7.10 — std::vector capacity and stack behavior

BY ALEX ON NOVEMBER 24TH, 2015 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In lesson **6.16 -- 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, whereas **capacity** 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 const &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
```

```
8
9       std::cout << "The length is: " << array.size() << '\n';
10      std::cout << "The capacity is: " << array.capacity() << '\n';
11  }
```

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:

```
1   #include <vector>
2   #include <iostream>
3
4   int main()
5   {
6   std::vector<int> array;
7   array = { 0, 1, 2, 3, 4 }; // okay, array length = 5
8   std::cout << "length: " << array.size() << "  capacity: " << array.capacity() << '\n';
9
10  array = { 9, 8, 7 }; // okay, array length is now 3!
11  std::cout << "length: " << array.size() << "  capacity: " << array.capacity() << '\n';
12
13  return 0;
14  }
```

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 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 (const 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() 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 (const 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()
```

```
5  {
6      std::vector<int> v = { 0, 1, 2, 3, 4 };
7      std::cout << "size: " << v.size() << "  cap: " << v.capacity() << '\n';
8
9      v.push_back(5); // add another element
10     std::cout << "size: " << v.size() << "  cap: " << v.capacity() << '\n';
11
12     return 0;
13 }
```

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.

 **7.11 -- Recursion**

 **Index**

 **7.9 -- The stack and the heap**

C++ TUTORIAL | PRINT THIS POST

121 comments to 7.10 — std::vector capacity and stack behavior

« Older Comments 1 2

Shahrooz Leon
December 16, 2019 at 9:31 am · Reply

Hi , thanks to your awesome tutorial, its really great.
what if we want to create a adjacency list for a graph and using a vector of vector.
then how I can access the outer vector elements and inner vector elements?
then how can I fill my inner vector with numbers by a loop.
and when and where I should use iterator, I've been reading solutions for this issue on many sites and videos and I got confused.
please give me an idea and tell me where I should use iterator , because every time I use iterator in vector of vector my program crashes in middle of compile. please help me.
so much thanks . I would be grateful if you send your answer to my email so I can read it as soon as you write.
shahroozleon@outlook.com

nascardriver
December 17, 2019 at 2:18 am · Reply

You shouldn't use a vector of vector for an adjacency list. Each entry of the list consists of a node and all of its neighboring nodes. A vector would only give you a list of nodes, without the separation into _node_ and _neighbors_.
Rather than a vector of vector, you can use a map of vector. `std::map` isn't covered on learncpp.

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

int main() {
  // Assuming your nodes are ints
  std::map<int, std::vector<int>> graph{
      {1, {2, 3}}, // 1 has neighbors 2 and 3
      {2, {1}},    // 2 has neighbor 1
      {3, {1, 4}}, // 3 has neighbor 1 and 4
      {4, {3}}     // 4 has neighbor 3
  };

  /*
  1 --- 2
  |
  |
  3 --- 4
  */

  std::cout << "nodes:\n";

  for (const auto& node : graph) {
      // @first is the key (node)
      // @second is the value (vector of neighbors)
      std::cout << node.first << ' ';
  }

  std::cout << '\n';

  std::cout << "neighbors of 3:\n";

  // graph[3] returns the neighbors of node 3. Not at index 3.
  for (int neighbor : graph[3])
  {
      std::cout << neighbor << ' ';
  }

  std::cout << '\n';

  return 0;
}
```

Output

```
nodes:
1 2 3 4
neighbors of 3:
1 4
```

You could use a vector of struct. That's not as easy to work with as a map.

Another alternative is a custom type

```
struct SNode
{
  int iNode{};
  std::vector<int> vNeighbors{};
};

std::vector<SNode> graph{
  {1, {2, 3}},
  {2, {1}},
  {3, {1, 4}},
  {4, {3}}
};

