# my::vector documentation

## 简介

这是一个由**我自己**编写的C++ vector模版类，是模仿std::vector类设计的，与std::vector有着相像的功能，包括vector长度的动态分配、自由插入或更改相同类型的元素、迭代器、range-based loop、使用initializer\_list进行初始化和赋值等功能。

* 在此文档中，有自定义的输出[运算符重载](#Xa39a3ee5e6b4b0d3255bfef95601890afd80709)，以及有关的[类型重定义](#Xa39a3ee5e6b4b0d3255bfef95601890afd80709)，请参见附录。
* 在此文档中，所有包含头文件和定义main()函数入口的行为均省略，在使用该类时，需要手动添加头文件，定义main()函数。

## 构造函数

### vector()

这是默认的构造函数。当未指定大小时，该vector为空。（即vector内的首地址与末地址(begin()和end())相等。）

Example Code:

my::vector<int> vect;  
std::cout << vect;

在my::vector类中并没有有关ostream的运算符重载，如需要使用ostream输出一个vector，需要手动重载相关的运算符。在本文档中的相关实现可见[附录](#Xa39a3ee5e6b4b0d3255bfef95601890afd80709)。

Output:

[]

### vector(const int length)

构造指定长度为length的vector。

Example Code:

my::vector<int> vect(5);  
std::cout << vect1;

Output:

[0, 0, 0, 0, 0]

### vector(const self\_type &source)/vector(self\_type &&source)

这两个构造函数分别为Copy Constructor和Move Constructor。创建一个和Source所含内容相同的另一个vector。

需要注意的是，两个vector内部所含有的首末指针所含地址不同，二者是互异的，仅仅只是内容相同

Example Code:

// using default constructor  
my::vector<int> vect1(5);  
vect1.at(0) = 1;  
std::cout << vect1;  
  
// using copy constructor (also using overloaded "=" operator)  
my::vector<int> vect2(vect1);  
vect2.at(1) = 3;  
vect2.at(2) = 5;  
std::cout << vect2;

Output:

[1, 0, 0, 0, 0]  
[1, 3, 5, 0, 0]

### vector(const std::initializer\_list<element\_type> init\_list)

使用initializer\_list构造一个vector，initializer\_list为在C++11中引入的概念，该概念允许使用初始化列表的方式初始化一个对象，拓展了C中只能使用初始化列表初始化一个固定内存区域的语法。

Example Code:

my::vector<int> vec1{1, 2, 3, 4, 5, 6, 7, 8};  
std::cout << vec1;

Output:

[1, 2, 3, 4, 5, 6, 7, 8]

### vector(const\_iterator first, const\_iterator last)

通过vector的两个iterator，进行初始化，初始化的vector包含两个迭代器之间包含的内容，不允许first > last。

Example Code:

my::vector<int> vec1{1, 2, 3, 4, 5, 6, 7, 8};  
my::vector<int> vec2(vec1.begin(), vec1.end() - 2);  
std::cout << vec1 << vec2;

Output:

[1, 2, 3, 4, 5, 6, 7, 8]  
[1, 2, 3, 4, 5, 6]

## 析构函数

### ~vector()

该析构函数不需要直接使用。其作用是在析构时，释放掉当前vector所分配的内存。

## 获取整体状态

获取当前vector的整体状态。

### bool empty() const

获取当前vector是否为空（长度/大小是否为0）。

Example Code:

my::vector<int> vect1;  
std::cout << vect1.empty() << std::endl;  
my::vector<int> vect2{1, 2, 3, 4, 5};  
std::cout << vect2.empty() << std::endl;

Output:

1  
0

### size\_type size() const/size\_type length() const

取得当前vector的大小/长度

注意：此方法的复杂度为O(1)。

Example Code:

my::vector<int> vect1;  
std::cout << vect1.size() << std::endl;  
std::cout << vect1.length() << std::endl;  
  
my::vector<int> vect2(15);  
std::cout << vect2.size() << std::endl;  
std::cout << vect2.length() << std::endl;

Output:

0  
0  
15  
15

### size\_type capacity() const

获取当前vector在内存中所占的实际容量。

Example Code:

my::vector<int> vect1;  
std::cout << vect1.capacity() << std::endl;  
  
my::vector<int> vect2(15);  
std::cout << vect2.capacity() << std::endl;

Output:

2  
16

## 获取特定元素（的引用）

### element\_reference operator[](const int index)

按照地址偏移量，给出特定位置元素的引用。（为提高性能，没有范围检查）

Example Code：

my::vector<int> vect{0, 1, 2, 3, 4, 5};  
std::cout << vect[2] << std::endl;  
// wrong, but no rumtime error  
// do not do this, has potential risk  
std::cout << vect[-1] << std::endl;

Output:

2  
0 (maybe...)

