

Lesson07---vector

【本节目标】

- 1.vector的介绍及使用
- 2.vector深度剖析及模拟实现

1.vector的介绍及使用

1.1 vector的介绍

[vector的文档介绍](#)

1. vector是表示可变大小数组的序列容器。
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来增加存储空间。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，因为每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。
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（deque, 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。

学习方法：使用STL的三个境界：能用，明理，能扩展，那么下面学习vector，我们也是按照这个方法去学习

1.2 vector的使用

vector学习时一定要学会查看文档：[vector的文档介绍](#)，vector在实际中非常的重要，在实际中我们熟悉常见的接口就可以，下面列出了哪些接口是要重点掌握的。

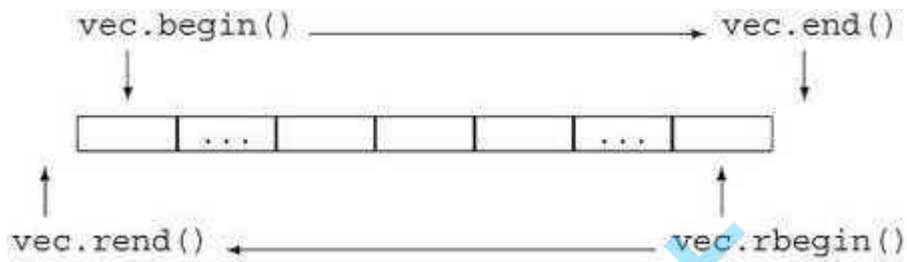
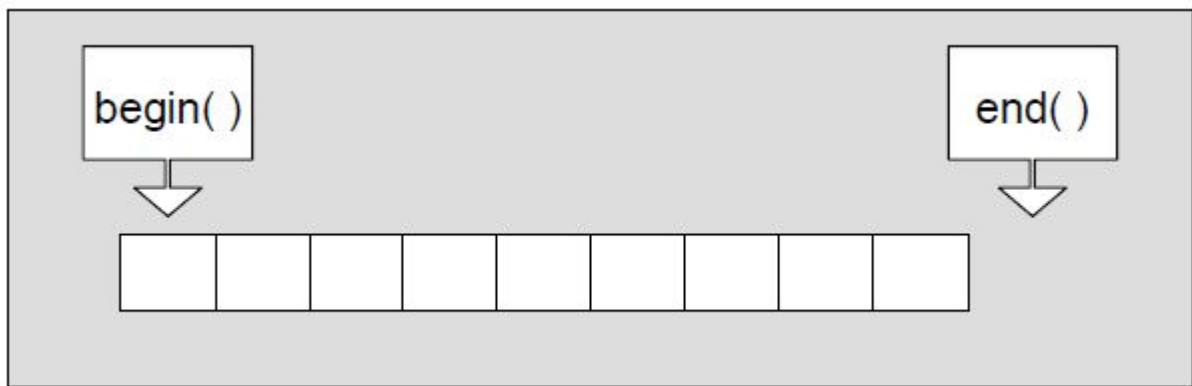
1.2.1 vector的定义

(constructor)构造函数声明	接口说明
vector() (重点)	无参构造
vector (size_type n, const value_type& val = value_type())	构造并初始化n个val
vector (const vector& x); (重点)	拷贝构造
vector (InputIterator first, InputIterator last);	使用迭代器进行初始化构造

```
1 // constructing vectors
2 #include <iostream>
3 #include <vector>
4
5 int main ()
6 {
7     // constructors used in the same order as described above:
8     std::vector<int> first; // empty vector of ints
9     std::vector<int> second (4,100); // four ints with value 100
10    std::vector<int> third (second.begin(),second.end()); // iterating through second
11    std::vector<int> fourth (third); // a copy of third
12
13    // 下面涉及迭代器初始化的部分，我们学习完迭代器再来看这部分
14    // the iterator constructor can also be used to construct from arrays:
15    int myints[] = {16,2,77,29};
16    std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
17
18    std::cout << "The contents of fifth are:";
19    for (std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    return 0;
24 }
```