// We can't access the nodes directly, we'd have to search
// graph[3] is an access to index 3, not to node 3.
```

A
September 22, 2019 at 2:05 am · Reply

Hello, I have some questions about vectors and the push_back() function.

I have a program where I use an array of pointers pointing to different items in a vector, but this program is not behaving as I would expect. I have written a program that fundamentally does the same thing (or at least show the same unwanted behavior)

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

class C //just a generic class
{
private:
    int m_int;

public:
    C() { static int id(5); m_int = id; id++; } //to differentiate the classes

    friend std::ostream& operator<<(std::ostream &out, const C &c); //for printing
};

std::ostream& operator<<(std::ostream &out, const C &c) //for printing
{
    out << c.m_int;
    return out;
}

int main()
{
    std::array<C*, 2> ptrArray; //static array with pointers to class C
    std::vector<C> classArray;    //vector with actual C


    for (int i(0); i < ptrArray.size(); i++)
    {
        classArray.push_back(C()); //add element to classArray
        std::cout << "Class: " << classArray.back() << ' ' << &classArray.back() << '\n';

        ptrArray[i] = &classArray.back(); //point to the last element added
        std::cout << "Pointer: " << *ptrArray[i] << ' ' << ptrArray[i] << "\n\n";
    }

```

```
37      //print the whole thing
38      std::cout            << "Class: "    << classArray[0] << ' ' << &classArray[0] << '\n'
39                  << "Pointer: "    << *ptrArray[0]  << ' ' << ptrArray[0]      << "\n\n"
40                  << "Class: "    << classArray[1] << ' ' << &classArray[1] << '\n'
41                  << "Pointer: "    << *ptrArray[1]  << ' ' << ptrArray[1];
42      return 0;
43  }
```

What I want is for the ptrArray elements to always point to same elements in classArray, however when I run this program it outputs:
Class: 5 0013D8B0
Pointer: 5 0013D8B0

Class: 6 0013FF04
Pointer: 6 0013FF04

Class: 5 0013FF00
Pointer: -572662307 0013D8B0

Class: 6 0013FF04
Pointer: 6 0013FF04

At first the program seems to be doing what I want it to, the pointers are pointing to the correct addresses with the correct class in them. When printing it all again I realize that's not the case.
The first class seems to have been moved from its previous address causing the pointer to it to become dangling. The second class and pointer is doing what I want them to do but as soon as I .push_back() another element to classArray they show the same behaviour.

I realize this is because the push_back() function does not work as i thought it would work and it is clear that it doesn't care at all about my pointers and keeping everything at the same memory location, which is what I need it to do for my program to work. So what I really am asking is if there's a plug-and-play container class somewhere that keeps everything at the same location while still having the stack behavior of std::vector. Alternatively if there is a way for me to use std::vector but avoid this problem with some other solution.

Thanks in advance.

nascardriver
September 22, 2019 at 2:15 am · Reply

`std::vector` is dynamically sized and has to re-allocate its internal array when the capacity is exceeded. You can use `std::vector::reserve` to reserve memory before you push anything. As long as you don't exceed the capacity, the array doesn't have to be re-allocated.
If you only need 2 elements:

```
1  classArray.reserve(2);
2
3  // loop (push_back won't resize unless you push more than 2 elements).
```

A
September 22, 2019 at 2:29 am · Reply

The thing is that I have no idea how many elements will be required in my program, and I assume it is bad practice to just reserve a very large number at the entry point of the program. I was gonna delete the comment anyway because std::list seems to be what I'm looking for but you were just too fast with replying.

