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，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（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。
>
> 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

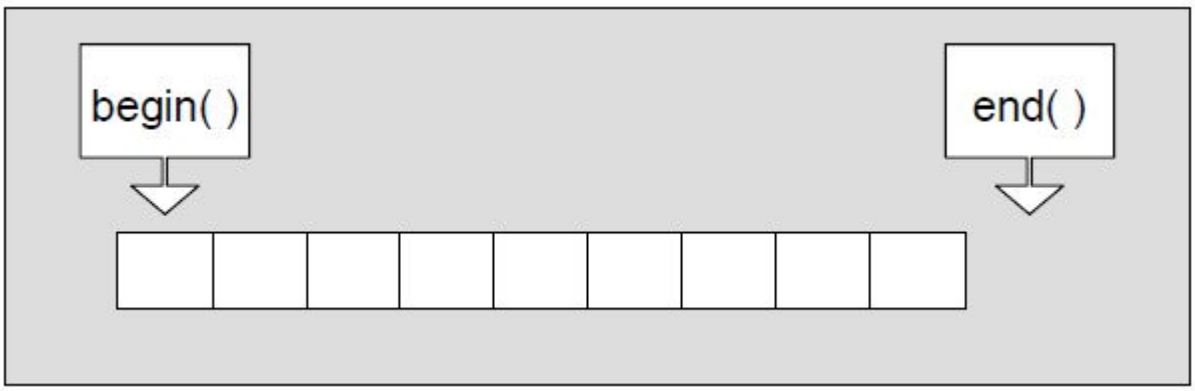
(constructor)构造函数声明	接口说明
vector()（重点）	无参构造
vector（size_type n, const value_type& val = value_type()）	构造并初始化n个val
vector (const vector& x);（重点）	拷贝构造
vector (InputIterator first, InputIterator last);	使用迭代器进行初始化构造

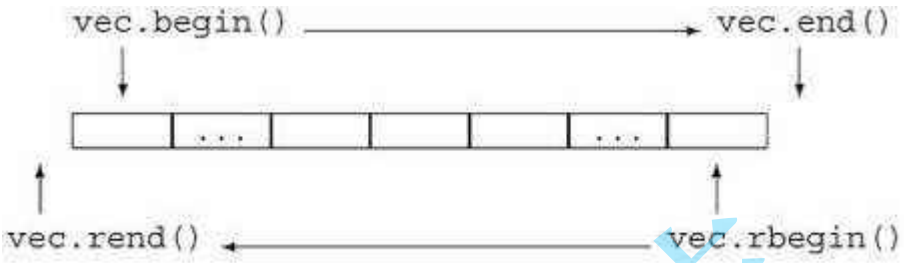
```
// constructing vectors
#include <iostream>
#include <vector>

int main ()
{
  // constructors used in the same order as described above:
  std::vector<int> first;                                // empty vector of ints
  std::vector<int> second (4,100);                       // four ints with value 100
  std::vector<int> third (second.begin(),second.end());  // iterating through second
  std::vector<int> fourth (third);                       // a copy of third

  // 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
  // the iterator constructor can also be used to construct from arrays:
  int myints[] = {16,2,77,29};
  std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  std::cout << "The contents of fifth are:";
  for (std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

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

比特科技






```cpp
#include <iostream>
#include <vector>
using namespace std;

void PrintVector(const vector<int>& v)
{
    // const对象使用const迭代器进行遍历打印
    vector<int>::const_iterator it = v.begin();
    while (it != v.end())
    {
        cout << *it << " ";
        ++it;
    }
    cout << endl;
}

int main()
{
    // 使用push_back插入4个数据
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);

    // 使用迭代器进行遍历打印
    vector<int>::iterator it = v.begin();
    while (it != v.end())
    {
        cout << *it << " ";
        ++it;

    }
```

比特科技

```
    cout << endl;

    // 使用迭代器进行修改
    it = v.begin();
    while (it != v.end())
    {
        *it *= 2;
        ++it;
    }

    // 使用反向迭代器进行遍历再打印
    vector<int>::reverse_iterator rit = v.rbegin();
    while (rit != v.rend())
    {
        cout << *rit << " ";
        ++rit;
    }
    cout << endl;

    PrintVector(v);

    return 0;
}
```

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。

```
// vector::capacity
#include <iostream>
#include <vector>

int main ()
{
```

比特科技

```
  size_t sz;
  std::vector<int> foo;
  sz = foo.capacity();
  std::cout << "making foo grow:\n";
  for (int i=0; i<100; ++i) {
    foo.push_back(i);
    if (sz!=foo.capacity()) {
      sz = foo.capacity();
      std::cout << "capacity changed: " << sz << '\n';
    }
  }
}

vs: 运行结果:
making foo grow:
capacity changed: 1
capacity changed: 2
capacity changed: 3
capacity changed: 4
capacity changed: 6
capacity changed: 9
capacity changed: 13
capacity changed: 19
capacity changed: 28
capacity changed: 42
capacity changed: 63
capacity changed: 94
capacity changed: 141

g++运行结果:
making foo grow:
capacity changed: 1
capacity changed: 2
capacity changed: 4
capacity changed: 8
capacity changed: 16
capacity changed: 32
capacity changed: 64
capacity changed: 128
```

