

[https://blog.csdn.net/qq\\_36534818/article/details/76559835](https://blog.csdn.net/qq_36534818/article/details/76559835)

<https://www.cnblogs.com/zhonghuasong/p/5975979.html>

**vector**：序列容器；任意元素的读取、修改具有常数时间复杂度，在序列尾部进行插入、删除是常数时间复杂度，但在序列的头部插入、删除的时间复杂度是 $O(n)$ ，可以在任何位置插入新元素，有随机访问功能，插入删除操作需要考虑；vector在大量添加元素的时候问题最大，因为他的一种最常见的内存分配实现方法是当前的容量(capacity)不足就申请一块当前容量2倍的新内存空间，然后将所有的老元素全部拷贝到新内存中，添加大量元素的时候的花费的惊人的大。如果由于其他因素必须使用vector，并且还需要大量添加新元素，那么可以使用成员函数reserve来事先分配内存，这样可以减少很多不必要的消耗。

1、vector是表示可变大数组的序列容器；

2、就像数组一样，vector也采用的连续存储空间来存储元素。也就是意味着可以采用下标对vector的元素进行访问，和数组一样高效。但是又不像数组，它的大小是可以动态改变的，而且它的大小会被容器自动处理。

3、本质讲，vector使用动态分配数组来存储它的元素。当新元素插入时候，这个数组需要被重新分配大小为了增加存储空间。其做法是，分配一个新的数组，然后将全部元素移到这个数组。就时间而言，这是一个相对代价高的任务；当一个新的元素加入到容器的时候，vector并不会每次都重新分配大小，只有当大小不够用时，才一次性分配足够的大小；

4、vector分配空间策略：vector会分配一些额外的空间以适应可能的增长，因为存储空间比实际需要的存储空间更大。不同的库采用不同的策略权衡空间的使用和重新分配。但是无论如何，重新分配都应该是对数增长的间隔大小，以至于在末尾插入一个元素的时候是在常数时间的复杂度完成的。

5、因此，vector占用了更多的存储空间，为了获得管理存储空间的能力，并且以一种有效的方式动态增长。

6、与其它动态序列容器相比(deques, lists and forward\_lists)，vector在访问元素的时候更加高效，在末尾添加和删除元素相对高效。对于其它不在末尾的删除和插入操作，效率更低。比起lists和forward\_lists统一的迭代器和引用更好。

**特性：**

**有序**：序列容器中的元素以严格的线性顺序排列。单个元素按其顺序通过其位置访问；

**动态数组**：允许直接访问序列中的任何元素，甚至通过指针算术，并在序列结尾提供相对快速的添加/删除元素；

**自动存储**：容器使用allocator对象动态处理其存储需求；

size = \_last - \_first; //元素个数

capacity = \_end - \_first; //容量

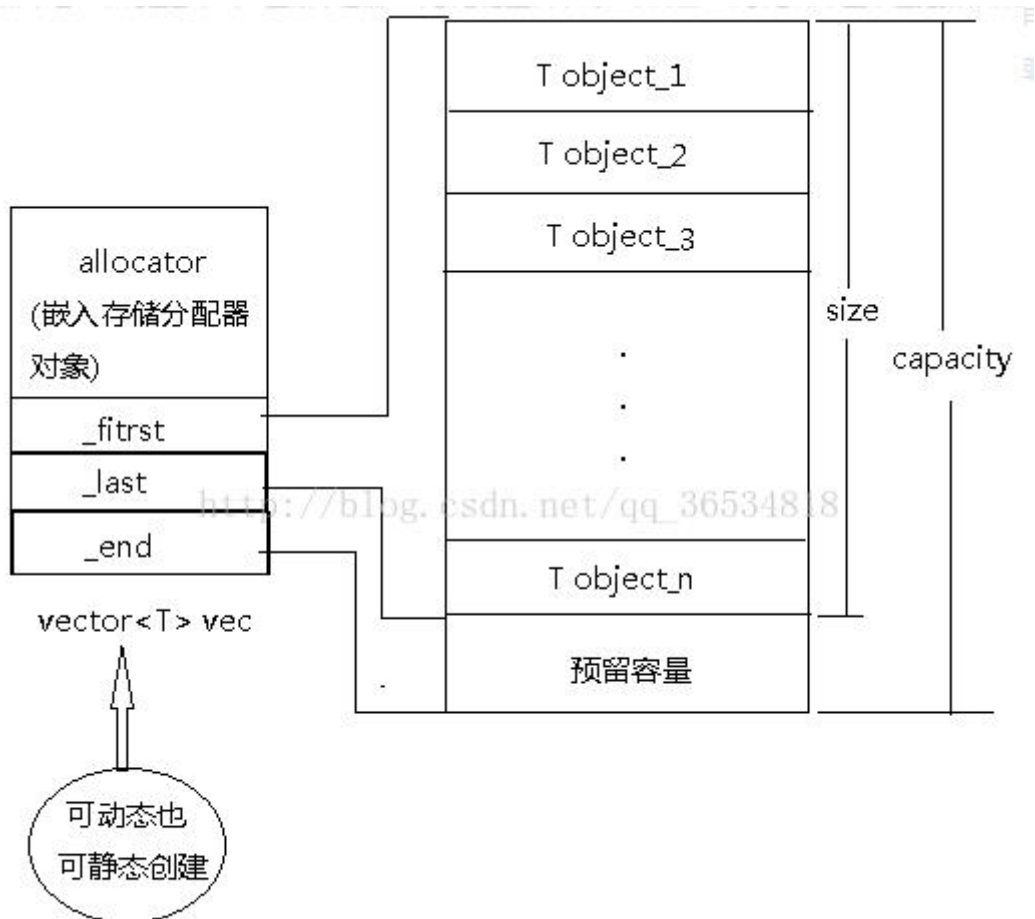
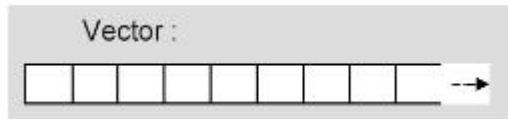


