vector

2021年3月3日 20:21

Table of Contents

- 要点
- 容器源码
- 构造函数
- 属性获取
- 操作函数
  - push和pop操作
    - push_back
    - pop_back
    - 从尾部进行删除
  - erase删除元素
  - insert插入元素
    - 插入点之后的现有元素个数 > 新增元素个数
    - 插入点之后的现有元素个数 <= 新增元素个数
    - 备用空间不足

要点

1. `vector`的迭代器是一个普通的指针
2. 构造函数的重载满足不同的用户需求
3. `vector`因为是类, 所以在生命周期结束后会自动调用析构函数, 用户不再手动的释放内存, 也不会出现内存泄露的问题, 用户也可以主动调用析构函数释放内存
4. finish是指向最后一个元素的后一位地址, 不是直接指向最后一个元素

优点

- 在内存中分配一块连续的内存空间进行存，可以像数组一样操作，动态扩容。
- 随机访问方便，支持下标访问和vector.at()操作。
- 节省空间。

缺点

- 由于其顺序存储的特性，vector 插入删除操作的时间复杂度是 O(n)。
- 只能在末端进行pop和push。
- 当动态长度超过默认分配大小后，要整体重新分配、拷贝和释放空间。

容器源码

3 个迭代器分别指向：数据的头(start)，数据的尾(finish)，数组的尾(end_of_storage)

```
template <class T, class Alloc = alloc>
class vector
{
public:
  // 定义vector自身的嵌套型别
  typedef T value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  // 定义迭代器, 这里就只是一个普通的指针
  typedef value_type* iterator;
  typedef const value_type* const_iterator;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  ...
 protected:
  typedef simple_alloc<value_type, Alloc> data_allocator; // 设置其空间配置器
  iterator start;          // 使用空间的头
  iterator finish;         // 使用空间的尾
  iterator end_of_storage;  // 可用空间的尾
  ...
};
```

构造函数

vector 有多个构造函数, 为了满足多种初始化

属性获取

位置参数的获取， 比如返回 vector 的开始和结尾，返回最后一个元素，返回当前元素个数，元素容量，是否为空等；

需要注意其中部分是返回迭代器，部分是返回元素

```
// 获取数据的开始以及结束位置的指针. 记住这里返回的是迭代器, 也就是vector迭代器就是该类型的指针.
iterator begin() { return start; }
iterator end() { return finish; }
// 获取值
reference front() { return *begin(); }
reference back() { return *(end() - 1); }
// 获取右值
const_iterator begin() const { return start; }
const_iterator end() const { return finish; }
const_reference front() const { return *begin(); }
const_reference back() const { return *(end() - 1); }
// 获取基本数组信息
size_type size() const { return size_type(end() - begin()); }    // 数组元素的个数
size_type max_size() const { return size_type(-1) / sizeof(T); }    // 最大能存储的元素个数
size_type capacity() const { return size_type(end_of_storage - begin()); }  // 数组的实际大小
bool empty() const { return begin() == end(); }     //判断vector是否为空, 并不是比较元素为0, 是直接比较头尾指针
```