```
// vector::reserve
#include <iostream>
#include <vector>

int main ()
{
  size_t sz;
  std::vector<int> foo;
  sz = foo.capacity();
  std::cout << "making foo grow:\n";
  for (int i=0; i<100; ++i) {

    foo.push_back(i);
```

比特科技

```
    if (sz!=foo.capacity()) {
      sz = foo.capacity();
      std::cout << "capacity changed: " << sz << '\n';
    }
  }

  std::vector<int> bar;
  sz = bar.capacity();
  bar.reserve(100);   // this is the only difference with foo above
  std::cout << "making bar grow:\n";
  for (int i=0; i<100; ++i) {
    bar.push_back(i);
    if (sz!=bar.capacity()) {
      sz = bar.capacity();
      std::cout << "capacity changed: " << sz << '\n';
    }
  }
  return 0;
}
```

```
// vector::resize
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;

  // set some initial content:
  for (int i=1;i<10;i++)
      myvector.push_back(i);

  myvector.resize(5);
  myvector.resize(8,100);
  myvector.resize(12);

  std::cout << "myvector contains:";
  for (int i=0;i<myvector.size();i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';

  return 0;
}
```

1.2.3 vector 增删查改

比特科技

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

```cpp
// push_back/pop_back
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int a[] = { 1, 2, 3, 4 };
    vector<int> v(a, a+sizeof(a)/sizeof(int));

    vector<int>::iterator it = v.begin();
    while (it != v.end()) {
        cout << *it << " ";
        ++it;
    }
    cout << endl;

    v.pop_back();
    v.pop_back();

    it = v.begin();
    while (it != v.end()) {
        cout << *it << " ";
        ++it;
    }
    cout << endl;

    return 0;
}
```

```cpp
// find / insert / erase
#include <iostream>
#include <algorithm>
#include <vector>
```

比特科技

```cpp
using namespace std;

int main()
{
    int a[] = { 1, 2, 3, 4 };
    vector<int> v(a, a + sizeof(a) / sizeof(int));

    // 使用find查找3所在位置的iterator
    vector<int>::iterator pos = find(v.begin(), v.end(), 3);

    // 在pos位置之前插入30
    v.insert(pos, 30);

    vector<int>::iterator it = v.begin();
    while (it != v.end()) {
        cout << *it << " ";
        ++it;
    }
    cout << endl;

    pos = find(v.begin(), v.end(), 3);
    // 删除pos位置的数据
    v.erase(pos);

    it = v.begin();
    while (it != v.end()) {
        cout << *it << " ";
        ++it;
    }
    cout << endl;

    return 0;
}
```

```cpp
// operator[]+index 和 C++11中vector的新式for+auto的遍历
// vector使用这两种遍历方式是比较便捷的。
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int a[] = { 1, 2, 3, 4 };
    vector<int> v(a, a + sizeof(a) / sizeof(int));

    // 通过[]读写第0个位置。
    v[0] = 10;
    cout << v[0] << endl;

    // 通过[i]的方式遍历vector
    for (size_t i = 0; i < v.size(); ++i)

        cout << v[i] << " ";
```

比特科技

```
    cout << endl;

    vector<int> swapv;
    swapv.swap(v);

    cout << "v data:";
    for (size_t i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    cout << "swapv data:";
    for (size_t i = 0; i < swapv.size(); ++i)
        cout << swapv[i] << " ";
    cout << endl;

    // C++11支持的新式范围for遍历
    for(auto x : v)
        cout<< x << " ";
    cout<<endl;

    return 0;
}
```

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。

```
#include <iostream>
using namespace std;
#include <vector>

int main()
{
    vector<int> v{1,2,3,4,5,6};

    auto it = v.begin();

    // 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
    // v.resize(100, 8);

    // 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
    // v.reserve(100);

    // 插入元素期间，可能会引起扩容，而导致原空间被释放
    // v.insert(v.begin(), 0);
```

比特科技

```
    // v.push_back(8);

    // 给vector重新赋值，可能会引起底层容量改变
    v.assign(100, 8);

    /*
    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。
    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。
    */
    while(it != v.end())
    {
        cout<< *it << " " ;
        ++it;
    }
    cout<<endl;
    return 0;
}
```

2. 指定位置元素的删除操作--**erase**

```
#include <iostream>
using namespace std;
#include <vector>

int main()
{
    int a[] = { 1, 2, 3, 4 };
    vector<int> v(a, a + sizeof(a) / sizeof(int));

    // 使用find查找3所在位置的iterator
    vector<int>::iterator pos = find(v.begin(), v.end(), 3);

    // 删除pos位置的数据，导致pos迭代器失效。
    v.erase(pos);
    cout << *pos << endl; // 此处会导致非法访问
    return 0;
}
```

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？

```
#include <iostream>
using namespace std;

#include <vector>
```

比特科技

```
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i

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int value = 0;
        for(auto e : v) {value ^= e; }
        return value;
    }
};
```

2. 杨辉三角OJ

