vector复习

1. vector的介绍和使用

1.1 vector介绍

> vector文档

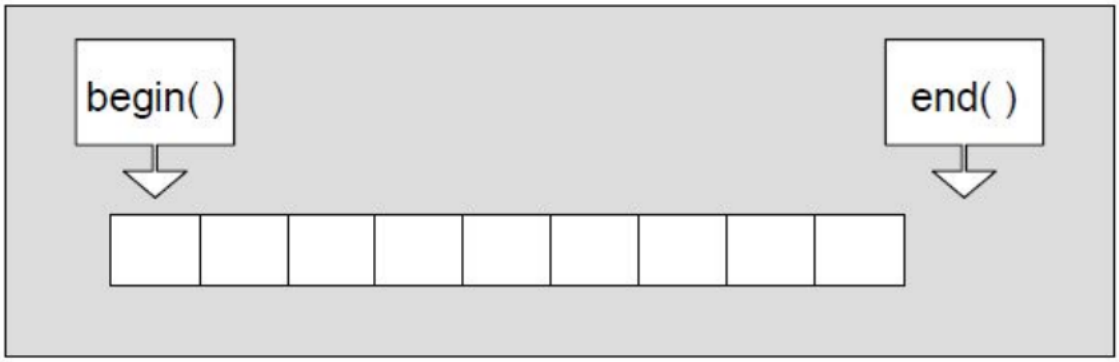
1.2 vector的使用

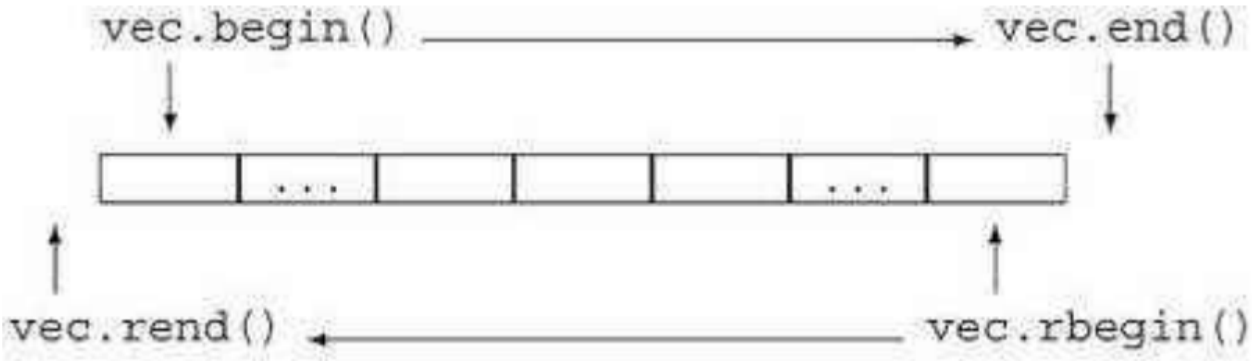
1.2.1 vector的定义

(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

1.2.2 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





1.2.3 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。

1.2.4 vector 的增删查改

vector的增删查改	接口说明
push_back	尾插
pop_back	尾删
erase	删除position位置的数据
find (algorithm中的接口)	查找
insert	在position之前插入val
operator[]	像数组一样访问
swap (algorithm中的接口)	交换两个vector的数据空间

1.2.5 ※vector 迭代器失效问题

> 在 C++ 中，`std::vector` 的迭代器可能会在某些操作后失效，继续使用这些失效的迭代器会导致未定义行为（通常是程序崩溃）。理解迭代器何时会失效是安全使用 vector 的关键。

🧐 **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)。**

1. 导致迭代器失效的常见操作

a. **插入元素 (`push_back`，`insert`，`emplace` 等)**

i. **尾部插入 (`push_back`，`emplace_back`):**

- 如果导致重新分配内存（容量不足时），所有迭代器、指针和引用都会失效
- 如果没有重新分配，只有 `end()` 迭代器会失效

ii. **中间或头部插入(`insert`，`emplace`)**

- 插入点及其后的所有迭代器都会失效
- 通常会导致所有迭代器失效（因为可能需要重新分配）

b. **删除元素(`erase`，`pop_back`，`clear`)**

> 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。

i. **删除元素(`erase`)**

- 被删除元素及其后的所有迭代器都会失效
- 删除点之前的迭代器通常保持有效

ii. **尾部删除(`pop_back`):**

- 只有 `end()` 和被删除元素的迭代器失效

iii. **清空容器(`clear`):**

- 所有迭代器都会失效

c. **改变容量(`reserve`，`resize`，`shrink_to_fit`)**

i. **增加容量(`reserve`，`resize` 增大):**

- 如果导致重新分配，所有迭代器都会失效
- 如果没有重新分配，迭代器保持有效

ii. **减少容量(`shrink_to_fit`):**

- 可能导致重新分配，使所有迭代器失效

2. 如何避免迭代器失效问题

a. **在修改操作后更新迭代器:**

代码块

```
auto it = vec.begin();
vec.insert(it, 10);  // it 现在可能失效
it = vec.begin();    // 重新获取有效的迭代器
```