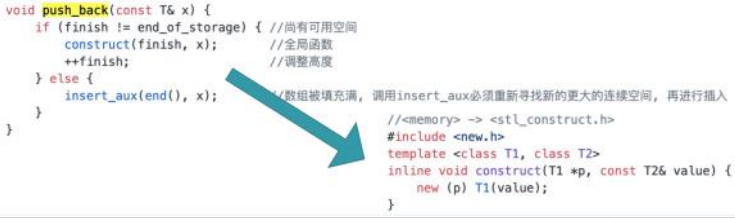
操作函数

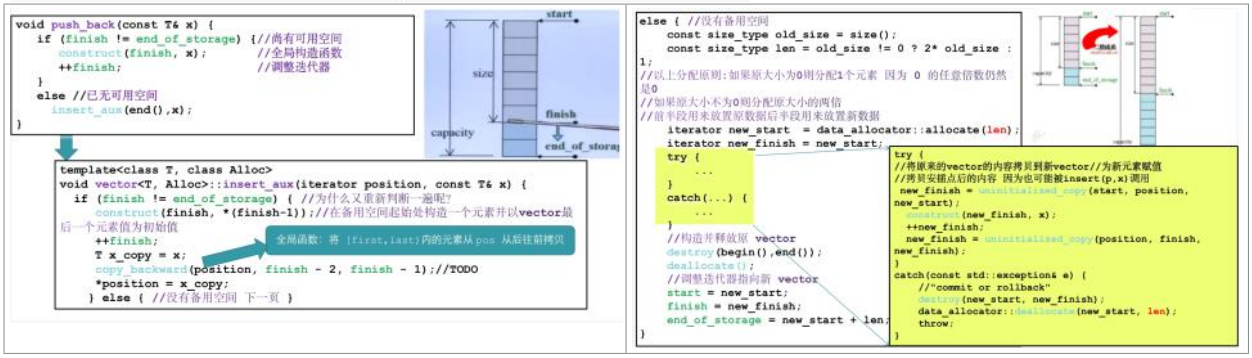
push和pop操作

vector的push和pop操作都只是对尾进行操作. 这里说的尾部是指数据的尾部, 而非数组空间的尾部.

push_back

从尾部插入数据. 当数组还有备用空间的时候就直接插入尾部就行了, 当没有备用空间后就重新寻找更大的空间再将数据全部复制过去





pop_back

从尾部进行删除

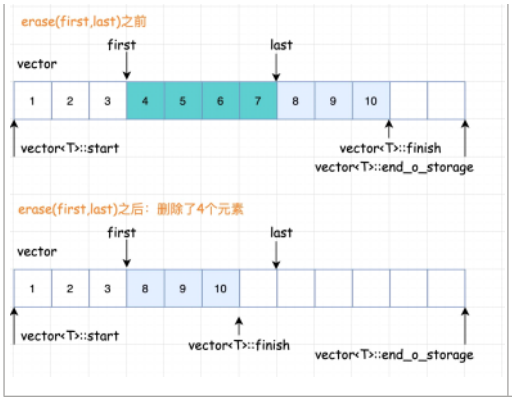
```
void pop_back()
{
    --finish;
    destroy(finish);
}
```

erase删除元素

清除指定位置的元素, 其重载函数用于清除一个范围内的所有元素. 实际实现就是将删除元素后面所有元素往前移动

1.删除指定位置的元素：就是实际就是将指定位置后面的所有元素向前移动, 最后析构掉最后一个元素

2.删除范围内的元素：第一步要将 finish 迭代器后面的元素拷贝回去，然后返回拷贝部分的尾部迭代器，最后在删除之前的。




```
// 清除指定位置的元素，实际就是将指定位置后面的所有元素向前移动，最后析构掉最后一个元素
iterator erase(iterator position)
{
    if (position + 1 != end())
        copy(position + 1, finish, position);
    --finish;
    destroy(finish);
    return position;
}

// 清除一个指定范围的元素，同样将指定范围后面的所有元素向前移动，最后析构掉整个范围的元素
// 清除的是左闭右开的区间 [ )
iterator erase(iterator first, iterator last)
{
    iterator i = copy(last, finish, first);
    destroy(i, finish);
    finish = finish - (last - first);
    return first;
}
```

insert插入元素

1、如果备用空间足够且插入点的现有元素多于新增元素；

2、如果备用空间足够且插入点的现有元素小于新增元素；

3、如果备用空间不够

4、其实还有一种，插入点为finish时，和push_back是一样的

插入点之后的现有元素个数 > 新增元素个数

分三步走：

1.先往后扩充大小为n的空间，并将finish开始的倒数n个元素复制过去（直接uninitialized_copy即可，会自动构造），在这里更新finish位置（之前需要保存旧位置）

2.把插入开始的插入点后个数减n个元素赋值到finish开始的倒数after-n的位置

3.从插入点开始填充n个x


```
if (size_type(end_of_storage - finish) >= n)
{
        T x_copy = x;
    // 保存插入位置到end的距离
        const size_type elems_after = finish - position;
        iterator old_finish = finish;
    // ******* a **********
    // 从插入的位置到数据结束的距离大于了要插入的个数n
        if (elems_after > n)
        {
        // 先构造出finish-n个大小的空间，再移动finish - n个元素的数据
            uninitialized_copy(finish - n, finish, finish);
            finish += n;
        // 在将从插入位置后的n个元素移动
            copy_backward(position, old_finish - n, old_finish);
        // 元素从插入位置开始进行填充即可
            fill(position, position + n, x_copy);
        }
```