```
// 涉及resize / operator[]

class Solution {
```

比特科技

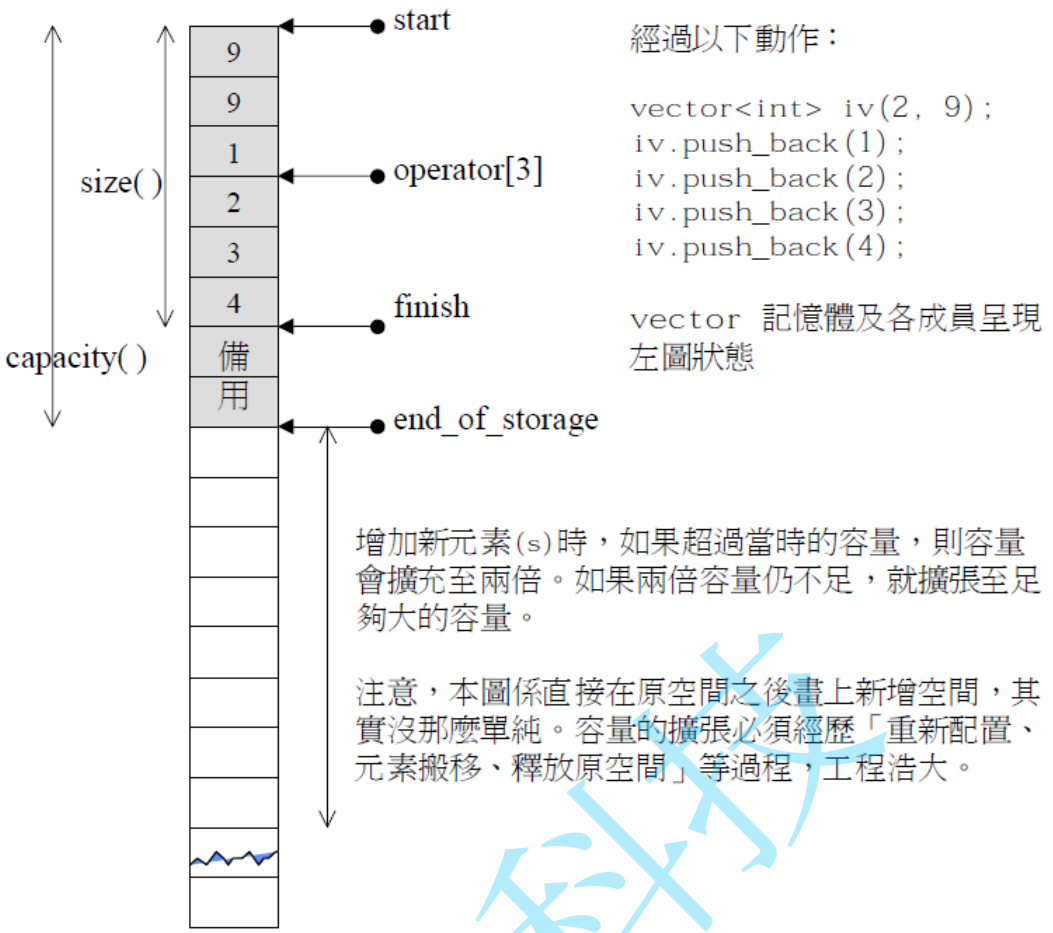
```
public:
    // 核心思想：找出杨辉三角的规律，发现每一行头尾都是1，中间第[j]个数等于上一行[j-1]+[j]
    vector<vector<int>> generate(int numRows) {
        vector<vector<int>> vv;
       // 先开辟杨辉三角的空间
        vv.resize(numRows);
        for(size_t i = 1; i <= numRows; ++i)
        {
            vv[i-1].resize(i, 0);

            // 每一行的第一个和最后一个都是1
            vv[i-1][0] = 1;
            vv[i-1][i-1] = 1;
        }

        for(size_t i = 0; i < vv.size(); ++i)
        {
            for(size_t j = 0; j < vv[i].size(); ++j)
            {
                if(vv[i][j] == 0)
                {
                    vv[i][j] = vv[i-1][j-1] + vv[i-1][j];
                }
            }
        }

        return vv;
    }
};
```

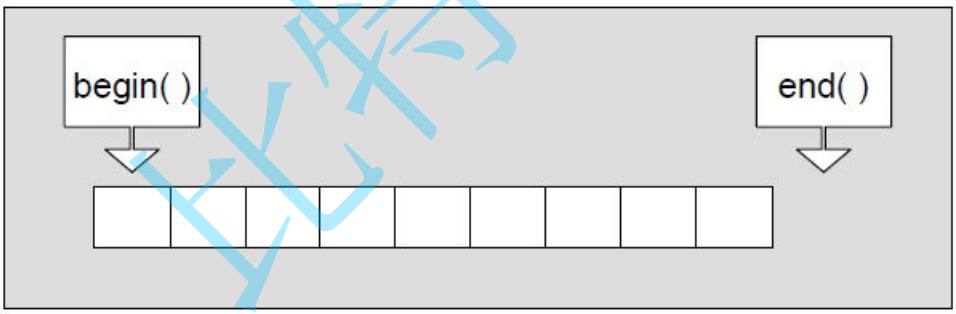
总结：通过上面的练习我们发现vector常用的接口更多是插入和遍历。遍历更喜欢用数组operator[i]的形式访问，因为这样便捷。课下除了再自己实现一遍上面课堂讲解的OJ练习，请自行完成下面题目的OJ练习。以此增强学习vector的使用。

3. 删除排序数组中的重复项 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
8. 连续子数组的最大和 OJ

2.vector深度剖析及模拟实现

比特科技





比特科技

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