Jonathan

August 10, 2019 at 5:43 am · Reply

Just want to share my code to delete vector element. Correct me if there are any suggestions!!

```
void printVector(const std::vector<int>& vector)
{
    std::cout << "The remaining vector is: ";
    for (int v{ 0 }; v < vector.size(); ++v)
    {
        std::cout << vector[v] << ' ';
    }
    std::cout << '\n';
}
void initializeVector(std::vector<int>& vector)
{
    for (int i{ 0 }; i < vector.size(); ++i)
    {
        std::cout << "Enter #" << i + 1 << ": ";
        std::cin >> vector[i];
    }
}

void initializeTempVector(const int& del, std::vector<int>& tempVector, const std::vector
<int>& vector)
{
    for (int j{ 1 }; j <= tempVector.size();++j )
        tempVector[j] = vector[del + j];
}
void vectorPopBack(const int &length, const int &del, std::vector<int>& vector)
{
    for (int k{ 1 }; k <= length - del; ++k)
        vector.pop_back();
}
void restoreVector(const int& tempLength, const std::vector<int>& tempVector, std::vector
<int>& vector)
{
    for (int u{ 1 }; u < tempLength; ++u)
        vector.push_back(tempVector[u]);
}
void invalidInput(int& ref)
{
    std::cout << "Invalid input, please enter again: ";
    std::cin >> ref;
}
void checkValidityLength(int& length)
{
    while (std::cin.fail())
    {
        std::cin.clear();
        std::cin.ignore(32767, '\n');
        invalidInput(length);
    }
    while (length < 1)
        invalidInput(length);
}
void checkValidityDel(int& del, const int& length)
{
    while (std::cin.fail())
    {
        std::cin.clear();
        std::cin.ignore(32767, '\n');
        invalidInput(del);
```

```
        --del;
    }
    while (del < 0 || del >= length)
    {
        invalidInput(del);
        --del;
    }
}
void deleteElement(const int &length, const int& del, std::vector<int>& vector)
{
    int tempLength{ length - del };
    std::vector<int> tempVector(tempLength);
    tempVector[0] = 0;

    initializeTempVector(del, tempVector, vector);
    vectorPopBack(length, del, vector);
    restoreVector(tempLength, tempVector, vector);
}
bool continueOrNot()
{
    char yn{ 'a' };
    while ((yn != 'y') && (yn != 'n'))
    {
        std::cout << "Do you want to delete other elements? (y/n)";
        std::cin >> yn;
    }
    if (yn == 'y')
        return true;
    else
        return false;

}
int main()
{
    std::cout << "Tell me how many numbers do you want to enter: ";
    int length{ 0 };
    std::cin >> length;
    checkValidityLength(length);

    std::vector<int> vector(length);
    initializeVector(vector);

    bool cont{ true };
    while (cont)
    {
    int del{ 0 };
    std::cout << "Which element do you want to delete? (#1, 2, ...)";
    std::cin >> del;
    --del;
    checkValidityDel(del, length);

    deleteElement(length, del, vector);
    printVector(vector);
    --length;
    if (length == 0)
    {
        std::cout << "No more numbers left!\n";
        break;
    }
    cont = continueOrNot();
    }
    std::cout << "Thank You!";
```

```
    return 0;
}
```

nascardriver
August 10, 2019 at 8:30 am · Reply

- Limit your lines to 80 characters in length for better readability on small displays.
- Don't pass 32767 to @std::cin.ignore. Pass @std::numeric_limits<std::streamsize>::max().
- You're using different iterator names everywhere. Variables are local, you can and should re-use names.
- Pass fundamental types (int) by value.
- Line 21+: `tempVector[tempVector.size()]` doesn't exist. This causes undefined behavior.
- Inconsistent formatting. Use your editor's auto-formatting feature.
- Line 69 has no effect. All elements are default-initialized.
- Line 78, 83: Duplicate comparison.
- Line 83: `return (yn == 'y')`.
- Line 100: Should be a do-while loop.
- If your program prints anything, the last thing it prints should be a line feed.
- Lots of signed/unsigned comparisons and conversions.
- You're not resizing the vector. This is a waste of memory and now you vector only works in combination with `length`.
- Line 80+: Should be a do-while loop.

Your idea is somewhat correct, but the implementation is rather rough.
For one, you could've used `std::vector::erase`. Since this replaces your entire code, I suppose you either didn't know about it or wanted to try implementing it yourself.
There's no need to create a new vector. All you need to do is shift every element after the one that's being deleted to the left.

```
2 9 2 0 1 2 8 4
//    ^ delete this

2 9 2   1 2 8 4

// shift everything that was to the right one to the left
2 9 2 1 2 8 4 4

// resize the vector to remove the last element
2 9 2 1 2 8 4
```

Here's a version of your program I wrote. Try to understand it. If you don't understand something or why I did something, please ask.
I used `std::vector::empty` and `std::vector::resize`. C++ is well documented, you can look up classes on cppreference (https://en.cppreference.com/w/cpp/container/vector) to see what they can do and how to use them.

