School of Computing and Informatics

BCS 362 – Generic Programming with C++

Chapter 5.2 – Vectors

1 Introduction

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.

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 onward)
```

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.

1.0.1 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 dont 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 wont 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.

1.0.2 Vectors remember their length

Unlike built-in dynamic arrays, which dont know the length of the array they are pointing to, std::vector keeps track of its length. We can ask for the vectors 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;
}
```

1.0.3 Resizing an vector

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;
}
```

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:

```cpp
#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';

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

    return 0;
}
```

Resizing a vector is computationally expensive, so you should strive to minimize the number of times you do so.

1.0.4 Length vs capacity

Consider the following example:

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

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

However, what if we only wanted to iterate over the elements weve initialized, reserving the unused ones for future expansion? In that case, wed 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, whereas capacity is how many elements were allocated in memory.

For Example

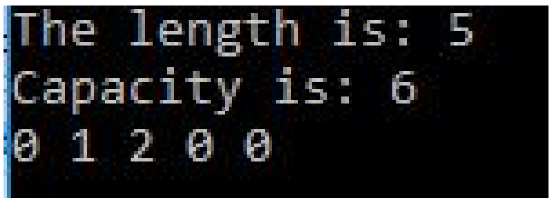
```cpp
#include <vector>
#include <iostream>

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 << "Capacity is: " << array.capacity() <<std::endl;
    for (auto const &element: array)
        std::cout << element << ' ';

    return 0;
}
```

The output of this code is shown in Figure 1
In the above example, weve used the resize() function to set the vectors length to 5. This tells variable array that were 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?



```
The length is: 5
Capacity is: 6
0 1 2 0 0
```

Figure 1: Output showing difference between length and 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.

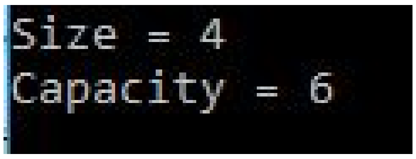
When we used push_back() to add a new element, our vector only needed room for 6 elements, but allocated room for 7. This was done so that if we were to push_back() another element, it wouldnt need to resize immediately, see the code below and corresponding output shown in Figure 2

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

using namespace std;

int main()
{
  vector<int> data(3);
  data.push_back(2);
  cout << "Size = " << data.size() << endl;
  cout << "Capacity = " << data.capacity() << endl;

  return 0;
}
```



```
Size = 4
Capacity = 6
```

Figure 2: Output showing capacity after code above has executed

push_back() added some 2 more slots to the vector capacity.

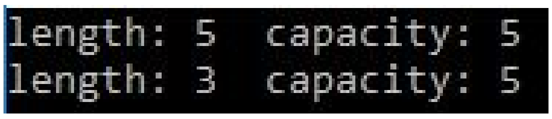
Why differentiate between length and capacity? std::vector will reallocate its memory if needed, but, it would prefer not to, because resizing an array is computationally expensive. Consider the following:

```
#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;
}
```



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

Figure 3: Output showing difference between length and capacity

The output to this code is shown in Figure 3
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.

Table 1 shows functions and there respective use in vectors

Table 1: Functions in std::vector and their respective functionality

Function	**Functionality (What id does)**
size()	returns the size of a vector (how many elements it contains)
capacity()	returns the number of elements that can be held in currently allocated storage
reserve(int)	reserves/allocates memory that can store specified number of elements
clear()	clears the contents
insert(location, value)	insert specified value at the specified location

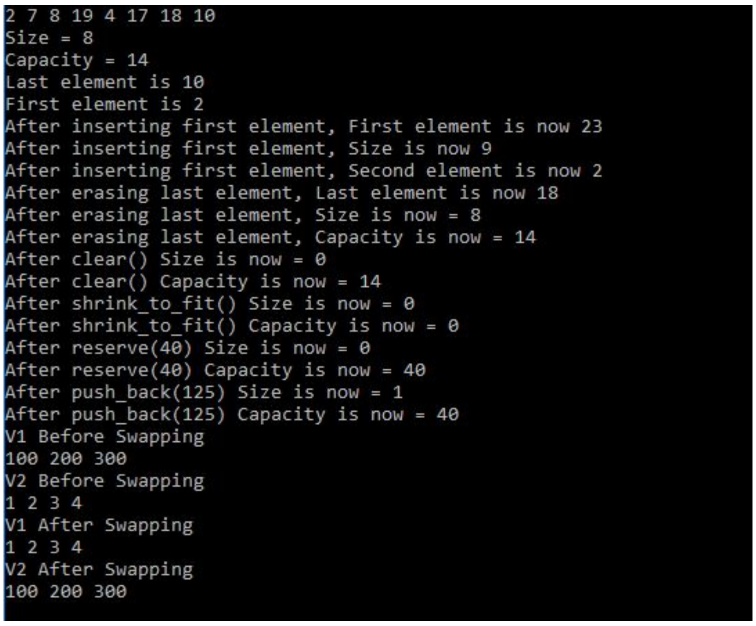
erase(iterator)
erase(start, stop)
	erase all or specified contents

