第1章

1. STL的六大组件

* **容器**(containers): 各种数据结构, vector, list, deque, set, map, 用来存放数据. 从实现角度看, 容器是一种 class template (类模板).
* **算法**(algorithms): 各种常用算法, sort, search, copy, erase. 从实现角度看, 算法是一种 function template(函数模板)
* **迭代器**(iterators): 也称为泛型指针, 是一种重载 operator\*, operator->, operator++, operator—等指针相关操作的 class template. 所有容器附带有自己专属的迭代器, 用来遍历容器元素.
* **仿函数**(functors): 行为类似函数. 仿函数是一种重载了 operator() 的类或者模板类.
* **适配器**(adapters): 用来修饰容器或迭代器接口的东西. queue, stack 是一种容器适配器, 底部完全借助 deque, 所有操作也由 deque 提供.
* **配置器**(allocators): 负责空间配置和管理

关系: 容器通过配置器取得数据存储空间, 算法通过迭代器取得容器的内容, 仿函数可以协助算法完成不同的策略变化, 适配器可以修饰容器.

第4章 顺序容器

1. vector

Vector 采用线性连续空间(数组), 用两个迭代器 start 和 finish 分别指向由配置器得到的连续空间中已经被使用的范围, 并用迭代器 end\_of\_storage 指向整块连续空间的末尾.

为了降低空间配置时的速度成本, vector 实际配置的大小可能比客户端需求量更大, 以备将来可能的扩充. 这也就是容量的概念.

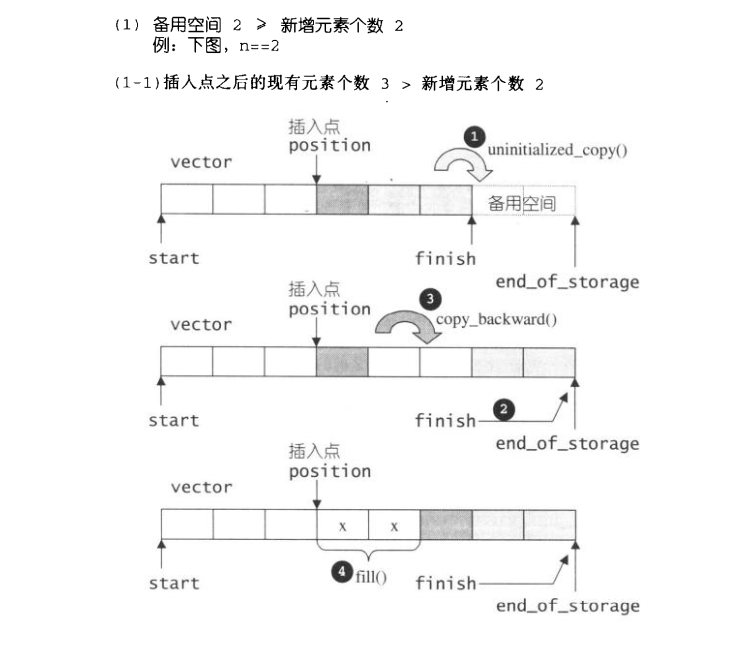
当插入新元素时, 首先检查是否是否还有备用空间, 如果有且足够就直接在备用空间上构造元素, 并调整迭代器 finish, 如果没有备用空间或者备用空间不足, 就扩充空间(重新配置, 移动数据, 释放原空间)