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

// Pass @iMin and @iMax for reusability
int readInt(
    int iMin = std::numeric_limits<int>::min(),
    int iMax = std::numeric_limits<int>::max())
{
  while (true)
  {
    int iInput{};
    std::cin >> iInput;

    bool bSuccess{ true };
```

```
15
16      if (std::cin.fail())
17      {
18        std::cin.clear();
19        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
20      }
21
22      if ((iInput < iMin) || (iInput > iMax))
23      {
24        bSuccess = false;
25      }
26
27      if (bSuccess)
28      {
29        return iInput;
30      }
31      else
32      {
33        std::cout << "Invalid input, please enter again: ";
34      }
35    }
36  }
37
38  std::vector<int> readVector(int iLength)
39  {
40    std::vector<int> vecInput(static_cast<std::vector<int>::size_type>(iLength));
41
42    for (int i{ 0 }; i < iLength; ++i)
43    {
44      std::cout << "Enter #" << i + 1 << ": ";
45      vecInput[static_cast<std::vector<int>::size_type>(i)] = readInt();
46    }
47
48    return vecInput;
49  }
50
51  // We don't need to pass the length, it's stored in the vector.
52  void printVector(const std::vector<int> vec)
53  {
54    std::cout << "The remaining vector is: ";
55
56    // We don't need the index, use a for-each loop.
57    for (int i : vec)
58    {
59      std::cout << i << ' ';
60    }
61
62    std::cout << '\n';
63  }
64
65  bool shouldContinue(void)
66  {
67    while (true)
68    {
69      std::cout << "Do you want to delete other elements? (y/n)";
70      char ch{};
71      std::cin >> ch;
72
73      // Limited number of options -> switch.
74      switch (ch)
75      {
76      case 'n':
```

```
77        return false;
78      case 'y':
79        return true;
80      }
81    }
82  }
83
84  void deleteElement(std::vector<int>& vec, int iIndex)
85  {
86    // What you'd usually do
87    // vecInput.erase(vecInput.begin() + iIndex);
88
89    // If you want to implement it yourself
90
91    // Shift all elements that were on the right one index to the left.
92    for (auto i{ static_cast<std::vector<int>::size_type>(iIndex) };
93         i < (vec.size() - 1);
94         ++i)
95    {
96      vec[i] = vec[i + 1];
97    }
98
99    // Remove the last element by resizing the vector
100   vec.resize(vec.size() - 1);
101 }
102
103 int main(void)
104 {
105   std::cout << "Tell me how many numbers do you want to enter: ";
106
107   int iLength{ readInt(1) };
108   auto vecInput{ readVector(iLength) };
109
110   do
111   {
112     std::cout << "Which element do you want to delete? (#1, 2, ...)";
113
114     int iDelete{ readInt(1, iLength + 1) - 1 };
115
116     deleteElement(vecInput, iDelete);
117
118     printVector(vecInput);
119
120     if (vecInput.empty())
121     {
122       std::cout << "No more numbers left!\n";
123       break;
124     }
125   } while (shouldContinue());
126
127   std::cout << "Thank You!\n";
128
129   return 0;
130 }
```

Jonathan
August 10, 2019 at 10:29 pm · Reply

Thank you so much for writing those codes, I understand most of the code but I have a few questions:

1. Why don't we just write vecInput[i] in line 45?
2. How is type "std::vector<int>::size_type" different from type "unsigned int" ?
3. line 108 shouldn't it be

```
1  int iDelete{ readInt(1, iLength ) - 1 };
```

instead? If length = 3 which holds 3 elements (0, 1, 2) the threshold "1 <= length <= 4" which passes to "readInput()"accept iDelete (0, 1, 2, 3)

I also changed my code base on your code!!