```cpp
#include <iostream>
using namespace std;
#include <assert.h>

// 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。^^
namespace bit
{
template<class T>
class vector
{
public:
        // Vector的迭代器是一个原生指针
        typedef T* iterator;
```

```cpp
        typedef const T* const_iterator;

        iterator begin() { return _start; }
        iterator end() { return _finish; }

        const_iterator cbegin() const { return _start; }
        const_iterator cend() const { return _finish; }

    // construct and destroy
        vector()
            : _start(nullptr)
            , _finish(nullptr)
            , _endOfStorage(nullptr)
        {}

        vector(int n, const T& value = T())
            : _start(nullptr)
            , _finish(nullptr)
            , _endOfStorage(nullptr)
        {
            reserve(n);
            while (n--)
            {
                push_back(value);
            }
        }

        // 若使用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
        // 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
        template<class InputIterator>
        vector(InputIterator first, InputIterator last)
        {
            reserve(last - first);
            while (first != last)
            {
                push_back(*first);
                ++first;
            }
        }

        vector(const vector<T>& v)
            : _start(nullptr)
            , _finish(nullptr)
            , _endOfStorage(nullptr)
        {
            reserve(v.capacity());
            iterator it = begin();
            const_iterator vit = v.cbegin();
            while (vit != v.cend())
            {
                *it++ = *vit++;
            }

        }
```

比特科技

```
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 + oldSize;
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
```

比特科技

```
            // 3.将size扩大到n
            iterator it = _finish;
            _finish = _start + n;
            while (it != _finish)
            {
                *it = value;
                ++it;
            }
        }

    ///////////////access//////////////////////////////
    T& operator[](size_t pos){ return _start[pos]; }
    const T& operator[](size_t pos)const { return _start[pos]; }

    ///////////////modify//////////////////////////////
        void push_back(const T& x){ insert(end(), x); }
        void pop_back(){ erase(--end()); }

    void swap(vector<T>& v)
        {
            std::swap(_start, v._start);
            std::swap(_finish, v._finish);
            std::swap(_endOfStorage, v._endOfStorage);
        }

        iterator insert(iterator pos, const T& x)
        {
            assert(pos <= _finish);

            // 空间不够先进行增容
            if (_finish == _endOfStorage)
            {
                //size_t size = size();
                size_t newCapacity = (0 == capacity()) ? 1 : capacity() * 2;
                reserve(newCapacity);

                // 如果发生了增容，需要重置pos
                pos = _start + size();
            }

            iterator end = _finish - 1;
            while (end >= pos)
            {
                *(end + 1) = *end;
                --end;
            }

            *pos = x;
            ++_finish;
            return pos;
        }


        // 返回删除数据的下一个数据
```

比特科技

```
        // 方便解决:一边遍历一边删除的迭代器失效问题
        iterator erase(iterator pos)
        {
            // 挪动数据进行删除
            iterator begin = pos + 1;
            while (begin != _finish) {
                *(begin - 1) = *begin;
                ++begin;
            }

            --_finish;