1.2.2 vector iterator 的使用

iterator的使用	接口说明
begin + end (重点)	获取第一个数据位置的iterator/const_iterator， 获取最后一个数据的下一个位置的iterator/const_iterator
rbegin + rend	获取最后一个数据位置的reverse_iterator， 获取第一个数据前一个位置的reverse_iterator



```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  void PrintVector(const vector<int>& v)
6  {
7      // const对象使用const迭代器进行遍历打印
8      vector<int>::const_iterator it = v.begin();
9      while (it != v.end())
10     {
11         cout << *it << " ";
12         ++it;
13     }
14     cout << endl;
15 }
16
17 int main()
18 {
19     // 使用push_back插入4个数据
20     vector<int> v;
21     v.push_back(1);
22     v.push_back(2);
23     v.push_back(3);
24     v.push_back(4);
25
26     // 使用迭代器进行遍历打印
27     vector<int>::iterator it = v.begin();
28     while (it != v.end())
29     {
30         cout << *it << " ";
31         ++it;
32     }
```

```

33     cout << endl;
34
35     // 使用迭代器进行修改
36     it = v.begin();
37     while (it != v.end())
38     {
39         *it *= 2;
40         ++it;
41     }
42
43     // 使用反向迭代器进行遍历再打印
44     vector<int>::reverse_iterator rit = v.rbegin();
45     while (rit != v.rend())
46     {
47         cout << *rit << " ";
48         ++rit;
49     }
50     cout << endl;
51
52     PrintVector(v);
53
54     return 0;
55 }

```

1.2.3 vector 空间增长问题

容量空间	接口说明
<u>size</u>	获取数据个数
<u>capacity</u>	获取容量大小
<u>empty</u>	判断是否为空
<u>resize</u> (重点)	改变vector的size
<u>reserve</u> (重点)	改变vector放入capacity

- capacity的代码在vs和g++下分别运行会发现，**vs下capacity是按1.5倍增长的，g++是按2倍增长的。**这个问题经常会考察，不要固化的认为，顺序表增容都是2倍，具体增长多少是根据具体的需求定义的。vs是PJ版本STL，g++是SGI版本STL。
- reserve只负责开辟空间，如果确定知道需要用多少空间，reserve可以缓解vector增容的代价缺陷问题。
- resize在开空间的同时还会进行初始化，影响size。

```

1  // vector::capacity
2  #include <iostream>
3  #include <vector>
4
5  int main ()
6  {

```

```

7   size_t sz;
8   std::vector<int> foo;
9   sz = foo.capacity();
10  std::cout << "making foo grow:\n";
11  for (int i=0; i<100; ++i) {
12      foo.push_back(i);
13      if (sz!=foo.capacity()) {
14          sz = foo.capacity();
15          std::cout << "capacity changed: " << sz << '\n';
16      }
17  }
18 }

```

19
20 vs: 运行结果:

```

21 making foo grow:
22 capacity changed: 1
23 capacity changed: 2
24 capacity changed: 3
25 capacity changed: 4
26 capacity changed: 6
27 capacity changed: 9
28 capacity changed: 13
29 capacity changed: 19
30 capacity changed: 28
31 capacity changed: 42
32 capacity changed: 63
33 capacity changed: 94
34 capacity changed: 141

```

35
36 g++运行结果:

```

37 making foo grow:
38 capacity changed: 1
39 capacity changed: 2
40 capacity changed: 4
41 capacity changed: 8
42 capacity changed: 16
43 capacity changed: 32
44 capacity changed: 64
45 capacity changed: 128

```

```

1  // vector::reserve
2  #include <iostream>
3  #include <vector>
4
5  int main ()
6  {
7      size_t sz;
8      std::vector<int> foo;
9      sz = foo.capacity();
10     std::cout << "making foo grow:\n";
11     for (int i=0; i<100; ++i) {
12         foo.push_back(i);

```

```

13     if (sz!=foo.capacity()) {
14         sz = foo.capacity();
15         std::cout << "capacity changed: " << sz << '\n';
16     }
17 }
18
19 std::vector<int> bar;
20 sz = bar.capacity();
21 bar.reserve(100); // this is the only difference with foo above
22 std::cout << "making bar grow:\n";
23 for (int i=0; i<100; ++i) {
24     bar.push_back(i);
25     if (sz!=bar.capacity()) {
26         sz = bar.capacity();
27         std::cout << "capacity changed: " << sz << '\n';
28     }
29 }
30 return 0;
31 }

```

```

1 // vector::resize
2 #include <iostream>
3 #include <vector>
4
5 int main ()
6 {
7     std::vector<int> myvector;
8
9     // set some initial content:
10    for (int i=1;i<10;i++)
11        myvector.push_back(i);
12
13    myvector.resize(5);
14    myvector.resize(8,100);
15    myvector.resize(12);
16
17    std::cout << "myvector contains:";
18    for (int i=0;i<myvector.size();i++)
19        std::cout << ' ' << myvector[i];
20    std::cout << '\n';
21
22    return 0;
23 }

```

1.2.3 vector 增删查改

vector增删查改	接口说明
push back (重点)	尾插
pop back (重点)	尾删
find	查找。(注意这个是算法模块实现, 不是vector的成员接口)
insert	在position之前插入val
erase	删除position位置的数据
swap	交换两个vector的数据空间
operator[] (重点)	像数组一样访问

```

1  // push_back/pop_back
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  int main()
7  {
8      int a[] = { 1, 2, 3, 4 };
9      vector<int> v(a, a+sizeof(a)/sizeof(int));
10
11     vector<int>::iterator it = v.begin();
12     while (it != v.end()) {
13         cout << *it << " ";
14         ++it;
15     }
16     cout << endl;
17
18     v.pop_back();
19     v.pop_back();
20
21     it = v.begin();
22     while (it != v.end()) {
23         cout << *it << " ";
24         ++it;
25     }
26     cout << endl;
27
28     return 0;
29 }