```
1  #include <iostream>
2  #include <vector>
3  int getInput(int min = std::numeric_limits<int>::min(), int max = std::numeric_l
4  imits<int>::max())
5  {
6      while (true)
7      {
8          int input{ 0 };
9          std::cin >> input;
10         if (std::cin.fail())
11         {
12             std::cin.clear();
13             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
14             input = min - 1;
15         }
16         if ((input < min) || (input > max))
17         {
18             std::cout << "Invalid input, please enter again: ";
19         }
20         else
21             return input;
22     }
23 }
24 void printVector(const std::vector<int>& vector)
25 {
26     std::cout << "The remaining vector is: ";
27     for (int j{ 0 }; j < vector.size(); ++j)
28         std::cout << vector[j] << ' ';
29
30     std::cout << '\n';
31 }
32 void initializeVector(std::vector<int>& vector)
33 {
34     for (int i{ 0 }; i < vector.size(); ++i)
35     {
36         std::cout << "Enter #" << i + 1 << ": ";
37         vector[i] = getInput();
38     }
39 }
40 void deleteElement(std::vector<int>::size_type del, std::vector<int>& vector)
41 {
42     for (; del < vector.size(); ++del)
43     {
44         vector[del] = vector[del + 1];
45     }
46     vector.pop_back();
47 }
48 bool shouldContinue()
49 {
50     while (true)
51     {
52         std::cout << "Do you want to delete other elements? (y/n)";
```

```
53          char yn{ 'a' };
54          std::cin >> yn;
55
56          switch (yn)
57          {
58          case 'n':
59              return false;
60          case 'y':
61              return true;
62          default:
63              std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
64          }
65      }
66  }
67  int main()
68  {
69      std::cout << "How many numbers do you want to enter? ";
70      int length{ getInput(1) };
71
72      std::vector<int> vector(length);
73      initializeVector(vector);
74
75      bool cont{ true };
76      do
77      {
78          std::cout << "Which element do you want to delete? (#1, 2, ...)";
79          int del{ getInput(1, length) - 1 };
80
81          deleteElement(del, vector);
82          printVector(vector);
83
84          if (vector.empty())
85          {
86              std::cout << "No more numbers left!\n";
87              break;
88          }
89          cont = shouldContinue();
90      } while (cont);
91      std::cout << "Thank You!\n";
92
93      return 0;
    }
```

nascardriver
August 10, 2019 at 11:39 pm · Reply

1/2.
`std::vector::operator[]` wants an `std::vector::size_type` (https://en.cppreference.com/w/cpp/container/vector/operator_at).
The only thing we know about `std::vector::size_type` is that it's an unsigned integer type (https://en.cppreference.com/w/cpp/container/vector). We don't know which type exactly. In your standard implementation, it might be an `unsigned int`, but that's not what it is everywhere.

3.
Yes it should, well spotted!

> I also changed my code base on your code!!
- Line 45: That's a nice idea.
- Line 13: This can (very likely) cause an integer underflow. Wrap line 15-20 in an `else` block

instead.
- Line 52: Should be `char yn{}`, because you're not using the initial value.
- You can use `shouldContinue()` as the loop's condition, then you don't need `cont`.

Looks good now, keep on posting your solutions :-)

Jonathan
August 11, 2019 at 12:29 am · Reply

Sure! Thanks!

Nguyen
April 16, 2019 at 7:00 pm · Reply

Hi nascardriver,

This lesson seems tough for me. I was wondering if I can safely skip it for now and move to the next lessons or chapter 8?

Thanks, Have a great day

nascardriver
April 17, 2019 at 2:56 am · Reply

You can skip it for now (and continue at 7.11), but return to it later.

Matt
August 7, 2019 at 7:17 pm · Reply

Nguyen,

For me it is helpful to think of the std::vector like a bucket of water.

capacity() tells you how big the bucket is (thus, how much water it can possibly hold)
size() tells you how much water is currently in the bucket
resize() tells the bucket that there is more water in it, and so the bucket will automatically get larger so it can hold all that extra water (maybe some extra)
push_back() adds some water to the bucket, if the bucket is full it will automatically resize() the bucket to make room
back() takes a look in the bucket and sees what the water at the top looks like, but it doesn't pour the water out
pop_back() pours some water out of the bucket, the bucket will not necessarily get smaller just because we poured water out, though

I'm not an expert but this is the way my brain works, hope it helps!
-Matt

Lakshya Malhotra
March 22, 2019 at 12:12 pm · Reply

Hi,
So according to the discussion on the topic above: "More length vs. capacity", it was discussed that "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."
But when I try to do something like this:

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

int main()
{
    std::vector<int> array;
    array = {0, 1, 2, 3, 4};

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

    array = {9, 7, 8};
    std::cout << "length: " << array.size() << "   capacity: " << array.capacity() << '\n'
;
    std::cout << "2nd element: " << array[1] << '\n'; //<< "   capacity: " << array.capaci
ty() << '\n';
    std::cout << "4th element: " << array[3] << '\n';
    for (const auto &element : array)
        std::cout << element << '\n';
}
```