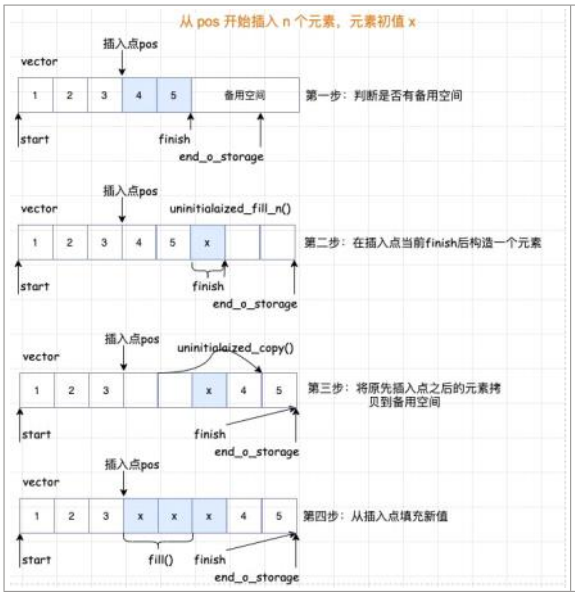
插入点之后的现有元素个数 <= 新增元素个数

分三步走：

1.finish后面构造出n-after个空间，并全赋值为x（用uninitialized_fill自动构造），在这里更新finish位置（保存旧finish）

2.插入点到旧finish全都复制到新finish后的位置（用uninitialized_copy自动构造），再次更新finish位置

3.插入点到旧finish开始fill




```
// 从插入的位置到end的距离小于了要插入的个数
    else
{
    // 先构造出n - elems_after个大小的空间，再从finish位置初始化n - elems_after为x
        uninitialized_fill_n(finish, n - elems_after, x_copy);
        finish += n - elems_after;
    // 从插入位置开始到原来的finish位置结束全部复制到新的结束位置后面
        uninitialized_copy(position, old_finish, finish);
        finish += elems_after;
    // 从插入位置进行填充x
        fill(position, old_finish, x_copy);
    }
```

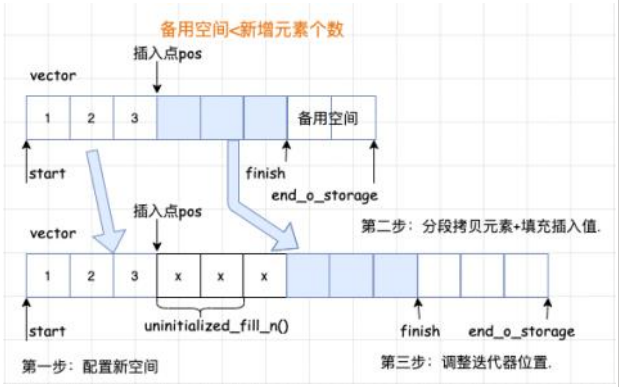
备用空间不足

分三步走：

1.重新申请空间（调用适配器），大小为max(当前的两倍，当前+n)，声明新的iterator（start和finish）

2.插入点前复制到新的空间（uninitialized_copy），更新new_finish，插入的元素接着new_finish复制，更新new_finish，插入点后元素复制，更新

3.析构原数组（destroy），deallocate释放空间，迭代器改为new的三个迭代器




```
// 空间不足处理
else
{
    // 重新申请一个当前两倍的空间或者当前大小+插入的空间，选择两者最大的方案.
    const size_type old_size = size();
    const size_type len = old_size + max(old_size, n);
    iterator new_start = data_allocator::allocate(len);
    iterator new_finish = new_start;
    __STL_TRY
    {
        // 同样进行分段复制到新的数组中，从而实现插入
        new_finish = uninitialized_copy(start, position, new_start);
        new_finish = uninitialized_fill_n(new_finish, n, x);
        new_finish = uninitialized_copy(position, finish, new_finish);
    }
    // 将当前数组的元素进行析构，最后释放空间
    destroy(start, finish);
    deallocate();
    // 修改3个迭代器
    start = new_start;
    finish = new_finish;
    end_of_storage = new_start + len;
}
```