

Lesson09---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, list and forward_list），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list和forward_list统一的迭代器和引用更好。

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。

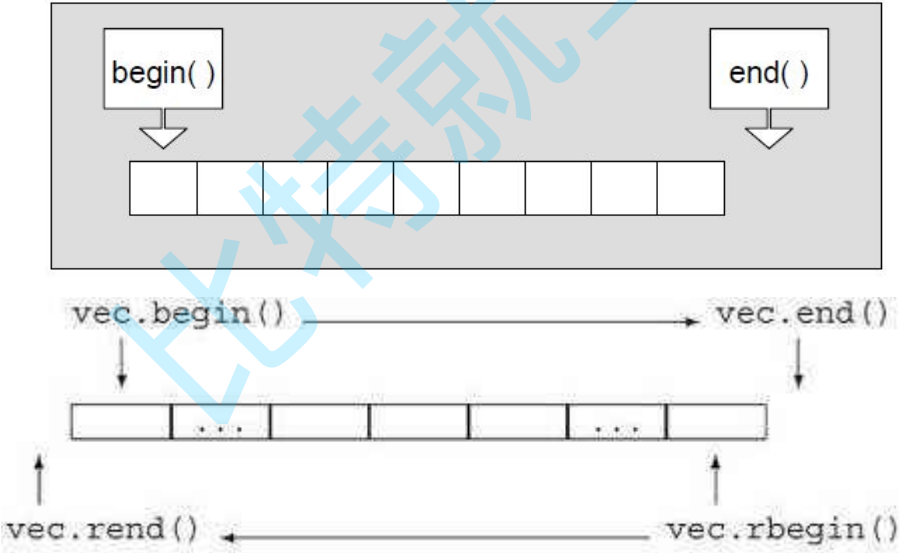
1.2.1 vector的定义

(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

[vector的构造代码演示](#)

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



[vector的迭代器使用代码演示](#)

1.2.3 vector 空间增长问题

容量空间	接口说明
size	获取数据个数
capacity	获取容量大小
empty	判断是否为空
resize (重点)	改变vector的size
reserve (重点)	改变vector的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，vector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  // 测试vector的默认扩容机制
2  void TestVectorExpand()
3  {
4      size_t sz;
5      vector<int> v;
6      sz = v.capacity();
7      cout << "making v grow:\n";
8      for (int i = 0; i < 100; ++i)
9      {
10         v.push_back(i);
11         if (sz != v.capacity())
12         {
13             sz = v.capacity();
14             cout << "capacity changed: " << sz << '\n';
15         }
16     }
17 }

```

18 vs: 运行结果: vs下使用的STL基本是按照1.5倍方式扩容

19 making foo grow:

```

20 capacity changed: 1
21 capacity changed: 2
22 capacity changed: 3
23 capacity changed: 4
24 capacity changed: 6
25 capacity changed: 9
26 capacity changed: 13
27 capacity changed: 19
28 capacity changed: 28
29 capacity changed: 42
30 capacity changed: 63
31 capacity changed: 94

```

```

33 capacity changed: 141
34
35 g++运行结果: linux下使用的STL基本是按照2倍方式扩容
36 making foo grow:
37 capacity changed: 1
38 capacity changed: 2
39 capacity changed: 4
40 capacity changed: 8
41 capacity changed: 16
42 capacity changed: 32
43 capacity changed: 64
44 capacity changed: 128

```

```

1 // 如果已经确定vector中要存储元素大概个数, 可以提前将空间设置足够
2 // 就可以避免边插入边扩容导致效率低下的问题了
3 void TestVectorExpandOP()
4 {
5     vector<int> v;
6     size_t sz = v.capacity();
7     v.reserve(100); // 提前将容量设置好, 可以避免一遍插入一遍扩容
8     cout << "making bar grow:\n";
9     for (int i = 0; i < 100; ++i)
10    {
11        v.push_back(i);
12        if (sz != v.capacity())
13        {
14            sz = v.capacity();
15            cout << "capacity changed: " << sz << '\n';
16        }
17    }
18 }

```

[vector容量接口使用代码演示](#)

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

1.2.4 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 }