I would assume that this line

```
   std::cout << "4th element: " << array[3] << '\n';
```

would give me errors becuase the compiler knows only the first three elements are valid this time. But I am getting output :
```
length: 5   capacity: 5
length: 3   capacity: 5
2nd element: 7
4th element: 3
9
7
8
```
Can you explain what is going on here? Thanks in advance!!

nascardriver
March 23, 2019 at 8:30 am · Reply

@std::vector::operator[] doesn't perform bounds checks. As long as you don't leave the capacity, there shouldn't be any errors.
If you don't trust yourself with keeping track of the indexes and length, you can use @std::vector::at, which does perform a bounds check.

NXPY
March 21, 2019 at 11:35 pm · Reply

Hi Alex !

I just want to point out that you forgot to include the iostream header file in the last code .

Also in this code :

```
#include<vector>
```

```
#include <iostream>
using namespace std ;
int main()
{
    vector<int> v = {0,1,2,3,4} ;

    std::cout << "size" << v.size() << "  cap :" << v.capacity() << "\n" ;

    v.push_back(5) ;
    std::cout << "size" << v.size() << "  cap :" << v.capacity() << "\n" ;

    v[11] = 5 ;
    std::cout <<v[11] << "\n" ;
    std::cout << "size" << v.size() << "  cap :" << v.capacity() << "\n" ;
}
```

The capacity changes to 10 when I resize it .
The capacity or the length of the vector v does not change when I access v[11] . So how is it stored ? Also if I try to access v[12] the program stops responding even though it had no problem with v[11]. Why is this so ?

nascardriver
March 22, 2019 at 8:13 am · Reply

> The capacity changes to 10 when I resize it
@std::vector::push_back can increase the capacity as much as is likes, there's no standard.

> The capacity or the length of the vector v does not change when I access v[11]
Access operations on vectors don't create new elements. @std::vector::operator[] doesn't perform any bound checks on the given index. You're accessing memory which you don't own, causing undefined behavior. Both with v[11] and v[12].

NXPY
March 23, 2019 at 4:44 am · Reply

Thank you for the reply !
But is there any particular reason accessing v[11] does not cause it to crash but accessing v[12] or beyond does ?

nascardriver
March 23, 2019 at 7:40 am · Reply

Not that I can think of.

Aqua
December 31, 2018 at 11:22 am · Reply

Thank you for these tutorials and explanations. std::vector<> is such an incredible tool. My only question would be: If needed is it possible to, more efficient, if we resized the capacity ourselves via reserve() if we hit the maximum capacity? So far I understand vector is dynamic. Resizing a vector is costly. vector has stack like qualities? Although since a vector is dynamic it is allocated on the heap? hmmm. I guess those are also questions to better my understanding. new is the only call to allocate memory on the heap and everything else uses the stack?

nascardriver

January 1, 2019 at 5:20 am · Reply

Hi!

Use of heap and stack isn't standardized. However, it's common for everything allocated dynamically (eg. using @new or @malloc), to be allocated on the heap and everything else on the stack.

If you know beforehand how big your vector is going to be, you should reserve space.
If you don't know how big the vector is going to be, there's no benefit in resizing it manually.

jasonguo
December 10, 2018 at 6:23 am · Reply

So, the capacity is basically the largest length the stack had? Is it not possible to shorten the capacity?