### element\_reference at(const int index)

按照地址偏移量，给出特定位置元素的引用。

Example Code:

my::vector<int> vect{0, 1, 2, 3, 4, 5};  
std::cout << vect.at(2) << std::endl;  
// has runtime error  
// std::cout << vect.at(-1) << std::endl;

Output:

2

### element\_reference front()/const\_element\_reference front() const

给出该vector第一个元素的引用。

Example Code:

my::vector<int> vect{0, 1, 2, 3, 4, 5};  
std::cout << vect.front() << std::endl;

Output:

0

### element\_reference back()/const\_element\_reference back() const

给出该vector最后一个元素的引用。

Example Code:

my::vector<int> vect{0, 1, 2, 3, 4, 5};  
std::cout << vect.back() << std::endl;

Output:

5

### iterator begin() const

给出第一个元素的迭代器。

Example Code:

my::vector<int> vect{0, 1, 2, 3, 4, 5};  
std::cout << \*vect.begin() << std::endl;

Output:

0

### iterator end() const

给出一个迭代器，指向最后一个元素之后的一个位置。

Example Code:

my::vector<int> vect{0, 1, 2, 3, 4, 5};  
std::cout << \*(vect.end() - 1) << std::endl;

Output:

5

## 比较运算符

### bool operator==(const self\_type &rhs) const/bool operator!=(const self\_type &rhs) const

比较两个vector在长度和每个元素上是否都相等。

Example Code:

my::vector<int> vect1{0, 1, 2, 3, 4, 5};  
my::vector<int> vect2{0, 1, 2, 3, 4};  
my::vector<int> vect3{0, 1, 2, 3, 4, 5};  
std::cout << (vect1 == vect2) << " " << (vect1 == vect3) << std::endl;  
std::cout << (vect1 != vect2) << " " << (vect1 != vect3) << std::endl;

Output:

0 1  
1 0

## Range-Based Loop

使用C++11提供的新语法，进行for循环。

具体实现方式可以参考[链接](https://stackoverflow.com/questions/8164567/how-to-make-my-custom-type-to-work-with-range-based-for-loops)，简单来说，需要在容器内实现一个iterator，支持解引用，自增，自减，以及比较运算符，在容器内实现begin()和end()类方法，返回首个元素的iterator以及末尾的iterator。

Example Code:

my::vector<int> vect1{0, 1, 2, 3, 4, 5};  
for (const auto it : vect1) {  
 std::cout << it << " ";  
}

Output:

0 1 2 3 4 5

## 改变元素/状态

### self\_type &resize(const int size)

重置当前vector到给定的大小。

使用了在g++和MSVC等编译器在实现vector容器中引入的[vector增长因子](https://www.zhihu.com/question/36538542)。在这个类的实现中，增长因子为2，即当vector空间已满时，会重新分配大小为原vector的2倍大小的空间。这也是allocator的基本实现。

* 如果给定的大小小于当前的size()，那么将会忽略给定大小之后的元素。
* 如果给定的大小大于或等于当前的size()但是小于当前的capacity()，则会在不重新申请新的空间下，将vector末尾的指针移到指定的大小处。
* 如果给定的大小大于当前的capacity()，那么会重新申请并分配大小为当前capacity()两倍的空间，并将原来的内容按顺序拷贝到新的空间中。

Example Code:

my::vector<int> vect1{0, 1, 2, 3, 4, 5};  
std::cout << vect1;  
std::cout << vect1.size() << " " << vect1.capacity() << std::endl;  
vect1.resize(15);  
std::cout << vect1;  
std::cout << vect1.size() << " " << vect1.capacity() << std::endl;  
vect1.resize(4);  
std::cout << vect1;  
std::cout << vect1.size() << " " << vect1.capacity() << std::endl;

Output:

[0, 1, 2, 3, 4, 5]  
6 8  
[0, 1, 2, 3, 4, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
15 16  
[0, 1, 2, 3]  
4 16