图1 vector容器对象及其元素的内存映像



头文件:

```
#include<vector>
```

基本操作:

```
vec.size();    //vector中现有元素的数量;
```

```
vec.capacity();    //vector现有内存大小可以存储的元素数量;
```

```
vec.max_size();    //vector中最多能够存储的元素数量; 容器可以无限扩展, 这个值可以很大;
```

```
vec.size() <= vec.capacity() <<vec.max_size();
```

//改变vector的size, 如果n小于vec当前的size, 则容器中只保留前n个元素, 此时该vector的capacity不变; 如果n大于vec当前的size, 在当前容器的尾部追加元素, 直到n个元素; 如果n也大于该vec当前的capacity, 则也需要扩展该vec的capacity直到满足存储n个元素的要求;

```
vec.resize(size_type n, value_type val = value_type());
```

//改变vector的capacity, 要求vector的capacity至少能够存储n个元素, 如果n大于vector当前的capacity, 就扩展vector的内存, 使得该vector能够存储至少n个元素; 如果n小于vector当前的capacity, 则该vector啥也不做;

```
vec.reserve(size_type n);
```

//返回的均为引用, 可以通过引用修改该元素的值

```
vec.at(size_type n); //返回容器中在位置n上的元素
```

```
vec.front(); //返回容器中的第一个元素
```

```
vec.back(); //返回容器中最后一个元素
```

```
vec.push_back(); //向容器的尾部添加一个元素, 导致size加1, 可能导致重新申请内存;
```

```
vec.pop_back(); //删除容器中最后一个元素, 改变size, 不会改变capacity;
```

```
vec.clear(); //删除容器中的所有元素, 改变size, 不保证会改变capacity;
```

//可以通过迭代器修改指向元素的值: \*it = xxx;

```
vec.begin(); //返回指向容器首元素的迭代器
```

```
vec.end(); //返回指向容器尾元素下一个位置的迭代器
```

//删除容器中元素, vector使用数组作为底层存储, 删除中间元素的时候, 就需要将删除点后面的元素全部挪动一次, 导致指向该位置以及该位置后面的迭代器、指针和引用全部失效; 返回值为一个迭代器, 该迭代器指向删除元素(position)后面的一个元素重新布局后的位置;

所以vector可以通过erase的返回值安全的迭代删除所有元素;

```
vec.erase(iterator position);
```

```
vec.erase(iterator first, iterator last);
```

//use iterator to delete all element safely

```
for (vector<int>::iterator iter = vec.begin(); iter != vec.end();){  
    iter = vec.erase(iter);  
}
```

//插入元素到指定位置的前面; 如果插入元素的数量超过vector剩余容量, 就需要重新分配内存; 返回值为迭代器, 该迭代器指向新插入的第一个元素;

```
vec.insert(iterator position, const value_type& val);
```

```
vec.insert(iterator position, size_type n, const value_type& val);
```

```
vec.insert(iterator position, InputIterator first, InputIterator last);
```

### vector迭代器iterator有效性问题:

在重新分配内存时, 迭代器失效; 当把超过capacity() - size()个元素插入vector中时, 内存会重新分配, 所有的迭代器都将失效; 否则, 指向当前元素以后的任何元素的迭代器都将失效。当删除元素时, 指向被删除元素以后的任何元素的迭代器都将失效;

In case the container shrinks, all iterators, pointers and references to elements that have not been removed remain valid after the resize and refer to the same elements they were referring to before the call. If the container expands, the [end iterator](#) is invalidated and, if it has to reallocate storage, all iterators, pointers and references related to this container are also invalidated.

resize();

If a reallocation happens, all iterators, pointers and references related to the container are invalidated.

Otherwise, they all keep referring to the same elements they were referring to before the call.

reserve();

If a reallocation happens, all iterators, pointers and references related to the container are invalidated.

Otherwise, only the [end iterator](#) is invalidated, and all iterators, pointers and references to elements are guaranteed to keep referring to the same elements they were referring to before the call.

push\_back();

The [end iterator](#) and any iterator, pointer and reference referring to the removed element are invalidated; Iterators, pointers and references referring to other elements that have not been removed are guaranteed to keep referring to the same elements they were referring to before the call.

pop\_back();

Iterators, pointers and references pointing to *position* (or *first*) and beyond are invalidated, with all iterators, pointers and references to elements before *position* (or *first*) are guaranteed to keep referring to the same elements they were referring to before the call.

erase();

All iterators, pointers and references related to this container are invalidated.

clear();

If a reallocation happens, all iterators, pointers and references related to the container are invalidated.

Otherwise, only those pointing to *position* and beyond are invalidated, with all iterators, pointers and references to elements before *position* guaranteed to keep referring to the same elements they were referring to before the call

`insert();`