Alex
December 10, 2018 at 12:41 pm · Reply

It is possible to shorten the capacity. Maybe the best way to do so is via the shrink_to_fit function (introduced in C++11), which is a non-binding (meaning it may or may not work) method to reduce the capacity down to the size.

Ahmed
December 2, 2018 at 5:50 pm · Reply

If you let me, What's the point of using std::vector with the stack while I can use built-in arrays or std::array?
Is it just for performance reasons? that it doesn't resize (or I can control the capacity)which it is expensive for the CPU.

nascardriver
December 3, 2018 at 11:00 am · Reply

Built-in arrays have no functionality, you have to do everything yourself. @std::array cannot be resized and the size needs to be known at compile-time.

Nigel Booth
August 28, 2018 at 3:02 am · Reply

Hi,

The stack commands - pop(), push(), peek() - sounded awfully familiar. pop and push, I suppose, are similar to their assembly language counterparts but peek took a little more thought. I don't suppose it is related in any way to the PEEK command used in early BASIC (home computers, UNIX systems) from the 80's? This, however, would then require a POKE to insert a character into a specific memory address? It was a long time ago since I played with BASIC (Commodore Amiga I seem to recall).

nascardriver
August 28, 2018 at 8:12 am · Reply

Hi Nigel!

push and pop in assembly are used to control the stack, so yes, they should sound familiar. I don't know BASIC, but from your description PEEK sounds like a function to access random memory, whereas here it's used to read the top of the stack without popping.

Liam
June 19, 2018 at 4:20 am · Reply

Why is it that some functions are formatted like "array.function()" whereas others are formatted like "function(array)" ?

I know that the answer to this question is written somewhere in these tutorials, but I can't find it.

Liam
June 19, 2018 at 6:10 pm · Reply

Never mind--it's all there in the next chapter!

Peter Baum
May 22, 2018 at 12:07 pm · Reply

It was only later in the lessons that I realized that there is something fundamental that I didn't understand at this point. std::array was introduced and that used the stack and then "new" was used and that used the heap. But suppose I want to use std::array and make sure it uses the heap. How would I do that? An example that does this and allows access from a function would be very helpful.

Also I saw a comment at stackoverflow awhile back about the stack and heap in which someone said that from the point of view of CPP there is no stack or heap. In trying to find that discussion (which I didn't) I ran across https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap which is very good, and learned that individual threads get their own stack. In that discussion (down a ways) I found "you are free to implement a compiler that doesn't even use a stack or a heap, but instead some other storage mechanisms." and that implies the disconnect between CPP and these storage areas/techniques. In any case, as a practical matter, IMHO in these lessons, you really can't easily and should not in any case, separate CPP from important OS considerations like this.

nascardriver
May 22, 2018 at 12:41 pm · Reply

Hi Peter!

> But suppose I want to use std::array and make sure it uses the heap. How would I do that?
You pretty much answered your own question in your quote "you are free to implement a compiler that doesn't even use a stack or a heap, but instead some other storage mechanisms.".
The standard doesn't require compiler developers to use the stack for X and the heap for Y, they're free to do whatever they want.
If you need control over heap and stack you'll have to use compiler/OS/CPU specific solutions.

Maxime
May 22, 2018 at 1:21 pm · Reply

Won't the "new" keyword make any data reside in the heap? If not I misunderstood some of the matter.

Alex
May 22, 2018 at 8:25 pm · Reply

If you want std::array to use the heap, then you need to dynamically allocate it on the heap in the first place:

```
std::array<int, 5> *array = new std::array<int, 5>;
```

However, I'd advise against this, as using std::vector is a better choice in such a case (as std::vector will clean itself up, whereas std::array on the heap relies on you to remember to do so).

Peter Baum
May 25, 2018 at 8:32 am · Reply

Hi Alex,
I tried something like that and ran into a problem with Visual Studio Community 2017. Maybe I'm just doing something stupid, but this was the code:

```
#include "stdafx.h"
#include <iostream>
#include <array>

void printit(std::array<int, 5> array)
    {
    std::cout << array[2]<<std::endl;
}
int main()
{
    std::array<int, 5> *array = new std::array<int, 5>;
    array->at(2) = 6; // this works and does bounds checking
    array[2] = 6; // this give a compile error = "no operator = matches these op
erands.
    // Operand types are std::array<int,5U> = int
    printit(*array);
    return 0;
}
```

The at() function worked but not the usual array element access method with square brackets.

