) of the vector is 2. The 3 remaining elements have memory allocated for them, but they are not considered to be in active use. They can be used later without overflowing the vector.

Key insight

The length of a vector is how many elements are “in use”.
The capacity of a vector is how many elements have been allocated in memory.

Getting the capacity of a `std::vector`

We can ask a `std::vector` for its capacity via the `capacity()` member function.

For example:

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

void printCapLen(const std::vector<int>& v)
{
        std::cout << "Capacity: " << v.capacity() << " Length:" << v.size() << '\n';
}

int main()
{
    std::vector v{ 0, 1, 2 }; // length is initially 3

    printCapLen(v);

    for (auto i : v)
        std::cout << i << ' ';
    std::cout << '\n';

    v.resize(5); // resize to 5 elements

    printCapLen(v);

    for (auto i : v)
        std::cout << i << ' ';
    std::cout << '\n';

    return 0;
}
```

On the author’s machine, this prints the following:

```
Capacity: 3  Length: 3
0 1 2
Capacity: 5  Length: 5
0 1 2 0 0
```

First, we've initialized the vector with 3 elements. This causes the vector to allocate storage for 3 elements (capacity is 3), and all 3 elements are considered to be in active use (length = 3).

We then call `resize(5)`, meaning we now want a vector with a length of 5. Since the vector only has storage for 3 elements, but it needs 5, the vector needs to get more storage to hold the additional elements.

After the call to `resize()` has completed, we can see that the vector now has space for 5 elements (capacity is 5), and that all 5 elements are now considered to be in use (length is 5).

Most of the time you won't need to use the `capacity()` function, but we'll use it a lot in the following examples so we can see what's happening to the storage of the vector.

Reallocation of storage, and why it is expensive

When a `std::vector` changes the amount of storage it is managing, this process is called **reallocation**. Informally, the reallocation process goes something like this:

- The `std::vector` acquires new memory with capacity for the desired number of elements. These elements are value-initialized.
- The elements in the old memory are copied (or moved, if possible) into the new memory. The old memory is then returned to the system.
- The capacity and length of the `std::vector` are set to the new values.

From the outside, it looks like the `std::vector` has been resized. But internally, the memory (and all of the elements) have actually been replaced!

Related content

The process of acquiring new memory at runtime is called dynamic memory allocation. We cover this in lesson 19.1 -- Dynamic memory allocation with new and delete.

Because reallocation typically requires every element in the array to be copied, reallocation is an expensive process. As a result, we want to avoid reallocation as much as reasonable.

Key insight

Reallocation is typically expensive. Avoid unnecessary reallocations.

Why differentiate length and capacity?

A `std::vector` will reallocate its storage if needed, but like Melville's Bartleby, it would prefer not to, because reallocating storage is computationally expensive.

If a `std::vector` only kept track of its length, then every `resize()` request would result in an expensive reallocation to the new length. Separating length and capacity gives the `std::vector` the ability to be smarter about when reallocations are needed.

The following example illustrates this:

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

void printCapLen(const std::vector<int>& v)
{
        std::cout << "Capacity: " << v.capacity() << " Length:" << v.size() << '\n';
}

int main()
{
    // Create a vector with length 5
    std::vector v{ 0, 1, 2, 3, 4 };
    v = { 0, 1, 2, 3, 4 }; // okay, array length = 5
    printCapLen(v);

    for (auto i : v)
        std::cout << i << ' ';
    std::cout << '\n';

    // Resize vector to 3 elements
    v.resize(3); // we could also assign a list of 3 elements here
    printCapLen(v);

    for (auto i : v)
        std::cout << i << ' ';
    std::cout << '\n';

    // Resize vector back to 5 elements
    v.resize(5);
    printCapLen(v);

    for (auto i : v)
        std::cout << i << ' ';
    std::cout << '\n';

    return 0;
}
```

This produces the following:

```
Capacity: 5  Length: 5
0 1 2 3 4
Capacity: 5  Length: 3
0 1 2
Capacity: 5  Length: 5
0 1 2 0 0
```

When we initialized our vector with 5 elements, the capacity was set to 5, indicating that our vector initially allocated space for 5 elements. The length was also set to 5, indicating that all of those elements are in use.

After we called `v.resize(3)`, the length was changed to 3 to fulfill our request for a smaller array. However, note that the capacity is still 5, meaning that the vector did not do a reallocation!

Finally, we called `v.resize(5)`. Because the vector already had a capacity of 5, it did not need to reallocate. It simply changed the length back to 5, and value-initialized the last two elements.

By separating length and capacity, in this example we avoided 2 reallocations that would have otherwise occurred. In the next lesson, we'll see examples where we are adding elements to a vector one by one. In such cases, the ability to not reallocate every time length changes is even more important.

Key insight

Tracking capacity separately from length allows the `std::vector` to avoid some reallocations when length is changed.

Vector indexing is based on length, not capacity

You may be surprised to find that the valid indices for the subscript operator (`operator[]`) and `at()` member function is based on the vector's length, not the capacity.

In the example above, when `v` has capacity 5 and length 3, only indices from 0 and 2 are valid. Even though the elements with indices between the length of 3 (inclusive) and capacity of 5 (exclusive) exist, their indices are considered to be out of bounds.

Warning

A subscript is only valid if it is between 0 and the vector's length (not its capacity)!

Shrinking a `std::vector`

Resizing a vector to be larger will increase the vector's length, and will increase its capacity if required. However, resizing a vector to be smaller will only decrease its length, and not its capacity.

Reallocating a vector just to reclaim the memory from a small number of elements that are no longer needed is a poor choice. However, in the case where we have a vector with a large number of elements that we no longer need, the memory waste could be substantive.

To help address this situation, `std::vector` has a member function called `shrink_to_fit()` that requests that the vector shrink its capacity to match its length. This request is non-binding, meaning the implementation is free to ignore it. Depending on what the implementation thinks is best, an implementation may decide to fulfill the request, may partially fulfill it (e.g. reduce the capacity but not all the way), or may completely ignore the request.

Here's an example:

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

void printCapLen(const std::vector<int>& v)
{
	std::cout << "Capacity: " << v.capacity() << " Length:" << v.size() << '\n';
}

int main()
{

	std::vector<int> v(1000); // allocate room for 1000 elements
	printCapLen(v);

	v.resize(0); // resize to 0 elements
	printCapLen(v);

	v.shrink_to_fit();
	printCapLen(v);

	return 0;
}
```

On the author's machine, this produces the following result:

```
Capacity: 1000  Length: 1000
Capacity: 1000  Length: 0
Capacity: 0  Length: 0
```

As you can see, when `v.shrink_to_fit()` was called, the vector reallocated its capacity to 0, freeing up the memory for 1000 elements.

Quiz time

Question #1

What do the length and capacity of a std::vector represent?

Show Solution

Why are length and capacity separate values?

Show Solution

Are the valid indices for `std::vector` based on length or capacity?

Show Solution