

Ravesli [Ravesli](#)


- [Уроки по C++](#)
- [OpenGL](#)
- [SFML](#)
- [Qt5](#)
- [RegExr](#)
- [Ассемблер](#)
- [Купить .PDF](#)


Урок №106. Ёмкость вектора

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 8 Сен 2020 |

 24646

[1](#)  [3](#)

Мы уже знаем, что такое `std::vector` в языке C++ и как его можно использовать в качестве **динамического массива**, который запоминает свою длину и длина которого может быть динамически изменена по мере необходимости. Хотя использование `std::vector` в качестве динамического массива — это самая полезная и наиболее часто применяемая его особенность, но он также имеет и некоторые другие способности, которые также могут быть полезными.

Оглавление:

1. [Длина vs. Ёмкость](#)
2. [Оператор индекса и функция `at\(\)`](#)
3. [std::vector в качестве стека](#)
4. [Дополнительная ёмкость](#)

Длина vs. Ёмкость

Рассмотрим следующий пример:

```
1 | int *array = new int[12] { 1, 2, 3, 4, 5, 6, 7 };
```

Мы можем сказать, что длина массива равна 12, но используется только 7 элементов (которые мы, собственно, выделили).

А что, если мы хотим выполнять итерации только с элементами, которые мы инициализировали, оставляя в резерве неиспользованные элементы для будущего применения? В таком случае нам потребуется отдельно отслеживать, сколько элементов было «использовано» из общего количества

выделенных элементов. В отличие от **фиксированного массива** или **std::array**, которые запоминают только свою длину, std::vector имеет два отдельных свойства:

- ➔ **Длина в std::vector** — это количество фактически используемых элементов.
- ➔ **Ёмкость (или «вместимость») в std::vector** — это количество выделенных элементов.

Рассмотрим пример из урока о std::vector:

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> array { 0, 1, 2, 3 };
7     array.resize(6); // устанавливаем длину, равную 6
8
9     std::cout << "The length is: " << array.size() << '\n';
10
11     for (auto const &element: array)
12         std::cout << element << ' ';
13
14     return 0;
15 }
```

Результат выполнения программы:

```
The length is: 6
0 1 2 3 0 0
```

В примере, приведенном выше, мы использовали функцию `resize()` для изменения длины вектора до 6 элементов. Это сообщает массиву, что мы намереваемся использовать только первые 6 элементов, поэтому он должен их учитывать, как активные (те, которые фактически используются). Следует вопрос: «Какова ёмкость этого массива?».

Мы можем спросить std::vector о его ёмкости, используя **функцию capacity()**:

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> array { 0, 1, 2, 3 };
7     array.resize(6); // устанавливаем длину, равную 6
8
9     std::cout << "The length is: " << array.size() << '\n';
10    std::cout << "The capacity is: " << array.capacity() << '\n';
11 }
```

Результат на моем компьютере:

The length is: 6
The capacity is: 6

В этом случае функция `resize()` заставила `std::vector` изменить как свою длину, так и ёмкость. Обратите внимание, ёмкость всегда должна быть не меньше длины массива (но может быть и больше), иначе доступ к элементам в конце массива будет за пределами выделенной памяти!

Зачем вообще нужны длина и ёмкость? `std::vector` может перераспределить свою память, если это необходимо, но он бы предпочел этого не делать, так как изменение размера массива является несколько затратной операцией. Например:

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

Результат выполнения программы:

length: 6 capacity: 6
length: 4 capacity: 6

Обратите внимание, хотя мы присвоили меньшее количество элементов массиву во второй раз — он не перераспределит свою память, ёмкость по-прежнему составляет 6 элементов. Он просто изменил свою длину. Таким образом, он понимает, что в настоящий момент активны только первые 4 элемента.

Оператор индекса и функция `at()`

Диапазон для оператора индекса `[]` и функции `at()` основан на длине вектора, а не на его ёмкости. Рассмотрим массив из вышеприведенного примера, длина которого равна 4, а ёмкость равна 6. Что произойдет, если мы попытаемся получить доступ к элементу массива под индексом 5? Ничего, поскольку индекс 5 находится за пределами длины массива.

Обратите внимание, вектор не будет изменять свой размер из-за вызова оператора индекса или функции `at()`!

`std::vector` в качестве стека

Если оператор индекса и функция `at()` основаны на длине массива, а его ёмкость всегда не меньше, чем его длина, то зачем беспокоиться о ёмкости вообще? Хотя `std::vector` может использоваться как динамический массив, его также можно использовать в качестве **стека**. Мы можем использовать **3 ключевые функции вектора**, которые соответствуют 3-м ключевым операциям стека:

- ➔ функция `push_back()` добавляет элемент в стек.
- ➔ функция `back()` возвращает значение верхнего элемента стека.
- ➔ функция `pop_back()` вытягивает элемент из стека.

Например:

```
1  #include <iostream>
2  #include <vector>
3
4  void printStack(const std::vector<int> &stack)
5  {
6      for (const auto &element : stack)
7          std::cout << element << ' ';
8      std::cout << "(cap " << stack.capacity() << " length " << stack.size() << ")\n";
9  }
10
11 int main()
12 {
13     std::vector<int> stack;
14
15     printStack(stack);
16
17     stack.push_back(7); // функция push_back() добавляет элемент в стек
18     printStack(stack);
19
20     stack.push_back(4);
21     printStack(stack);
22
23     stack.push_back(1);
24     printStack(stack);
25
26     std::cout << "top: " << stack.back() << '\n'; // функция back() возвращает последний элемент
27
28     stack.pop_back(); // функция pop_back() вытягивает элемент из стека
29     printStack(stack);
30
31     stack.pop_back();
32     printStack(stack);
33
34     stack.pop_back();
35     printStack(stack);
36
```

```
37 |     return 0;  
38 | }
```

Результат выполнения программы:

```
(cap 0 length 0)  
7 (cap 1 length 1)  
7 4 (cap 2 length 2)  
7 4 1 (cap 3 length 3)  
top: 1  
7 4 (cap 3 length 2)  
7 (cap 3 length 1)  
(cap 3 length 0)
```

В отличие от оператора индекса и функции `at()`, функции вектора-стека *изменяют* размер `std::vector` (выполняется функция `resize()`), если это необходимо. В примере, приведенном выше, вектор изменяет свой размер 3 раза (3 раза выполняется функция `resize()`: от ёмкости 0 до ёмкости 1, от 1 до 2 и от 2 до 3).