Also, I was avoiding vector because of the behind the scenes garbage collection and I was, after all, trying to do timing.

nascardriver
May 25, 2018 at 11:07 am · Reply

@array is a pointer, you can't use pointer operators like that.

```
array->operator[](2); // This should work, I prefer using @at for pointers.
```

Peter Baum
May 27, 2018 at 3:41 pm · Reply

Hi nascardriver,

I certainly hadn't tried that!!! Where do you come up with this stuff!?!? Did I miss a lesson along the way?

I am still confused though, because we saw in a previous lesson that we could do

```
int *array = new int[length]; // use array new.  Note that length does n
ot need to be constant!
    array[0] = 5; // set element 0 to value 5
```

and here "array" is also a pointer. So what is the difference?

nascardriver
May 28, 2018 at 1:32 am · Reply

> Where do you come up with this stuff!?!? Did I miss a lesson along the way?
I've been coding for a couple of years, learncpp doesn't cover every tiny piece of cpp.

> So what is the difference?
All pointers have the operator[], but it treats the underlying data as an array.

Let 'a' be an int pointer (or array), 'b' an integer.
The following rules apply (The casts make it look more complicated than it is)

```
(a + b) = *reinterpret_cast<int *>(reinterpret_cast<std::uintptr_t>(
a) + sizeof(int) * b) = (b + a)

a[b] = *(a + b) = b[a]
```

Applying this to your code

```
std::array<int, 5> *array{ new std::array<int, 5> };

// The following three are equivalent
array[0];
*array;
0[array]; // Don't ever do this please.

// This is valid, but might be slower than array->at(2), because w
e're dereferencing.
array[0].at(2);

// You can also do this, but there's no point in doing so,
// because you'll end up with an undefined value.
array[3];
```

Peter Baum
May 28, 2018 at 9:05 am · Reply

Hi nascardriver,

I followed what you said and it is helpful with respect to your alternative ways of accessing the array element. However, what I don't understand is why, in my original post, you get a compiler error with

```
array[2] = 6;
```

but in the printit() routine it is valid to ask for cout to output element array[2].

nascardriver
May 28, 2018 at 9:17 am · Reply

@array in @printit is an @std::array, @array in @main is an @std::array pointer.

std::array::operator[] returns an int, but
std::array pointer operator[] returns an std::array.

Peter Baum
May 28, 2018 at 9:35 am · Reply

Ahah... I now see what my problem is... we were taught that an array parameter would not be copied but passed as a pointer. Apparently this is not what happens with a standard array.

What an annoying inconsistency on the part of CPP. I could see adding functions like .at() that provide a special service, but why would the designers of CPP make the array aspect of standard array behave so differently from ordinary arrays?

nascardriver
May 28, 2018 at 9:38 am · Reply

std::array isn't a C-style array so it shouldn't behave like one, it should behave like a custom data type, because that's what it is.

Mireska
April 6, 2018 at 3:38 am · Reply

When I try to access a vector array with an index smaller than it's capacity but larger than its length, I get a runtime error. I know it's bad procedure, but I'd like to understand what's going on; what happens to those indices when the length is shortened? They're in its capacity, so I would assume the vector still has 'control' over them'; I expected undefined behaviour, or maybe it those indices returning the value they once had.

So my question: Why exactly does trying that make it crash? Does the compiler 'limit/restrain' itself from accessing that, even though it technically could, because it knows it's unintended (that's my guess as to why it gives an 'assertion' error)

Many thanks!

nascardriver
April 6, 2018 at 8:02 am · Reply

Hi Mireska!

> Why exactly does trying that make it crash?
It doesn't crash, it's a controlled termination to let you know you're doing something wrong.

> Does the compiler 'limit/restrain' itself from accessing that, even though it technically could, because it knows it's unintended
Yes

Assertions are covered in Lesson 7.12a

« Older Comments 1 2