```

```

1  // find / insert / erase
2  #include <iostream>
3  #include <algorithm>
4  #include <vector>

```

```

5 using namespace std;
6
7 int main()
8 {
9     int a[] = { 1, 2, 3, 4 };
10    vector<int> v(a, a + sizeof(a) / sizeof(int));
11
12    // 使用find查找3所在位置的iterator
13    vector<int>::iterator pos = find(v.begin(), v.end(), 3);
14
15    // 在pos位置之前插入30
16    v.insert(pos, 30);
17
18    vector<int>::iterator it = v.begin();
19    while (it != v.end()) {
20        cout << *it << " ";
21        ++it;
22    }
23    cout << endl;
24
25    pos = find(v.begin(), v.end(), 3);
26    // 删除pos位置的数据
27    v.erase(pos);
28
29    it = v.begin();
30    while (it != v.end()) {
31        cout << *it << " ";
32        ++it;
33    }
34    cout << endl;
35
36    return 0;
37 }

```

```

1 // operator[]+index 和 C++11中vector的新式for+auto的遍历
2 // vector使用这两种遍历方式是比较便捷的。
3 #include <iostream>
4 #include <vector>
5 using namespace std;
6
7 int main()
8 {
9     int a[] = { 1, 2, 3, 4 };
10    vector<int> v(a, a + sizeof(a) / sizeof(int));
11
12    // 通过[]读写第0个位置。
13    v[0] = 10;
14    cout << v[0] << endl;
15
16    // 通过[i]的方式遍历vector
17    for (size_t i = 0; i < v.size(); ++i)
18        cout << v[i] << " ";

```



```

19     cout << endl;
20
21     vector<int> swapv;
22     swapv.swap(v);
23
24     cout << "v data: ";
25     for (size_t i = 0; i < v.size(); ++i)
26         cout << v[i] << " ";
27     cout << endl;
28
29     cout << "swapv data: ";
30     for (size_t i = 0; i < swapv.size(); ++i)
31         cout << swapv[i] << " ";
32     cout << endl;
33
34     // C++11支持的新式范围for遍历
35     for(auto x : v)
36         cout << x << " ";
37     cout << endl;
38
39     return 0;
40 }

```

1.2.4 vector 迭代器失效问题。（重点）

迭代器的主要作用就是让算法能够不用关心底层数据结构，其底层实际就是一个指针，或者是对指针进行了封装，比如：vector的迭代器就是原生态指针T*。因此迭代器失效，实际就是迭代器底层对应指针所指向的空间被销毁了，而使用一块已经被释放的空间，造成的后果是程序崩溃(即如果继续使用已经失效的迭代器，程序可能会崩溃)。

对于vector可能会导致其迭代器失效的操作有：

1. 会引起其底层空间改变的操作，都有可能是迭代器失效，比如：resize、reserve、insert、assign、push_back等。

```

1  #include <iostream>
2  using namespace std;
3  #include <vector>
4
5  int main()
6  {
7      vector<int> v{1,2,3,4,5,6};
8
9      auto it = v.begin();
10
11     // 将有效元素个数增加到100个，多出的位置使用8填充，操作期间底层会扩容
12     // v.resize(100, 8);
13
14     // reserve的作用就是改变扩容大小但不改变有效元素个数，操作期间可能会引起底层容量改变
15     // v.reserve(100);
16
17     // 插入元素期间，可能会引起扩容，而导致原空间被释放
18     // v.insert(v.begin(), 0);

```

```

19 // v.push_back(8);
20
21 // 给vector重新赋值，可能会引起底层容量改变
22 v.assign(100, 8);
23
24 /*
25 出错原因：以上操作，都有可能会导致vector扩容，也就是说vector底层原理旧空间被释放掉，
而在打印时，it还使用的是释放之间的旧空间，在对it迭代器操作时，实际操作的是一块已经被释放的
空间，而引起代码运行时崩溃。
26 解决方式：在以上操作完成之后，如果想要继续通过迭代器操作vector中的元素，只需给it重新
赋值即可。
27 */
28 while(it != v.end())
29 {
30     cout<< *it << " ";
31     ++it;
32 }
33 cout<<endl;
34 return 0;
35 }