            return pos;
        }
private:
        iterator _start;         // 指向数据块的开始
        iterator _finish;        // 指向有效数据的尾
        iterator _endOfStorage;  // 指向存储容量的尾
};
}
```

2.2 使用memcpy拷贝问题

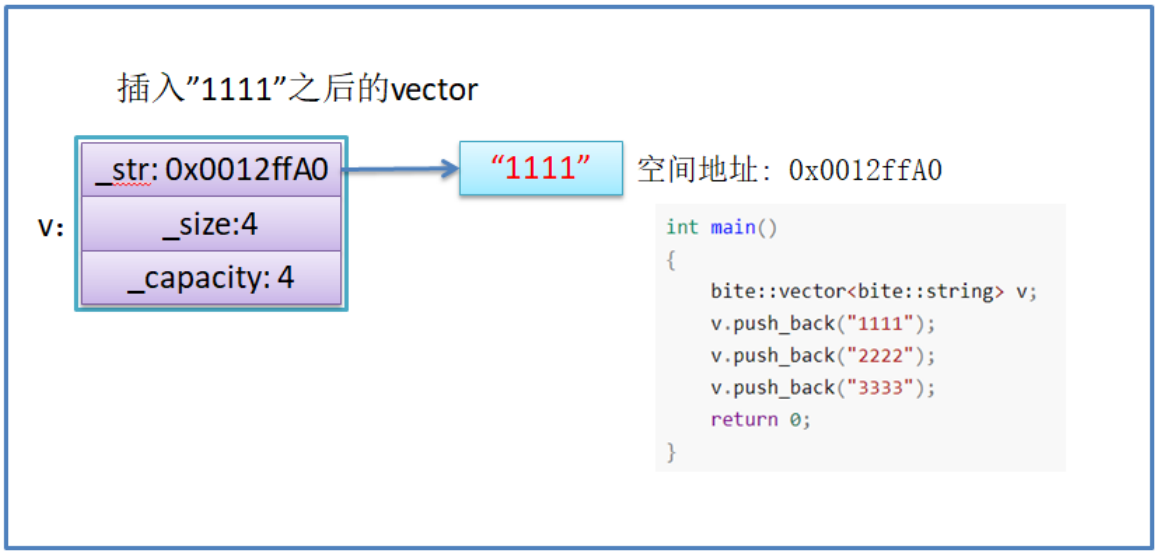
假设模拟实现的vector中的reserve接口中，使用memcpy进行的拷贝，以下代码会发生什么问题?

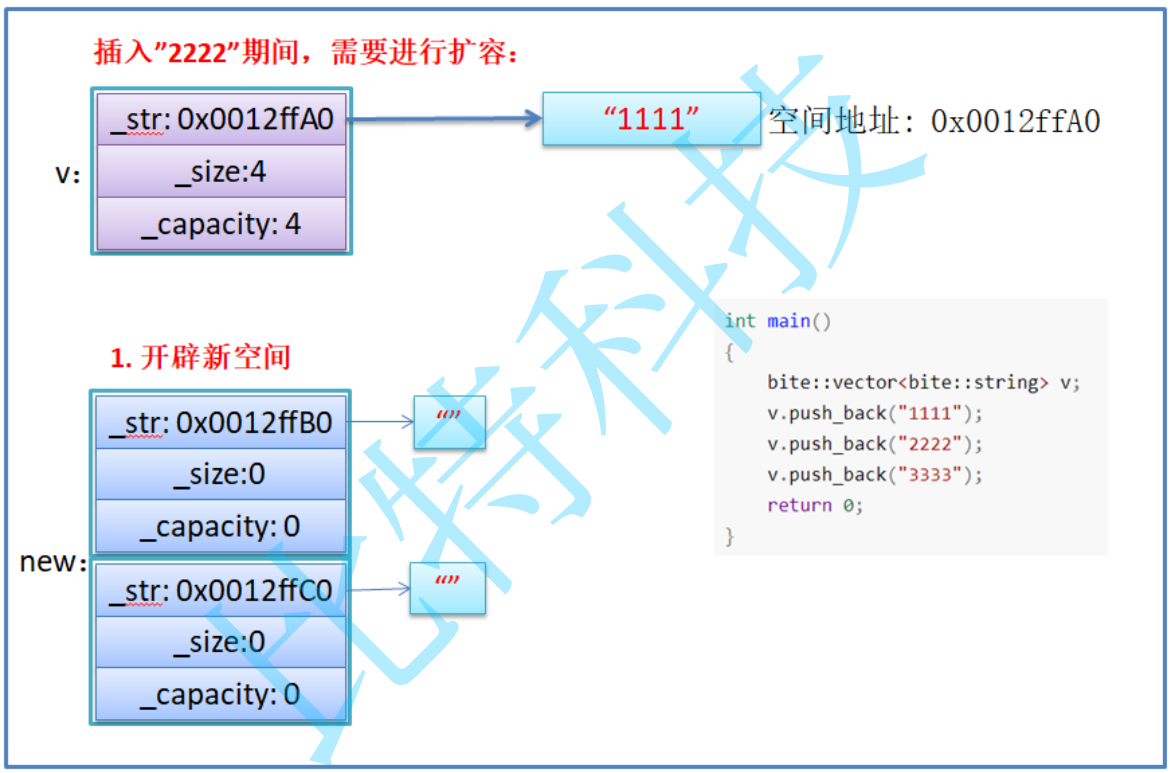
```
int main()
{
    bite::vector<bite::string> v;
    v.push_back("1111");
    v.push_back("2222");
    v.push_back("3333");
    return 0;
}
```

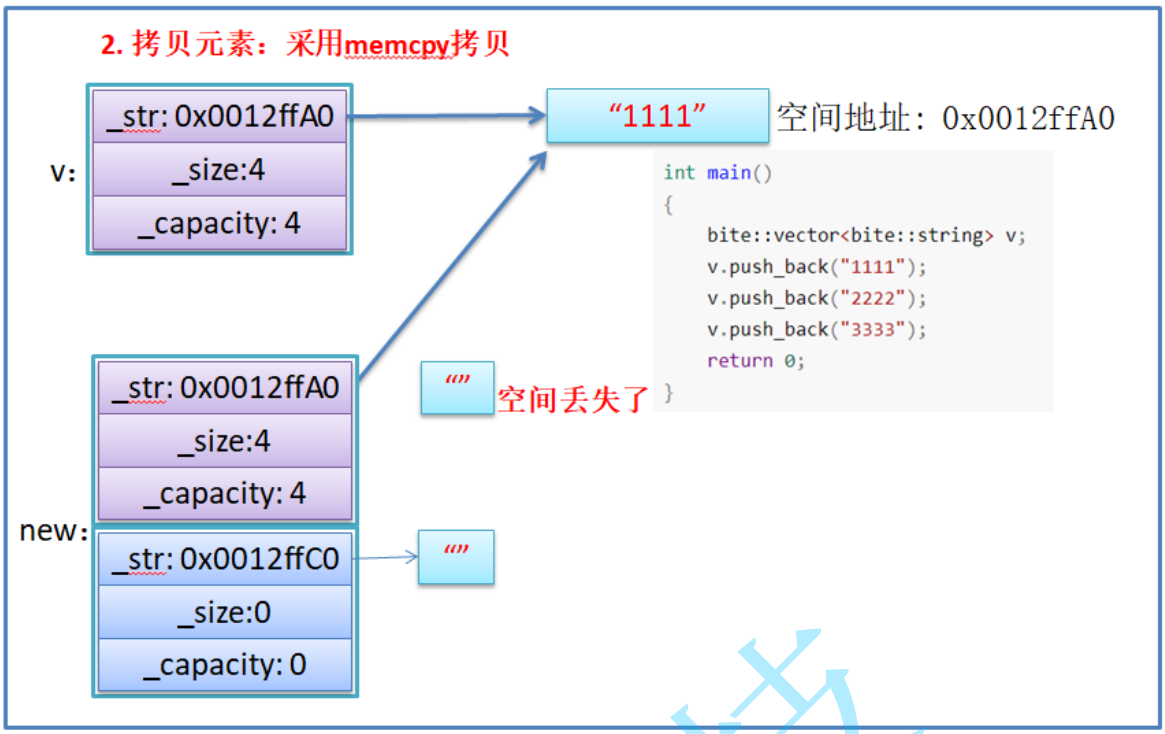
问题分析:

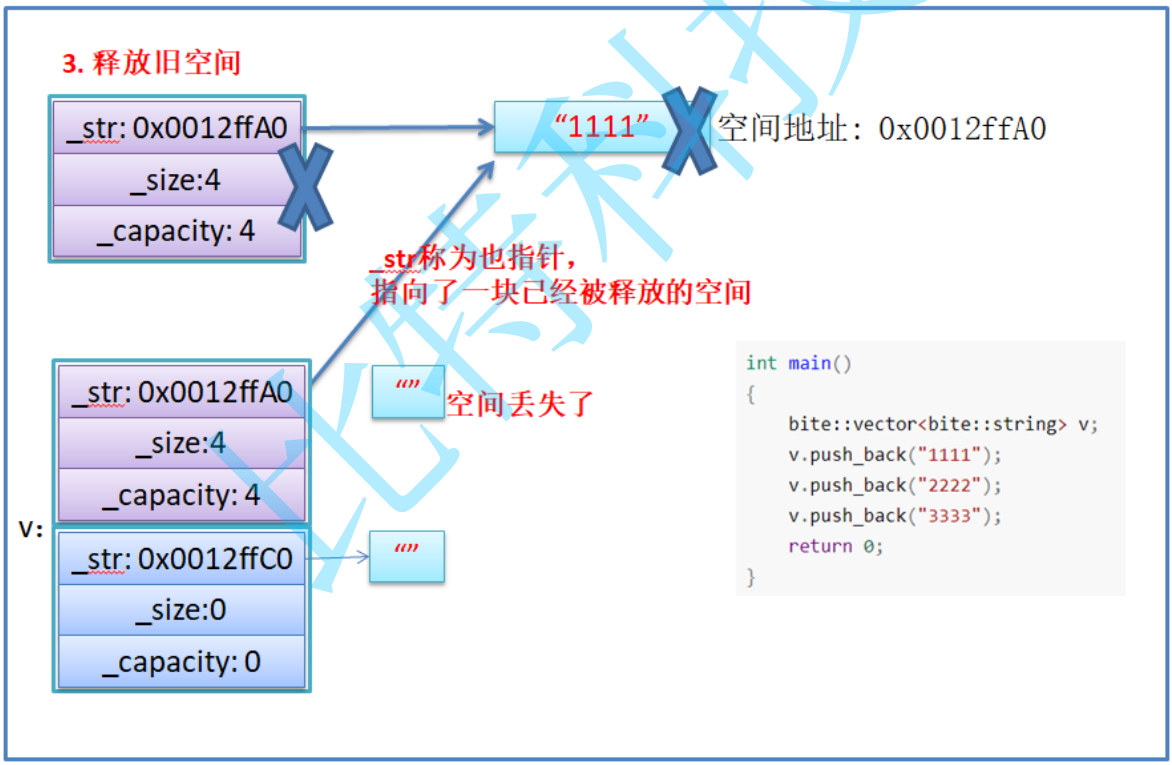
1. memcpy是内存的二进制格式拷贝，将一段内存空间中内容原封不动的拷贝到另外一段内存空间中
2. 如果拷贝的是自定义类型的元素，memcpy即高效又不会出错，但如果拷贝的是自定义类型元素，并且自定义类型元素中涉及到资源管理时，就会出错，因为memcpy的拷贝实际是浅拷贝。

比特科技


插入”1111”之后的vector
v:
_str: 0x0012ffA0
_size:4
_capacity: 4
“1111”
空间地址：0x0012ffA0
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：
v:
_str: 0x0012ffA0
_size:4
_capacity: 4
“1111”
空间地址：0x0012ffA0
1. 开辟新空间
new:
_str: 0x0012ffB0
_size:0
_capacity: 0
“”
_str: 0x0012ffC0
_size:0
_capacity: 0
“”
int main()
{
bite::vector<bite::string> v;
v.push_back("1111");
v.push_back("2222");
v.push_back("3333");
return 0;
}





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bit::vector核心接口的测试