```

3. 注意: Linux下, g++编译器对迭代器失效的检测并不是非常严格, 处理也没有vs下极端。

```

1  // 1. 扩容之后, 迭代器已经失效了, 程序虽然可以运行, 但是运行结果已经不对了
2  int main()
3  {
4      vector<int> v{1,2,3,4,5};
5      for(size_t i = 0; i < v.size(); ++i)
6          cout << v[i] << " ";
7      cout << endl;
8
9      auto it = v.begin();
10     cout << "扩容之前, vector的容量为: " << v.capacity() << endl;
11     // 通过reserve将底层空间设置为100, 目的是为了让vector的迭代器失效
12     v.reserve(100);
13     cout << "扩容之后, vector的容量为: " << v.capacity() << endl;
14
15     // 经过上述reserve之后, it迭代器肯定会失效, 在vs下程序就直接崩溃了, 但是linux下不会
16     // 虽然可能运行, 但是输出的结果是不对的
17     while(it != v.end())
18     {
19         cout << *it << " ";
20         ++it;
21     }
22     cout << endl;
23     return 0;
24 }

```

程序输出:

1 2 3 4 5

扩容之前, vector的容量为: 5

扩容之后, vector的容量为: 100

0 2 3 4 5 409 1 2 3 4 5

```

32 // 2. erase删除任意位置代码后, linux下迭代器并没有失效
33 // 因为空间还是原来的空间, 后序元素往前搬移了, it的位置还是有效的
34 #include <vector>
35 #include <algorithm>
36
37 int main()
38 {
39     vector<int> v{1,2,3,4,5};
40     vector<int>::iterator it = find(v.begin(), v.end(), 3);
41     v.erase(it);

```

```

42     cout << *it << endl;
43     while(it != v.end())
44     {
45         cout << *it << " ";
46         ++it;
47     }
48     cout << endl;
49     return 0;
50 }
51
52 程序可以正常运行，并打印：
53 4
54 4 5
55
56 // 3: erase删除的迭代器如果是最后一个元素，删除之后it已经超过end
57 // 此时迭代器是无效的，++it导致程序崩溃
58 int main()
59 {
60     vector<int> v{1,2,3,4,5};
61     // vector<int> v{1,2,3,4,5,6};
62     auto it = v.begin();
63     while(it != v.end())
64     {
65         if(*it % 2 == 0)
66             v.erase(it);
67         ++it;
68     }
69
70     for(auto e : v)
71         cout << e << " ";
72     cout << endl;
73     return 0;
74 }
75
76 =====
77 // 使用第一组数据时，程序可以运行
78 [sly@VM-0-3-centos 20220114]$ g++ testVector.cpp -std=c++11
79 [sly@VM-0-3-centos 20220114]$ ./a.out
80 1 3 5
81 =====
82 // 使用第二组数据时，程序最终会崩溃
83 [sly@VM-0-3-centos 20220114]$ vim testVector.cpp
84 [sly@VM-0-3-centos 20220114]$ g++ testVector.cpp -std=c++11
85 [sly@VM-0-3-centos 20220114]$ ./a.out
86 Segmentation fault

```

从上述三个例子中可以看到：SGI STL中，迭代器失效后，代码并不一定会崩溃，但是运行结果肯定不对，如果it不在begin和end范围内，肯定会崩溃的。

4. 与vector类似，string在插入+扩容操作+erase之后，迭代器也会失效

```

1 #include <string>
2 void TestString()

```



```

3  {
4      string s("hello");
5      auto it = s.begin();
6
7      // 放开之后代码会崩溃，因为resize到20会string会进行扩容
8      // 扩容之后，it指向之前旧空间已经被释放了，该迭代器就失效了
9      // 后序打印时，再访问it指向的空间程序就会崩溃
10     //s.resize(20, '!');
11     while (it != s.end())
12     {
13         cout << *it;
14         ++it;
15     }
16     cout << endl;
17
18     it = s.begin();
19     while (it != s.end())
20     {
21         it = s.erase(it);
22         // 按照下面方式写，运行时程序会崩溃，因为erase(it)之后
23         // it位置的迭代器就失效了
24         // s.erase(it);
25         ++it;
26     }
27 }

```

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  // 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]
3  class Solution {
4  public:
5      vector<vector<int>> generate(int numRows) {
6          vector<vector<int>> vv(numRows);
7          for(int i = 0; i < numRows; ++i)
8          {
9              vv[i].resize(i+1, 1);

```

