11.9 — std::vector capacity and stack behavior

ALEX AUGUST 28, 2021

In lesson 10.23 -- An introduction to std::vector, we introduced std::vector and talked about how std::vector can be used as a dynamic array that both remembers its length and can be dynamically resized as required.

Although this is the most useful and commonly used part of std::vector, std::vector has some additional attributes and capabilities that make it useful in some other capacities as well.

Length vs capacity

Consider the following example:

```
int* array{ new int[10] { 1, 2, 3, 4, 5 }
};
```

We would say that this array has a length of 10, even though we're only using 5 of the elements that we allocated.

However, what if we only wanted to iterate over the elements we've initialized, reserving the unused ones for future expansion? In that case, we'd need to separately track how many elements were "used" from how many elements were allocated. Unlike a built-in array or a std::array, which only remembers its length, std::vector contains two separate attributes: length and capacity. In the context of a std::vector, length is how many elements are being used in the array, whereascapacity is how many elements were allocated in memory.

Taking a look at an example from the previous lesson on std::vector:

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

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

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

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

    return 0;
};
```

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

In the above example, we've used the resize() function to set the vector's length to 5. This tells variable array that we're intending to use the first 5 elements of the array, so it should consider those in active use. However, that leaves an interesting question: what is the capacity of this array?

We can ask the std::vector what its capacity is via the capacity() function:

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

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

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

On the authors machine, this printed:

```
The length is: 5
The capacity is: 5
```

In this case, the resize() function caused the std::vector to change both its length and capacity. Note that the capacity is guaranteed to be at least as large as the array length (but could be larger), otherwise accessing the elements at the end of the array would be outside of the allocated memory!

More length vs. capacity

Why differentiate between length and capacity? std::vector will reallocate its memory if needed, but like Melville's Bartleby, it would prefer not to, because resizing an array is computationally expensive. Consider the following:

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

int main()
{
  std::vector<int> array{};
  array = { 0, 1, 2, 3, 4 }; // okay, array length = 5
  std::cout << "length: " << array.size() << "  capacity: " << array.capacity() << '\n';

  array = { 9, 8, 7 }; // okay, array length is now 3!
  std::cout << "length: " << array.size() << "  capacity: " << array.capacity() << '\n';

  return 0;
}
```

This produces the following:

```
length: 5  capacity: 5
length: 3  capacity: 5
```

Note that although we assigned a smaller array to our vector, it did not reallocate its memory (the capacity is still 5). It simply changed its length, so it knows that only the first 3 elements are valid at this time.

Array subscripts and at() are based on length, not capacity

The range for the subscript operator ([]) and at() function is based on the vector's length, not the capacity. Consider the array in the previous example, which has length 3 and capacity 5. What happens if we try to access the array element with index 4? The answer is that it fails, since 4 is greater than the length of the array.

Note that a vector will not resize itself based on a call to the subscript operator or at() function!

Stack behavior with std::vector

If the subscript operator and at() function are based on the array length, and the capacity is always at least as large as the array length, why even worry about the capacity at all? Although std::vector can be used as a dynamic array, it can also be used as a stack. To do this, we can use 3 functions that match our key stack operations:

- **push_back() pushes an element on the stack.**
- **back() returns the value of the top element on the stack.**
- **pop_back() pops an element off the stack.**

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

void printStack(const std::vector<int>& stack)
{
 for (auto element : stack)
  std::cout << element << ' ';
 std::cout << "(cap " << stack.capacity() << " length " << stack.size() <<
")\n";
}

int main()
{
 std::vector<int> stack{};

 printStack(stack);

 stack.push_back(5); // push_back() pushes an element on the stack
 printStack(stack);

 stack.push_back(3);
 printStack(stack);

 stack.push_back(2);
 printStack(stack);

 std::cout << "top: " << stack.back() << '\n'; // back() returns the last
element

 stack.pop_back(); // pop_back() pops an element off the stack
 printStack(stack);

 stack.pop_back();
 printStack(stack);

 stack.pop_back();
 printStack(stack);

 return 0;
}
```

This prints:

```
(cap 0 length 0)
5 (cap 1 length 1)
5 3 (cap 2 length 2)
5 3 2 (cap 3 length 3)
top: 2
5 3 (cap 3 length 2)
5 (cap 3 length 1)
(cap 3 length 0)
```

Unlike array subscripts or at(), the stack-based functions*will* resize the std::vector if necessary. In the example above, the vector gets resized 3 times (from a capacity of 0 to 1, 1 to 2, and 2 to 3).

Because resizing the vector is expensive, we can tell the vector to allocate a certain amount of capacity up front using the reserve() function:

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

void printStack(const std::vector<int>& stack)
{
 for (auto element : stack)
  std::cout << element << ' ';
 std::cout << "(cap " << stack.capacity() << " length " << stack.size() <<
")\n";
}

int main()
{
 std::vector<int> stack{};

 stack.reserve(5); // Set the capacity to (at least) 5

 printStack(stack);

 stack.push_back(5);
 printStack(stack);

 stack.push_back(3);
 printStack(stack);

 stack.push_back(2);
 printStack(stack);

 std::cout << "top: " << stack.back() << '\n';

 stack.pop_back();
 printStack(stack);

 stack.pop_back();
 printStack(stack);

 stack.pop_back();
 printStack(stack);

 return 0;
}
```

This program prints:

```
(cap 5 length 0)
5 (cap 5 length 1)
5 3 (cap 5 length 2)
5 3 2 (cap 5 length 3)
top: 2
5 3 (cap 5 length 2)
5 (cap 5 length 1)
(cap 5 length 0)
```

We can see that the capacity was preset to 5 and didn't change over the lifetime of the program.

Vectors may allocate extra capacity

When a vector is resized, the vector may allocate more capacity than is needed. This is done to provide some "breathing room" for additional elements, to minimize the number of resize operations needed. Let's take a look at this:

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

int main()
{
 std::vector<int> v{ 0, 1, 2, 3, 4 };
 std::cout << "size: " << v.size() << "  cap: " << v.capacity() <<
'\n';

 v.push_back(5); // add another element
 std::cout << "size: " << v.size() << "  cap: " << v.capacity() <<
'\n';

 return 0;
}
```

On the author's machine, this prints:

```
size: 5  cap: 5
size: 6  cap: 7
```

When we used push_back() to add a new element, our vector only needed room for 6 elements, but allocated room for 7. This was done so that if we were to push_back() another element, it wouldn't need to resize immediately.

If, when, and how much additional capacity is allocated is left up to the compiler implementer.

Next lesson
11.10 Recursion

Back to table of contents

Previous lesson
11.8 The stack and the heap