```

2. 指定位置元素的删除操作--erase

```

1  #include <iostream>
2  using namespace std;
3  #include <vector>
4
5  int main()
6  {
7      int a[] = { 1, 2, 3, 4 };
8      vector<int> v(a, a + sizeof(a) / sizeof(int));
9
10     // 使用find查找3所在位置的iterator
11     vector<int>::iterator pos = find(v.begin(), v.end(), 3);
12
13     // 删除pos位置的数据，导致pos迭代器失效。
14     v.erase(pos);
15     cout << *pos << endl; // 此处会导致非法访问
16     return 0;
17 }

```

erase删除pos位置元素后，pos位置之后的元素会往前搬移，没有导致底层空间的改变，理论上讲迭代器不应该会失效，但是：如果pos刚好是最后一个元素，删完之后pos刚好是end的位置，而end位置是没有元素的，那么pos就失效了。因此删除vector中任意位置上元素时，vs就认为该位置迭代器失效了。

以下代码的功能是删除vector中所有的偶数，请问那个代码是正确的，为什么？

```

1  #include <iostream>
2  using namespace std;
3  #include <vector>

```

```

4
5 int main()
6 {
7     vector<int> v{ 1, 2, 3, 4 };
8     auto it = v.begin();
9     while (it != v.end())
10    {
11        if (*it % 2 == 0)
12            v.erase(it);
13
14        ++it;
15    }
16
17    return 0;
18 }
19
20 int main()
21 {
22     vector<int> v{ 1, 2, 3, 4 };
23     auto it = v.begin();
24     while (it != v.end())
25    {
26        if (*it % 2 == 0)
27            it = v.erase(it);
28        else
29            ++it;
30    }
31
32    return 0;
33 }

```

迭代器失效解决办法：在使用前，对迭代器重新赋值即可。

1.2.5 vector 在OJ中的使用。

1. [只出现一次的数字](#)

```

1 class Solution {
2 public:
3     int singleNumber(vector<int>& nums) {
4         int value = 0;
5         for(auto e : v) {value ^= e; }
6         return value;
7     }
8 };

```

2. [杨辉三角](#)

```

1 // 涉及resize / operator[]
2 class Solution {

```

```

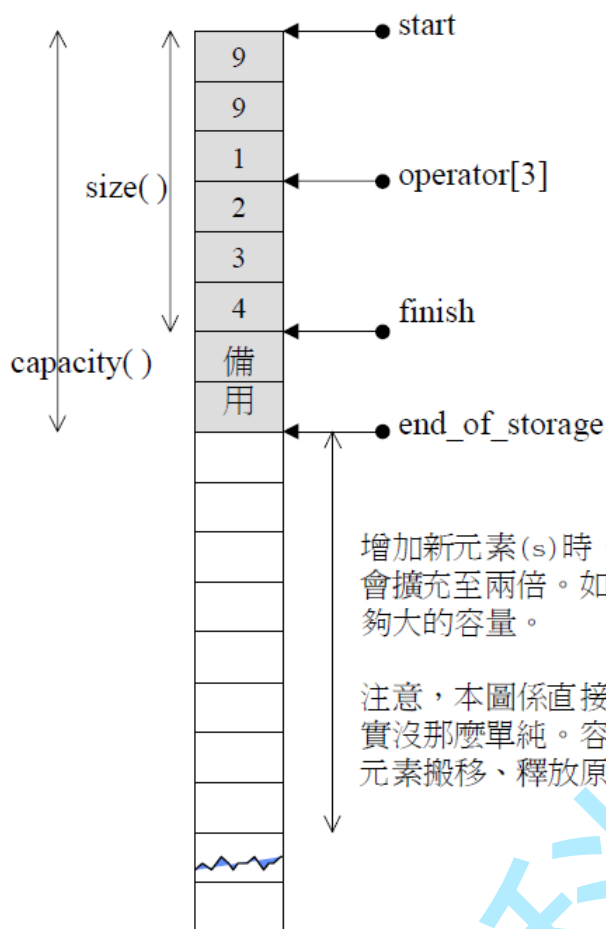
3 public:
4     // 核心思想: 找出杨辉三角的规律, 发现每一行头尾都是1, 中间第[j]个数等于上一行[j-1]+[j]
5     vector<vector<int>> generate(int numRows) {
6         vector<vector<int>> vv;
7         // 先开辟杨辉三角的空间
8         vv.resize(numRows);
9         for(size_t i = 1; i <= numRows; ++i)
10        {
11            vv[i-1].resize(i, 0);
12
13            // 每一行的第一个和最后一个都是1
14            vv[i-1][0] = 1;
15            vv[i-1][i-1] = 1;
16        }
17
18        for(size_t i = 0; i < vv.size(); ++i)
19        {
20            for(size_t j = 0; j < vv[i].size(); ++j)
21            {
22                if(vv[i][j] == 0)
23                {
24                    vv[i][j] = vv[i-1][j-1] + vv[i-1][j];
25                }
26            }
27        }
28
29        return vv;
30    }
31 };