```

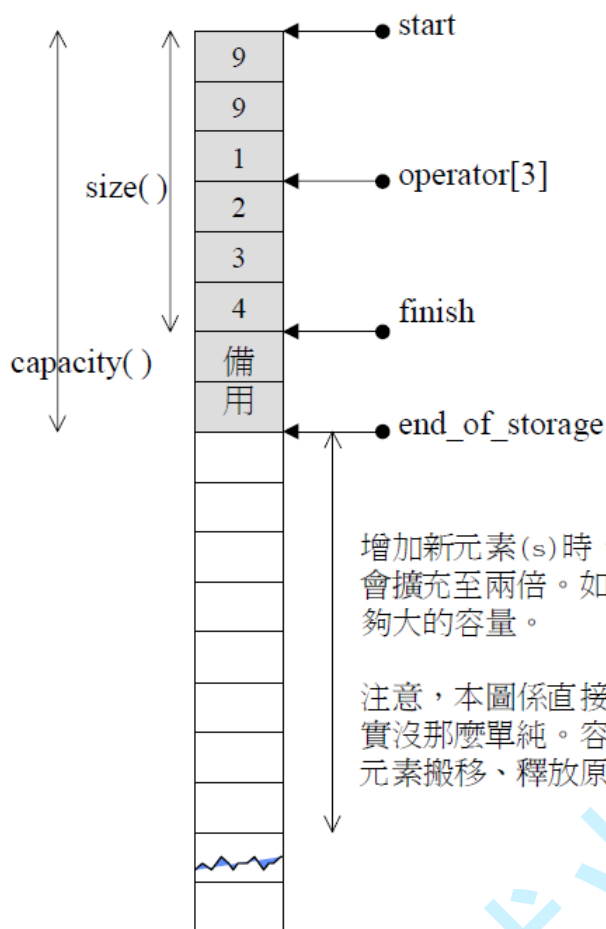
10     }
11
12     for(int i = 2; i < numRows; ++i)
13     {
14         for(int j = 1; j < i; ++j)
15         {
16             vv[i][j] = vv[i-1][j] + vv[i-1][j-1];
17         }
18     }
19
20     return vv;
21 }
22 };

```

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。课下自己实现一遍上面课堂讲解的OJ练习，然后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 OJ](#)
4. [只出现一次的数ii OJ](#)
5. [只出现一次的数iii OJ](#)
6. [数组中出现次数超过一半的数字 OJ](#)
7. [电话号码字母组合 OJ](#)

2.vector深度剖析及模拟实现



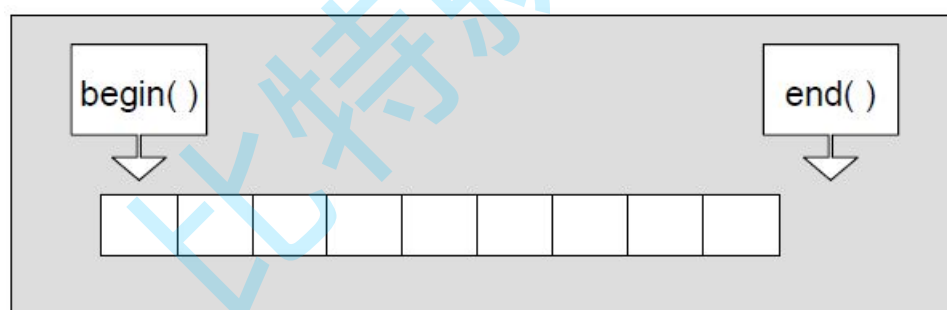
經過以下動作：

```
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

[vector的模拟实现](#)

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？

```

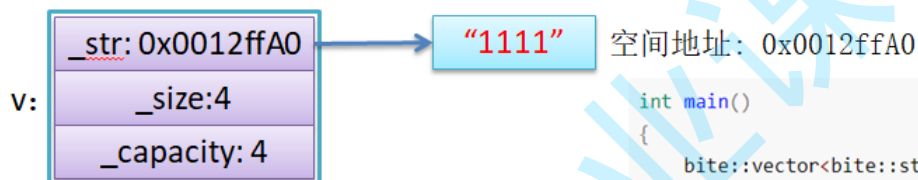
1  int main()
2  {
3      bite::vector<bite::string> v;
4      v.push_back("1111");
5      v.push_back("2222");
6      v.push_back("3333");
7      return 0;
8  }