```
// constructing vectors
void TestVector1()
{
    // constructors used in the same order as described above:

    bite::vector<int> first;                                // empty vector of
```

```
ints
    bite::vector<int> second(4, 100);                       // four ints with value
100
    bite::vector<int> third(second.Begin(), second.End());  // iterating through
second
    bite::vector<int> fourth(third);                            // a copy of third

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = { 16, 2, 77, 29 };
    bit::vector<int> fifth(myints, myints + sizeof(myints) / sizeof(int));

    std::cout << "The contents of fifth are:";
    for (bit::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
        std::cout <<*it<< " ";
    std::cout << endl;

    // 测试T是string时，拷贝问题
    bit::vector<string> strV;
    strV.PushBack("1111");
    strV.PushBack("2222");
    strV.PushBack("3333");
    strV.PushBack("4444");
    for (size_t i = 0; i < strV.size(); ++i)
    {
        cout << strV[i] << " ";
    }
    cout << endl;
}

//vector iterator的使用
void PrintVector(const bite::vector<int>& v)
{
    // 使用const迭代器进行遍历打印
    bit::vector<int>::const_iterator it = v.begin();
    while (it != v.end())
    {
        cout << *it << " ";
        ++it;
    }
    cout << endl;
}

void TestVector2()
{
    // 使用push_back插入4个数据
    bite::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    PrintVector(v);


    // 使用迭代器进行修改
```

比特科技

```
    auto it = v.begin();
    while (it != v.end())
    {
        *it *= 2;
        ++it;
    }

    PrintVector(v);

    // 这里可以看出C++11支持iterator及接口，就支持范围for
    for(auto e : v)
        cout<<e<<" ";
}

// find / insert / erase
void TestVector3()
{
    int a[] = { 1, 2, 3, 4 };
    bite::vector<int> v(a, a + sizeof(a) / sizeof(a[0]));

    // 使用find查找3所在位置的iterator
    auto pos = find(v.begin(), v.end(), 3);

    // 在pos位置之前插入30
    v.insert(pos, 30);
    PrintVector(v);

    // 删除pos位置的数据
    pos = find(v.begin(), v.end(), 3);
    v.Erase(pos);
    PrintVector(v);
}

// iterator失效问题
void TestVector4()
{
    int a[] = { 1, 2, 3, 4 };
    bite::vector<int> v(a, a + sizeof(a) / sizeof(a[0]));

    // 删除pos位置的数据，导致pos迭代器失效
    auto pos = find(v.begin(), v.end(), 3);
    v.erase(pos);
    cout << *pos << endl; // 此处会导致非法访问

    // 在pos位置插入数据，导致pos迭代器失效。
    // insert会导致迭代器失效，是因为insert可
    // 能会导致增容，增容后pos还指向原来的空间，而原来的空间已经释放了。
    pos = find(v.begin(), v.end(), 3);
    v.insert(pos, 30);
    cout << *pos << endl; // 此处会导致非法访问

    // 实现删除v中的所有偶数

    // 下面的程序会崩溃掉，如果是偶数，erase导致it失效
```

比特科技

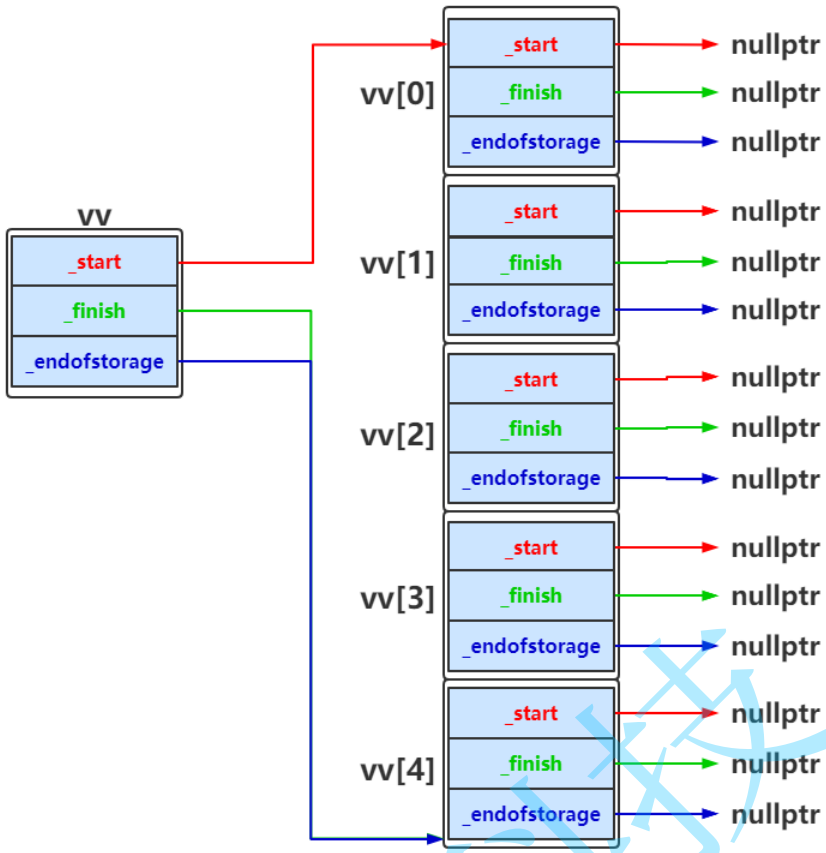
```
    // 对失效的迭代器进行++it，会导致程序崩溃
    auto it = v.begin();
    while (it != v.end())
    {
        if (*it % 2 == 0)
            v.erase(it);