```

总结: 通过上面的练习我们发现vector常用的接口更多是插入和遍历。遍历更喜欢用数组operator[i]的形式访问, 因为这样便捷。课下除了再自己实现一遍上面课堂讲解的OJ练习, 请自行完成下面题目的OJ练习。以此增强学习vector的使用。

3. [删除排序数组中的重复项 OI](#)
4. [只出现一次的数ii OI](#)
5. [只出现一次的数iii OI](#)
6. [数组中出现次数超过一半的数字 OI](#)
7. [电话号码字母组合OI](#)
8. [连续子数组的最大和 OI](#)

2.vector深度剖析及模拟实现



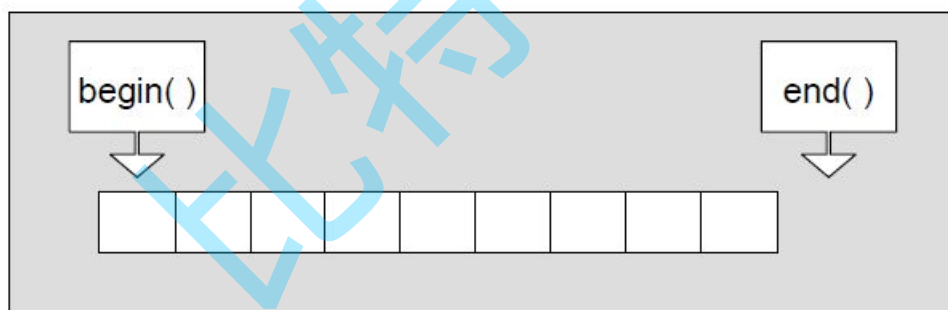
經過以下動作：

```
vector<int> iv(2, 9);
iv.push_back(1);
iv.push_back(2);
iv.push_back(3);
iv.push_back(4);
```

vector 記憶體及各成員呈現左圖狀態

增加新元素(s)時，如果超過當時的容量，則容量會擴充至兩倍。如果兩倍容量仍不足，就擴張至足夠大的容量。

注意，本圖係直接在原空間之後畫上新增空間，其實沒那麼單純。容量的擴張必須經歷「重新配置、元素搬移、釋放原空間」等過程，工程浩大。



2.1 std::vector的核心框架接口的模拟实现bit::vector

```
1  #include <iostream>
2  using namespace std;
3  #include <assert.h>
4
5  // 注意这里namespace大家下去就不要取名为bit了，否则出去容易翻车。^^
6  namespace bit
7  {
8      template<class T>
9      class vector
10     {
11     public:
12         // Vector的迭代器是一个原生指针
13         typedef T* iterator;
```

```

14 typedef const T* const_iterator;
15
16 iterator begin() { return _start; }
17 iterator end() { return _finish; }
18
19 const_iterator cbegin() const { return _start; }
20 const_iterator cend() const { return _finish; }
21
22 // construct and destroy
23 vector()
24     : _start(nullptr)
25     , _finish(nullptr)
26     , _endOfStorage(nullptr)
27 {}
28
29 vector(int n, const T& value = T())
30     : _start(nullptr)
31     , _finish(nullptr)
32     , _endOfStorage(nullptr)
33 {
34     reserve(n);
35     while (n--)
36     {
37         push_back(value);
38     }
39 }
40
41 // 如果使用iterator做迭代器, 会导致初始化的迭代器区间[first,last)只能是vector的迭代器
42 // 重新声明迭代器, 迭代器区间[first,last]可以是任意容器的迭代器
43 template<class InputIterator>
44 vector(InputIterator first, InputIterator last)
45 {
46     reserve(last - first);
47     while (first != last)
48     {
49         push_back(*first);
50         ++first;
51     }
52 }
53
54 vector(const vector<T>& v)
55     : _start(nullptr)
56     , _finish(nullptr)
57     , _endOfStorage(nullptr)
58 {
59     reserve(v.capacity());
60     iterator it = begin();
61     const_iterator vit = v.cbegin();
62     while (vit != v.cend())
63     {
64         *it++ = *vit++;
65     }
66 }

```

```

67
68 vector<T>& operator=(vector<T> v)
69 {
70     swap(v);
71     return *this;
72 }
73
74 ~vector()
75 {
76     delete[] _start;
77     _start = _finish = _endOfStorage = nullptr;
78 }
79
80 // capacity
81 size_t size() const { return _finish - _start; }
82 size_t capacity() const { return _endOfStorage - _start; }
83 bool empty() const { return _first == _finish; }
84 void reserve(size_t n)
85 {
86     if (n > capacity())
87     {
88         size_t oldSize = size();
89         T* tmp = new T[n];
90
91         // 这里直接使用memcpy?
92         //if (_start)
93         //    memcpy(tmp, _start, sizeof(T)*size);
94
95         if (_start)
96         {
97             for (size_t i = 0; i < oldSize; ++i)
98                 tmp[i] = _start[i];
99         }
100
101         _start = tmp;
102         _finish = _start + size;
103         _endOfStorage = _start + n;
104     }
105 }
106
107 void resize(size_t n, const T& value = T())
108 {
109     // 1.如果n小于当前的size, 则数据个数缩小到n
110     if (n <= size())
111     {
112         _finish = _start + n;
113         return;
114     }
115
116     // 2.空间不够则增容
117     if (n > capacity())
118         reserve(n);
119