```

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. 如果拷贝的是内置类型的元素，memcpy既高效又不会出错，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

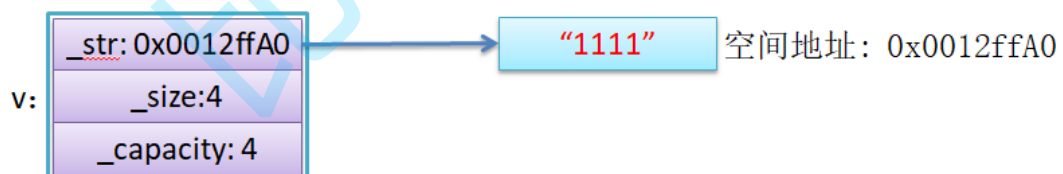


```

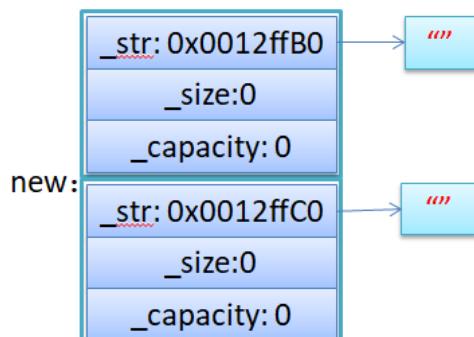
int main()
{
    bite::vector<bite::string> v;
    v.push_back("1111");
    v.push_back("2222");
    v.push_back("3333");
    return 0;
}

```

插入"2222"期间，需要进行扩容：



1. 开辟新空间

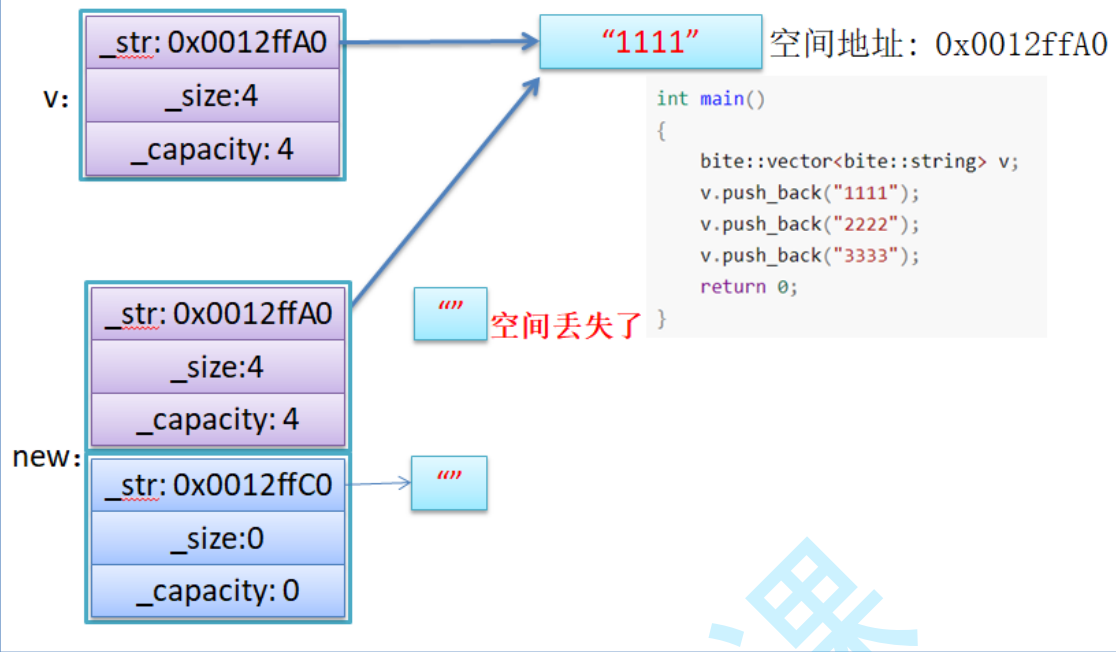


```

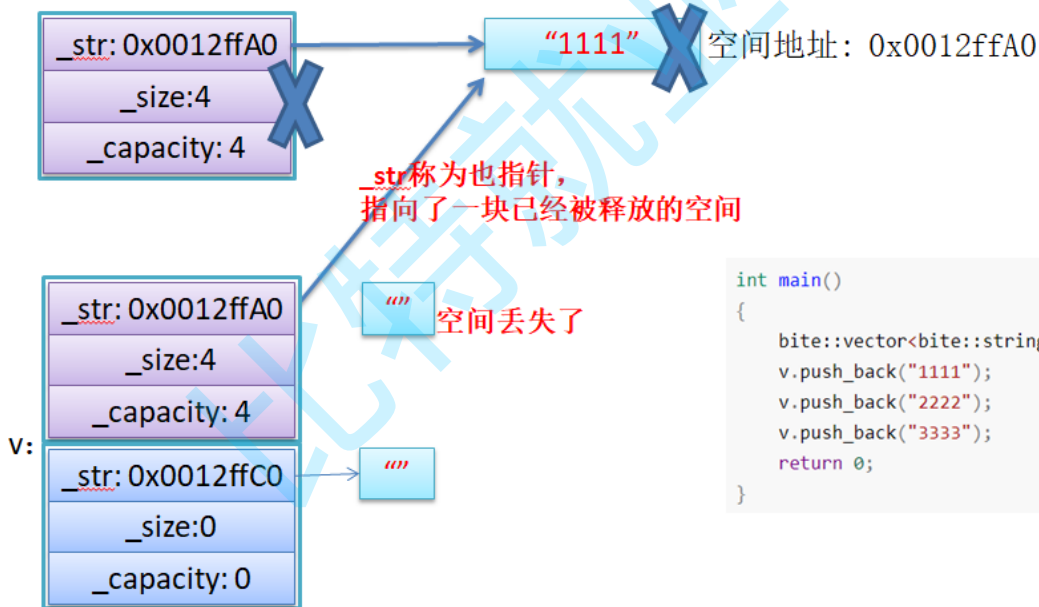
int main()
{
    bite::vector<bite::string> v;
    v.push_back("1111");
    v.push_back("2222");
    v.push_back("3333");
    return 0;
}