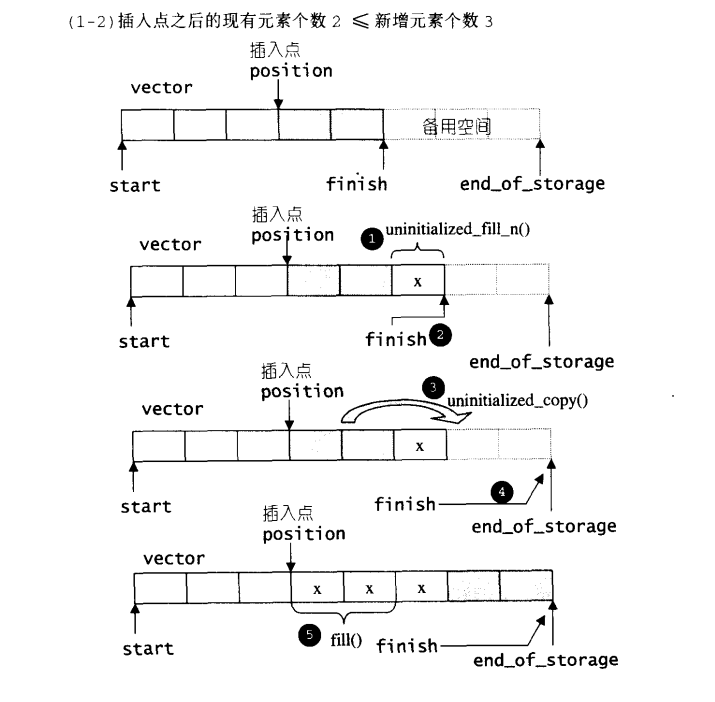
如果原大小为0, 则配置为1个元素大小;

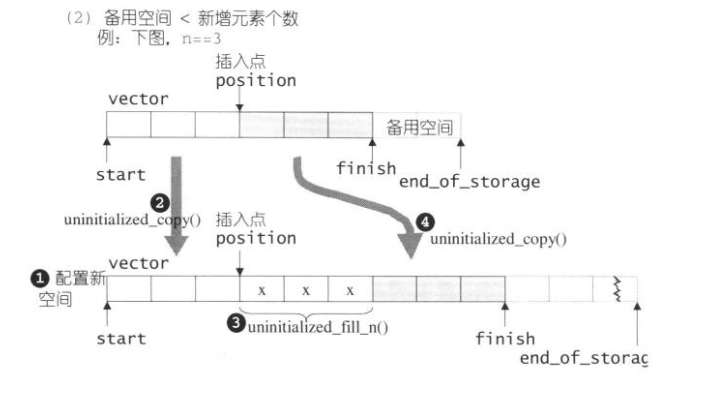
如果不为0, 则配置为原大小的两倍, 前半段用来放置原数据, 后半段备用.

如果插入多个元素, 在备用空间不够时, 新空间长度是旧长度的两倍与旧长度加新增元素个数和的最大值.

因此, 对vector 的任何操作, 如果引起空间重新配置, 指向原vector 的所有迭代器就都失效了.





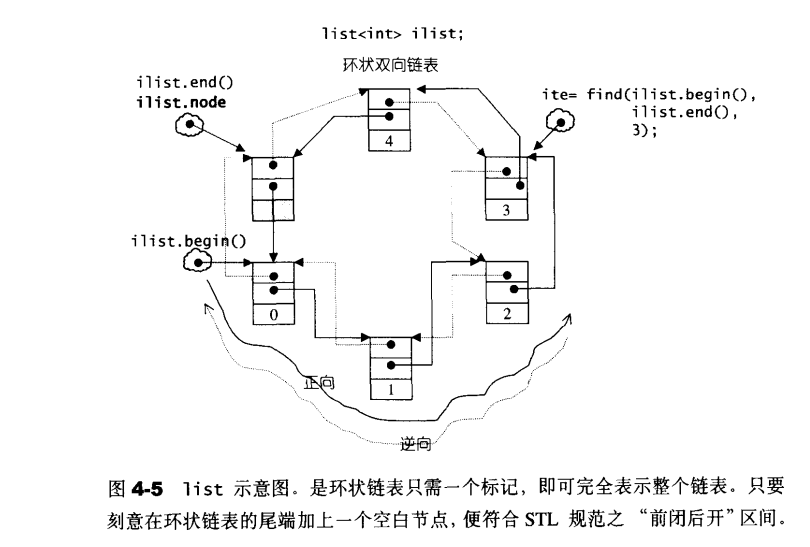


2. list

List 每次插入或删除一个元素, 就配置或释放一个元素空间. 可以常数时间对任何位置进行插入或删除操作.

List 的插入, 删除操作不会造成所有的迭代器失效. 删除操作只有指向被删除元素的那个迭代器失效,其他迭代器不受影响.