```

```

120 // 3.将size扩大到n
121 iterator it = _finish;
122 iterator _finish = _start + n;
123 while (it != _finish)
124 {
125     *it = value;
126     ++it;
127 }
128 }
129
130 ////////////////access////////////////////////
131 T& operator[](size_t pos){return _start[pos];}
132 const T& operator[](size_t pos)const {return _start[pos];}
133
134 ////////////////modify////////////////////////
135 void push_back(const T& x){insert(end(), x);}
136 void pop_back(){erase(--end());}
137
138 void swap(vector<T>& v)
139 {
140     swap(_start, v._start);
141     swap(_finish, v._finish);
142     swap(_endOfStorage, v._endOfStorage);
143 }
144
145 iterator insert(iterator pos, const T& x)
146 {
147     assert(pos <= _finish);
148
149     // 空间不够先进行增容
150     if (_finish == _endOfStorage)
151     {
152         size_t size = size();
153         size_t newCapacity = (0 == capacity())? 1 : capacity() * 2;
154         reserve(newCapacity);
155
156         // 如果发生了增容, 需要重置pos
157         pos = _start + size;
158     }
159
160     iterator end = _finish - 1;
161     while (end >= pos)
162     {
163         *(end + 1) = *end;
164         --end;
165     }
166
167     *pos = x;
168     ++_finish;
169     return pos;
170 }
171
172 // 返回删除数据的下一个数据

```



```

173 // 方便解决:一边遍历一边删除的迭代器失效问题
174 iterator erase(iterator pos)
175 {
176     // 挪动数据进行删除
177     iterator begin = pos + 1;
178     while (begin != _finish) {
179         *(begin - 1) = *begin;
180         ++begin;
181     }
182
183     --_finish;
184
185     return pos;
186 }
187 private:
188 iterator _start;        // 指向数据块的开始
189 iterator _finish;       // 指向有效数据的尾
190 iterator _endOfStorage; // 指向存储容量的尾
191 };
192 }

```

2.2 使用memcpy拷贝问题

假设模拟实现的vector中的reserve接口中,使用memcpy进行的拷贝,以下代码会发生什么问题?

```

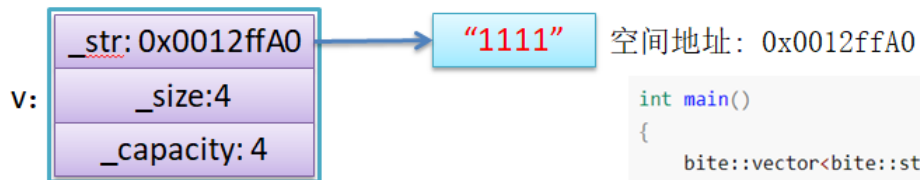
1 int main()
2 {
3     bite::vector<bite::string> v;
4     v.push_back("1111");
5     v.push_back("2222");
6     v.push_back("3333");
7     return 0;
8 }

```

问题分析:

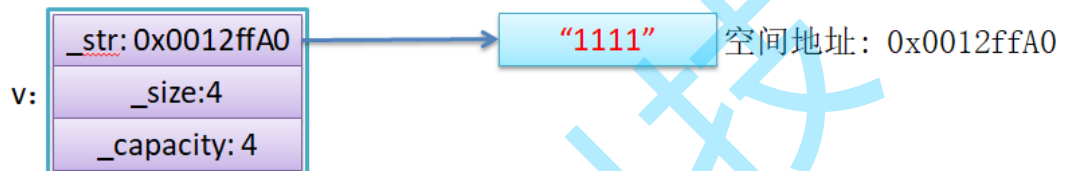
1. memcpy是内存的二进制格式拷贝,将一段内存空间中内容原封不动的拷贝到另外一段内存空间中
2. 如果拷贝的是自定义类型的元素,memcpy即高效又不会出错,但如果拷贝的是自定义类型元素,并且自定义类型元素中涉及到资源管理时,就会出错,因为memcpy的拷贝实际是浅拷贝。