## 其他

* self\_type &push\_front(element\_right\_reference argument)/self\_type &push\_front(const\_element\_reference argument)
* self\_type &push\_back(element\_right\_reference argument)/self\_type &push\_back(const\_element\_reference argument)
* self\_type &pop\_front()
* self\_type &pop\_back()
* element\_type pop\_back(element\_reference out\_param)
* self\_type &insert(const\_iterator position, const\_element\_reference val)
* self\_type &insert(const\_iterator position, size\_type num, const\_element\_reference val)
* self\_type &insert(const\_iterator position, InputIterator first, InputIterator last)
* self\_type &insert(const\_iterator position, element\_right\_reference val)
* self\_type &insert(const\_iterator position, std::initializer\_list<element\_type> object)
* self\_type &erase(const\_iterator position)
* self\_type &erase(const\_iterator first, const\_iterator end)
* self\_type &clear()
* self\_type &reverse()
* self\_type slice(const\_iterator begin, const\_iterator end)
* void swap(self\_type &object)
* self\_type &operator=(const self\_type &object)
* self\_type &operator=(self\_type &&object)
* self\_type &operator=(const std::initializer\_list<element\_type> init\_list)

Example Code:

my::vector<int> vect1{0, 1, 2, 3, 4, 5};  
my::vector<int> vect2{1, 2, 3};  
vect1.insert(vect1.begin() + 3, vect2.begin(), vect2.end());  
std::cout << vect1;  
vect1.insert(vect1.begin(), static\_cast<size\_t>(3), 3);  
std::cout << vect1;  
vect1.insert(vect1.begin() + 2, {2, 0, 4, 2, 5, 4});  
std::cout << vect1;  
vect1.insert(vect1.end(), 1145);  
std::cout << vect1;  
vect1.erase(vect1.begin() + 3);  
std::cout << vect1;  
vect1.erase(vect1.begin() + 2, vect1.end() - 7);  
std::cout << vect1;  
vect1.push\_back(3).push\_front(6).push\_back(9);  
std::cout << vect1;  
vect1.reverse();  
std::cout << vect1;

Output:

[0, 1, 2, 1, 2, 3, 3, 4, 5]  
[3, 3, 3, 0, 1, 2, 1, 2, 3, 3, 4, 5]  
[3, 3, 2, 0, 4, 2, 5, 4, 3, 0, 1, 2, 1, 2, 3, 3, 4, 5]  
[3, 3, 2, 0, 4, 2, 5, 4, 3, 0, 1, 2, 1, 2, 3, 3, 4, 5, 1145]  
[3, 3, 2, 4, 2, 5, 4, 3, 0, 1, 2, 1, 2, 3, 3, 4, 5, 1145]  
[3, 3, 1, 2, 3, 3, 4, 5, 1145]  
[6, 3, 3, 1, 2, 3, 3, 4, 5, 1145, 3, 9]  
[9, 3, 1145, 5, 4, 3, 3, 2, 1, 3, 3, 6]

更多的示例代码，可以参见test.cpp。

## 迭代器

实现方法与C++ STL的方法类似，用法基本相同，具体用法参见[C++参考手册](https://zh.cppreference.com/mwiki/index.php?title=首页&variant=zh-cn)。

## 附录

### 类型重定义

Code, written in header:

// header files...  
  
template <typename T>  
class vector{  
// ......  
// ......  
  
// in class vector<T>:  
/\* Type definitions \*/  
 typedef T element\_type;  
 typedef T \*element\_pointer;  
 typedef T &element\_reference;  
 typedef T &&element\_right\_reference;  
 typedef const T \*const\_element\_pointer;  
 typedef const T &const\_element\_reference;  
 typedef size\_t size\_type;  
 typedef vector<T> self\_type;  
   
// ......  
// ......  
  
 class const\_iterator{  
 // in class vector<T>::const\_iterator:  
 public:  
 /\* Type definitions \*/  
 typedef const\_iterator self\_type;  
  
 // ......  
 // ......  
 };  
  
 class iterator: public const\_iterator{  
 // in class vector<T>::iterator:  
 public:  
 /\* Type definitions \*/  
 typedef iterator self\_type;  
  
 // ......  
 // ......  
 }  
   
};

### 有关使用ostream进行输出vector的运算符重载

此处借鉴了Python对list进行输出的格式。

Example Code:

// output the vector using operator "<<"  
template <typename T>  
std::ostream &operator<<(std::ostream &os, const my::vector<T> &vect) {  
 if (vect.empty()) {  
 os << "[]" << std::endl;  
 return os;  
 }  
  
 os << '[';  
 for (typename my::vector<T>::iterator it = vect.begin(); it != vect.end() - 1;  
 it++) {  
 os << \*it << ", ";  
 }  
 os << vect.back() << ']' << std::endl;  
  
 return os;  
}