```

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 动态二维数组理解

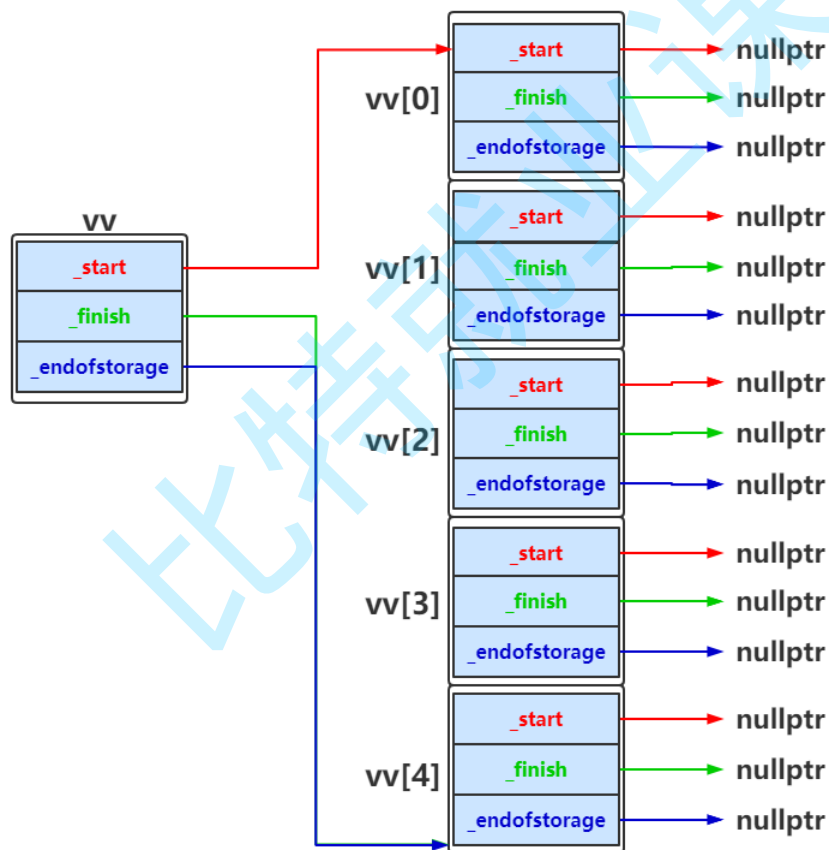
```
1 // 以杨慧三角的前n行为例：假设n为5
2 void test2vector(size_t n)
3 {
4     // 使用vector定义二维数组vv，vv中的每个元素都是vector<int>
```

```

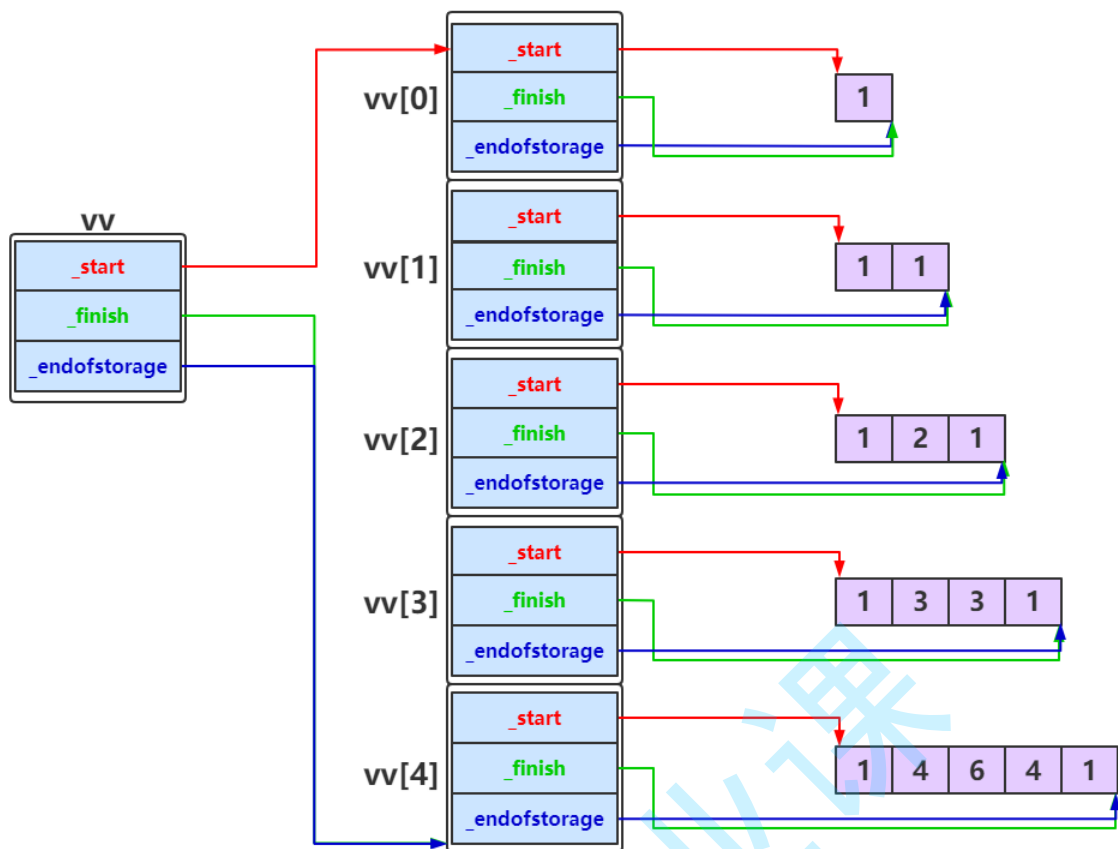
5   bit::vector<bit::vector<int>> vv(n);
6
7   // 将二维数组每一行中的vecotr<int>中的元素全部设置为1
8   for (size_t i = 0; i < n; ++i)
9       vv[i].resize(i + 1, 1);
10
11  // 给杨慧三角出第一列和对角线的所有元素赋值
12  for (int i = 2; i < n; ++i)
13  {
14      for (int j = 1; j < i; ++j)
15      {
16          vv[i][j] = vv[i - 1][j] + vv[i - 1][j - 1];
17      }
18  }
19  }

```

`bit::vector<bit::vector<int>> vv(n);` 构造一个w动态二维数组，vv中总共有n个元素，每个元素都是vector类型的，每行没有包含任何元素，如果n为5时如下所示：



vv中元素填充完成之后，如下图所示：



使用标准库中vector构建动态二维数组时与上图实际是一致的。