插入"1111"之后的vector

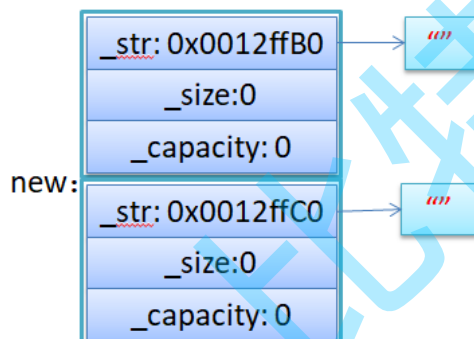


```
int main()
{
    bite::vector<bite::string> v;
    v.push_back("1111");
    v.push_back("2222");
    v.push_back("3333");
    return 0;
}
```

插入"2222"期间, 需要进行扩容:

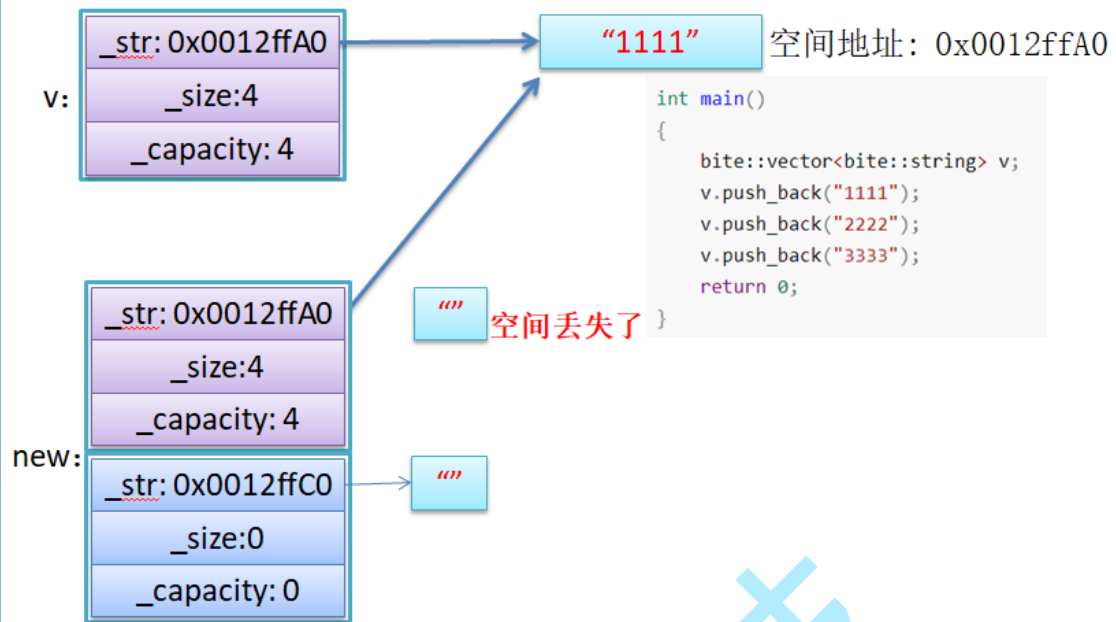


1. 开辟新空间

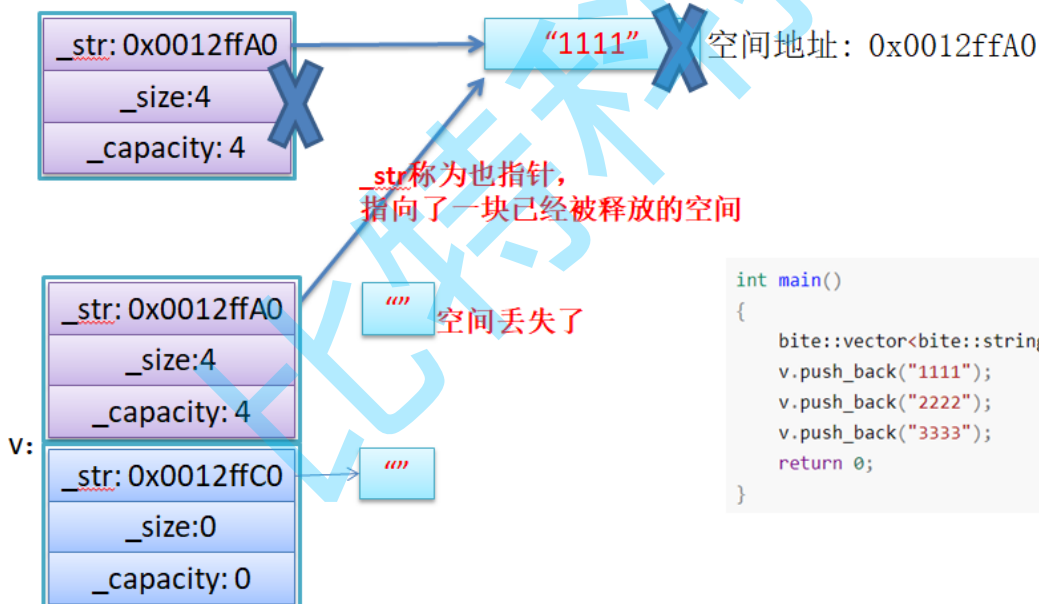


```
int main()
{
    bite::vector<bite::string> v;
    v.push_back("1111");
    v.push_back("2222");
    v.push_back("3333");
    return 0;
}
```