        ++it;
    }

    // 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
    it = v.begin();
    while (it != v.end())
    {
        if (*it % 2 == 0)
            it = v.erase(it);
        else
            ++it;
    }
}
```

2.3 动态二维数组理解

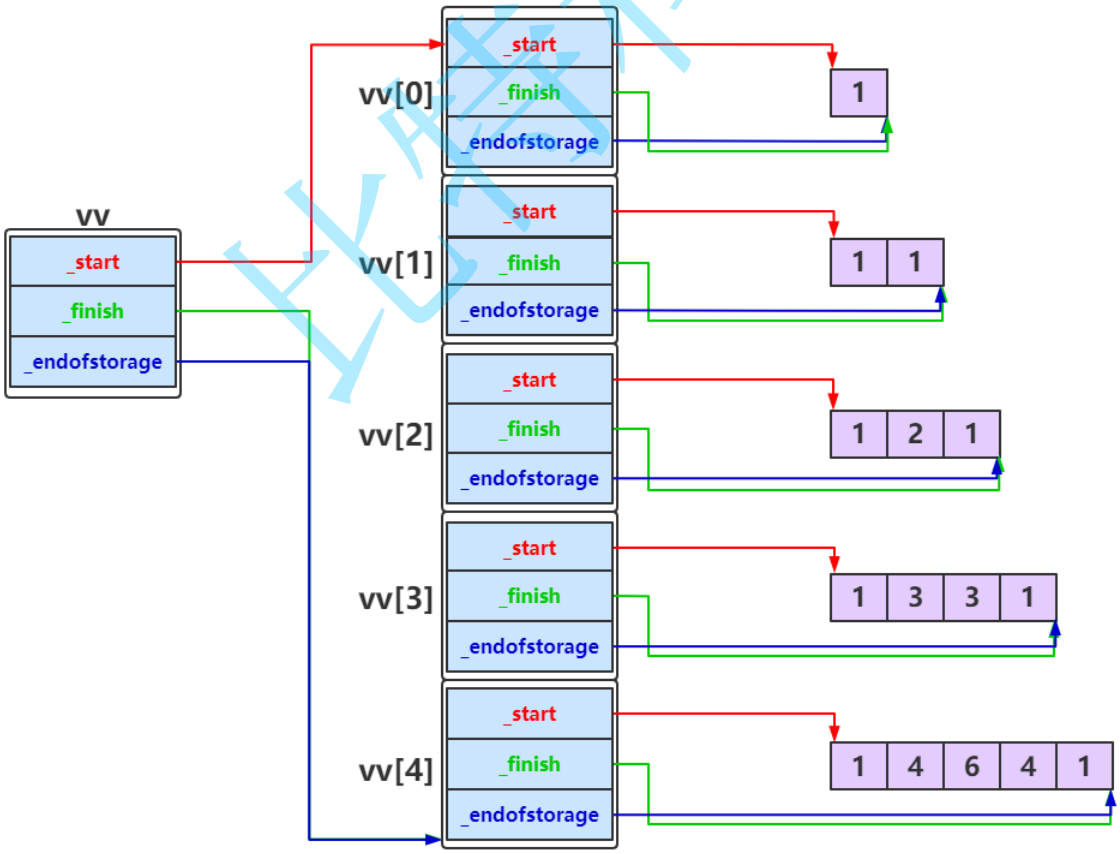
```
// 以杨慧三角的前n行为例：假设n为5
void test5(size_t n)
{
    // 使用vector定义二维数组vv，vv中的每个元素都是vector<int>
    bit::vector<bit::vector<int>> vv(n);

    // 将二维数组每一行中的vecotr<int>中的元素全部设置为1
    for (size_t i = 0; i < n; ++i)
        vv[i].resize(i + 1, 1);

    // 给杨慧三角出第一列和对角线的所有元素赋值
    for (int i = 2; i < n; ++i)
    {
        for (int j = 1; j < i; ++j)
        {
            vv[i][j] = vv[i - 1][j] + vv[i - 1][j - 1];
        }
    }
}
```

`bit::vector<bit::vector<int>> vv(n);`构造一个vv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：

比特科技



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。