b. **使用索引代替迭代器(不一定使用):**

```
代码块
1   size_t index = 2;
2   vec.erase(vec.begin() + index);
3   // 索引仍然有效（对于未被删除的元素）
```

c. **利用返回值更新迭代器:**

> - `insert` 返回指向新插入元素的迭代器
> - `erase` 返回指向被删除元素之后元素的迭代器

```
代码块
1   auto it = vec.begin() + 2;
2   it = vec.erase(it);  // it 现在指向原来的第三个元素（如果存在）
```

d. **预分配足够容量 (如果知道元素数量) :**

```
代码块
1   vec.reserve(100);  // 避免后续插入导致重新分配
```

3. 典型错误示例

◦ **错误1：在循环中删除元素**

```
代码块
1   std::vector<int> vec = {1, 2, 3, 4, 5};
2   for (auto it = vec.begin(); it != vec.end(); it++)
3       if (*it % 2 == 0) vec.erase(it);  // 错误！erase后it失效，再++会导致未定义行为
4
```

改正后:

```
代码块
1   for (auto it = vec.begin(); it != vec.end(); )
2   {
3       if (*it % 2 == 0) it = vec.erase(it);  // 使用erase的返回值更新迭代器
4       else it++;
5   }
6
```

- **错误2：插入后使用旧迭代器**

```
代码块
1   auto it = vec.begin() + 2;
2   vec.insert(it, 10);  // 可能导致重新分配
3   *it = 5;             // 错误！it可能已失效
```

改正后：

```
代码块
1   auto it = vec.begin() + 2;
2   it = vec.insert(it, 10);  // 使用insert返回的新迭代器
3   *it = 5;                  // 现在安全
```

4. 注意事项

a. `reserve()` 不改变大小，只改变容量 - 不会使迭代器失效（因为没有重新分配）

b. `swap()` 会使两个容器的所有迭代器失效 - 迭代器不会"转移"到另一个容器

c. C++11 后的 `shrink_to_fit()` - 可能导致重新分配，使迭代器失效

2. MyVector实现

库中位置

```
代码块
1   #pragma once
2   #include<iostream>
3
4   namespace Vector
5   {
6       template<class T>
7       class vector
8       {
9       public:
10          typedef T* iterator;
11          typedef const T* const_iterator;
12
13          vector(size_t n, const T& val = T())
14          {
```

```
15              _start = new T[n];
16              _end_of_storage = (_end_of_storage == nullptr ? n : (capacity() > n ? capacity() : 2 * capacity())) + _start;
17              _finish = _start + n;
18
19              for (int i = 0; i < n; i++) _start[i] = val;
20          }
21
22          vector(const vector& x)
23          {
24              _start = x._start;
25              _finish = x._finish;
26              _end_of_storage = x._end_of_storage;
27          }
28
29          vector& operator= (const vector& x)
30          {
31              vector<T> tmp(x);
32              swap(tmp);
33              return *this;
34          }
35
36          size_t size() { return _finish - _start; }
37          size_t capacity() { return _end_of_storage - _start; }
38
39          iterator begin() { return _start; }
40          iterator end() { return _finish; }
41
42          const_iterator begin() const { return _start; }
43          const_iterator end() const { return _finish; }
44
45          void swap(vector& x)
46          {
47              _start = x._start;
48              _finish = x._finish;
49              _end_of_storage = x._end_of_storage;
50          }
51
52          bool empty() { return _end_of_storage == nullptr; }
53
54          void reserve(size_t n)
55          {
56              if (n > capacity())
57              {
58                  size_t old_size = size();
59                  T* tmp = new T[n];
60                  memcpy(tmp, _start, size() * sizeof(T));
```

```
                delete[] _start;
                _start = tmp;

                _finish = _start + old_size;
                _end_of_storage = _start + n;
            }
        }

        void resize(size_t n, T val = T())
        {
            _finish = _start + n;
            for (int i = 0; i < n; i++) _start[i] = val;
        }

        T& front() { return _start[0]; }

        T& back()  { return _start[size() - 1]; }

        void push_back(const T& x)
        {
            if (_finish == _end_of_storage) reserve(2 * (_end_of_storage - _start));
            _start[size()] = x;
            _finish++;
        }

        T& operator[](size_t i) { return *(_start + i); }
```

代码块

```
        iterator insert(iterator pos, const T& val)
        {
            if (_finish == _end_of_storage) reserve(2 * (_end_of_storage - _start));

            for (int i = size() - 1; i >= (pos - _start); i--)
                    _start[i + 1] = _start[i];
            _start[pos - _start] = val;
            _finish++;

            return pos + 1;
        }

        void pop_back()
        {
            _finish--;
        }
```

```
17
18          iterator erase(iterator pos)
19          {
20              for (int i = pos - _start; i < size(); i++)
21                      _start[i - 1] = _start[i];
22              _finish--;
23
24              return pos + 1;
25          }
26
27          iterator erase(iterator first, iterator last)
28          {
29              size_t len = last - first;
30              for (int i = last - _start; i < size(); i++) _start[i - len] =
    _start[i];
31              _finish -= len;
32
33              return first + 1;
34          }
35
36      private:
37          iterator _start = nullptr;
38          iterator _finish = nullptr;
39          iterator _end_of_storage = nullptr;
40      };
41  }
```