2. 拷贝元素：采用memcpy拷贝



3. 释放旧空间



结论：如果对象中涉及到资源管理时，千万不能使用memcpy进行对象之间的拷贝，因为memcpy是浅拷贝，否则可能会引起内存泄漏甚至程序崩溃。

2.2 对bite::vector核心接口的测试

```
1 // constructing vectors
2 void TestVector1()
3 {
4     // constructors used in the same order as described above:
5     bite::vector<int> first; // empty vector of
```

```

ints
6     bite::vector<int> second(4, 100); // four ints with value
100
7     bite::vector<int> third(second.Begin(), second.End()); // iterating through
second
8     bite::vector<int> fourth(third); // a copy of third
9
10    // the iterator constructor can also be used to construct from arrays:
11    int myints[] = { 16, 2, 77, 29 };
12    bite::vector<int> fifth(myints, myints + sizeof(myints) / sizeof(int));
13
14    std::cout << "The contents of fifth are:";
15    for (bite::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
16        std::cout << *it << " ";
17    std::cout << endl;
18
19    // 测试T是string时, 拷贝问题
20    bite::vector<string> strV;
21    strV.PushBack("1111");
22    strV.PushBack("2222");
23    strV.PushBack("3333");
24    strV.PushBack("4444");
25    for (size_t i = 0; i < strV.size(); ++i)
26    {
27        cout << strV[i] << " ";
28    }
29    cout << endl;
30 }
31
32 //vector iterator的使用
33 void PrintVector(const bite::vector<int>& v)
34 {
35     // 使用const迭代器进行遍历打印
36     bite::vector<int>::const_iterator it = v.begin();
37     while (it != v.end())
38     {
39         cout << *it << " ";
40         ++it;
41     }
42     cout << endl;
43 }
44
45 void TestVector2()
46 {
47     // 使用push_back插入4个数据
48     bite::vector<int> v;
49     v.push_back(1);
50     v.push_back(2);
51     v.push_back(3);
52     v.push_back(4);
53     PrintVector(v);
54
55     // 使用迭代器进行修改

```

```

56     auto it = v.begin();
57     while (it != v.end())
58     {
59         *it *= 2;
60         ++it;
61     }
62
63     PrintVector(v);
64
65     // 这里可以看出C++11支持iterator及接口，就支持范围for
66     for(auto e : v)
67         cout<<e<<" ";
68 }
69
70 // find / insert / erase
71 void TestVector3()
72 {
73     int a[] = { 1, 2, 3, 4 };
74     bite::vector<int> v(a, a + sizeof(a) / sizeof(a[0]));
75
76     // 使用find查找3所在位置的iterator
77     auto pos = find(v.begin(), v.end(), 3);
78
79     // 在pos位置之前插入30
80     v.insert(pos, 30);
81     PrintVector(v);
82
83     // 删除pos位置的数据
84     pos = find(v.begin(), v.end(), 3);
85     v.Erase(pos);
86     PrintVector(v);
87 }
88
89 // iterator失效问题
90 void TestVector4()
91 {
92     int a[] = { 1, 2, 3, 4 };
93     bite::vector<int> v(a, a + sizeof(a) / sizeof(a[0]));
94
95     // 删除pos位置的数据，导致pos迭代器失效
96     auto pos = find(v.begin(), v.end(), 3);
97     v.erase(pos);
98     cout << *pos << endl; // 此处会导致非法访问
99
100    // 在pos位置插入数据，导致pos迭代器失效。
101    // insert会导致迭代器失效，是因为insert可
102    // 能会导致扩容，扩容后pos还指向原来的空间，而原来的空间已经释放了。
103    pos = find(v.begin(), v.end(), 3);
104    v.insert(pos, 30);
105    cout << *pos << endl; // 此处会导致非法访问
106
107    // 实现删除v中的所有偶数
108
109    // 下面的程序会崩溃掉，如果是偶数，erase导致it失效

```

```
109 // 对失效的迭代器进行++it, 会导致程序崩溃
110 auto it = v.begin();
111 while (it != v.end())
112 {
113     if (*it % 2 == 0)
114         v.erase(it);
115
116     ++it;
117 }
118
119 // 以上程序要改成下面这样, erase会返回删除位置的下一个位置
120 it = v.begin();
121 while (it != v.end())
122 {
123     if (*it % 2 == 0)
124         it = v.erase(it);
125     else
126         ++it;
127 }
128 }
```