push_back(value)	adds an element at end of vector
pop_back()	removes the last element
swap(vector)	swap the contents of one vector with the other
shrink_to_fit()	reduces memory usage by freeing unused memory
back()	returns the last element
front()	returns the first element
at(int)	access specified element with bound checking
operator[int]	access specified element without bound checking
begin()	return an iterator to the beginning of the vector
end()	return an iterator to the end of the vector

The code below shows how to use some of these functions. Figure 4 shows the output of the code above.

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
  vector<float> v1{100, 200, 300};
  vector<float> v2{1, 2, 3, 4};

  vector<int> data = {2, 7, 8, 19, 4, 17, 18};
  data.push_back(10);
  //lets display elements in the vector
  for(auto it = data.begin(); it != data.end(); it++)
    cout << *it << " ";
  cout << endl;
  cout << "Size = " << data.size() << endl;
  cout << "Capacity = " << data.capacity() << endl;
  cout << "Last element is " << data.back() << endl;
  cout << "First element is " << data.front() << endl;
  data.insert(data.begin(), 23);
  cout << "After inserting first element, First element is now " << data.front() << endl;
  cout << "After inserting first element, Size is now " << data.size() << endl;
  cout << "After inserting first element, Second element is now " << data.at(1) << endl;
  data.erase(data.end() - 1); //Lets erase the last element
  cout << "After erasing last element, Last element is now " << data.back() << endl;
  cout << "After erasing last element, Size is now = " << data.size() << endl;
  cout << "After erasing last element, Capacity is now = " << data.capacity() << endl;
  data.clear(); //cleans the vector deleting all elements
  cout << "After clear() Size is now = " << data.size() << endl;
  cout << "After clear() Capacity is now = " << data.capacity() << endl;
  data.shrink_to_fit();
  cout << "After shrink_to_fit() Size is now = " << data.size() << endl;
  cout << "After shrink_to_fit() Capacity is now = " << data.capacity() << endl;
  data.reserve(40);
  cout << "After reserve(40) Size is now = " << data.size() << endl;
  cout << "After reserve(40) Capacity is now = " << data.capacity() << endl;
  data.push_back(125);
  cout << "After push_back(125) Size is now = " << data.size() << endl;
  cout << "After push_back(125) Capacity is now = " << data.capacity() << endl;
  cout << "V1 Before Swapping \n" ;
  for(auto i = v1.begin(); i != v1.end(); i++)
    cout << *i << " ";
  cout << endl;
  cout << "V2 Before Swapping \n" ;
  for(auto i = v2.begin(); i != v2.end(); i++)
    cout << *i << " ";
  cout << endl;
  v1.swap(v2); //lets swap these vectors (notice that they dont need to be of the same size)
  cout << "V1 After Swapping \n" ;
  for(auto i = v1.begin(); i != v1.end(); i++)
    cout << *i << " ";
  cout << endl;
  cout << "V2 After Swapping \n" ;
  for(auto i = v2.begin(); i != v2.end(); i++)
    cout << *i << " ";
  cout << endl;
}
```



```
2 7 8 19 4 17 18 10
Size = 8
Capacity = 14
Last element is 10
First element is 2
After inserting first element, First element is now 23
After inserting first element, Size is now 9
After inserting first element, Second element is now 2
After erasing last element, Last element is now 18
After erasing last element, Size is now = 8
After erasing last element, Capacity is now = 14
After clear() Size is now = 0
After clear() Capacity is now = 14
After shrink_to_fit() Size is now = 0
After shrink_to_fit() Capacity is now = 0
After reserve(40) Size is now = 0
After reserve(40) Capacity is now = 40
After push_back(125) Size is now = 1
After push_back(125) Capacity is now = 40
V1 Before Swapping
100 200 300
V2 Before Swapping
1 2 3 4
V1 After Swapping
1 2 3 4
V2 After Swapping
100 200 300
```

Figure 4: Output showing difference between length and capacity