Поскольку изменение размера вектора является затратной операцией, то мы можем сообщить вектору выделить заранее заданный объем ёмкости, используя **функцию `reserve()`**:

```
1  #include <iostream>  
2  #include <vector>  
3  
4  void printStack(const std::vector<int> &stack)  
5  {  
6      for (const auto &element : stack)  
7          std::cout << element << ' ';  
8      std::cout << "(cap " << stack.capacity() << " length " << stack.size() << ")\n";  
9  }  
10  
11 int main()  
12 {  
13     std::vector<int> stack;  
14  
15     stack.reserve(7); // устанавливаем ёмкость (как минимум), равную 7  
16  
17     printStack(stack);  
18  
19     stack.push_back(7);  
20     printStack(stack);  
21  
22     stack.push_back(4);  
23     printStack(stack);  
24  
25     stack.push_back(1);  
26     printStack(stack);  
27  
28     std::cout << "top: " << stack.back() << '\n';
```

```
29
30     stack.pop_back();
31     printStack(stack);
32
33     stack.pop_back();
34     printStack(stack);
35
36     stack.pop_back();
37     printStack(stack);
38
39     return 0;
40 }
```

Результат выполнения программы:

```
(cap 7 length 0)
7 (cap 7 length 1)
7 4 (cap 7 length 2)
7 4 1 (cap 7 length 3)
top: 1
7 4 (cap 7 length 2)
7 (cap 7 length 1)
(cap 7 length 0)
```

Ёмкость вектора была заранее предустановлена (значением 7) и не изменялась в течение всего времени выполнения программы.

Дополнительная ёмкость

При изменении вектором своего размера, он может выделить больше ёмкости, чем требуется. Это делается для обеспечения некоего резерва для дополнительных элементов, чтобы свести к минимуму количество операций изменения размера. Например:

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<int> vect = { 0, 1, 2, 3, 4, 5 };
7      std::cout << "size: " << vect.size() << "   cap: " << vect.capacity() << '\n';
8
9      vect.push_back(6); // добавляем другой элемент
10     std::cout << "size: " << vect.size() << "   cap: " << vect.capacity() << '\n';
11
12     return 0;
13 }
```

Результат на моем компьютере:

```
size: 6 cap: 6  
size: 7 cap: 9
```

Когда мы использовали функцию `push_back()` для добавления нового элемента, то нашему вектору потребовалось выделить комнату только для 7 элементов, но он выделил комнату для 9 элементов. Это было сделано для того, чтобы при использовании функции `push_back()` в случае добавления еще одного элемента, вектору не пришлось опять выполнять операцию изменения своего размера (экономя, таким образом, ресурсы).

Как, когда и сколько выделяется дополнительной ёмкости — зависит от каждого компилятора отдельно.

Оценить статью:

★★★★★ (188 оценок, среднее: **4,93** из 5)



[← Урок №105. Стек и Куча](#)



[Урок №107. Рекурсия и Числа Фибоначчи →](#)

Комментариев: 3



1. *somebox:*
[13 июля 2019 в 21:33](#)

"Это было сделано для того, что, если бы мы использовали `push_back()` для добавления ещё одного элемента, вектору не пришлось бы опять выполнять операцию изменения своего размера (экономя, таким образом, ресурсы)."

То есть `push_back()` увеличивает емкость автоматически?

[Ответить](#)



1. *Анастасия:*
[16 июля 2019 в 14:19](#)

Как я поняла:

- 1) это зависит от компилятора
- 2) если текущей ёмкости не достаточно (при изменении длины) — да, она увеличивается автоматически. Другое дело, что в примере выше она увеличилась не до 7, а сразу до 9, немного впрок, т.к. изменение ёмкости — затратная операция.

[Ответить](#)1. *Алексей:*[2 сентября 2019 в 14:47](#)

Предусмотрительные были ребята.

Думаю, что более новые компиляторы и выделяют больше.

[Ответить](#)

Добавить комментарий

Ваш E-mail не будет опубликован. Обязательные поля помечены *

Имя *

Email *

Комментарий






☐ Сохранить моё Имя и E-mail. Видеть комментарии, отправленные на модерацию

☐ Получать уведомления о новых комментариях по электронной почте. Вы можете [подписаться](#) без комментирования.

[TELEGRAM](#)  [КАНАЛ](#)

[ПАБЛИК](#) 

ТОП СТАТЬИ

-  [Словарь программиста. Сленг, который должен знать каждый кодер](#)
-  [Урок №1. Введение в программирование](#)
-  [70+ бесплатных ресурсов для изучения программирования](#)
-  [Урок №1: Введение в создание игры «SameGame» на C++/MFC](#)
-  [Урок №4. Установка IDE \(Интегрированной Среды Разработки\)](#)

- [Ravesli](#)
- - [О проекте/Контакты](#) -
- - [Пользовательское Соглашение](#) -
- - [Все статьи](#) -
- Copyright © 2015 - 2020