List 是一个双向环状链表, 所有它只有一个指针. 如果在list 的尾端加一个空白的节点, 让这个指针指向这个空白的节点, 就可以构造出迭代器前闭后开区间的要求.



3. deque

Deque 是一种双向开口的连续线性空间, 可以在头部和尾部进行插入和删除操作.

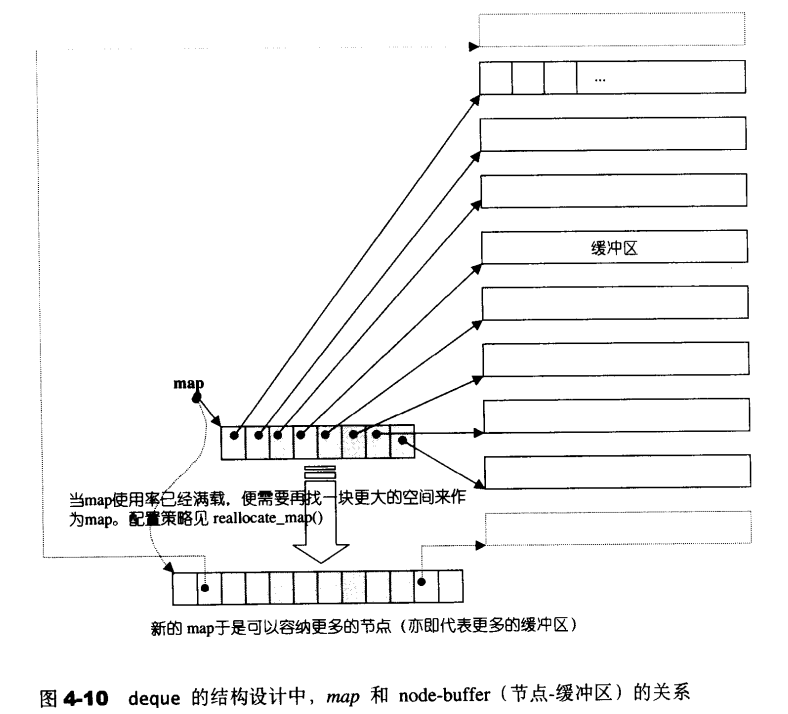
Vector也可以在头部操作,但是效率很差.

Deque和vector 的差异:

* Deque 可以在常数时间内对头部进行插入或删除操作;
* Deque 没有容量的概念, 它是动态地以分段连续空间组合而成,随时可以增加一段空间并链接起来.

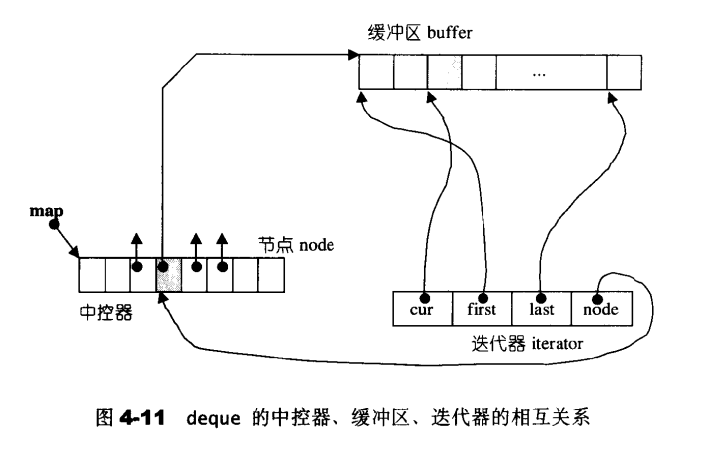
如果要对deque进行排序操作, 尽可能选择vector , 因为 deque 的随机存取效率低. 可以先把deque完整复制到一个vector中, 对vector 排序后, 再复制回 deque.

Deque 用一个 map (不是STL中的map) 指针, 指向一个连续空间, 连续空间的每个元素都是一个指针, 指向一块缓冲区



如何保持分段连续空间是整体连续的”假象”?

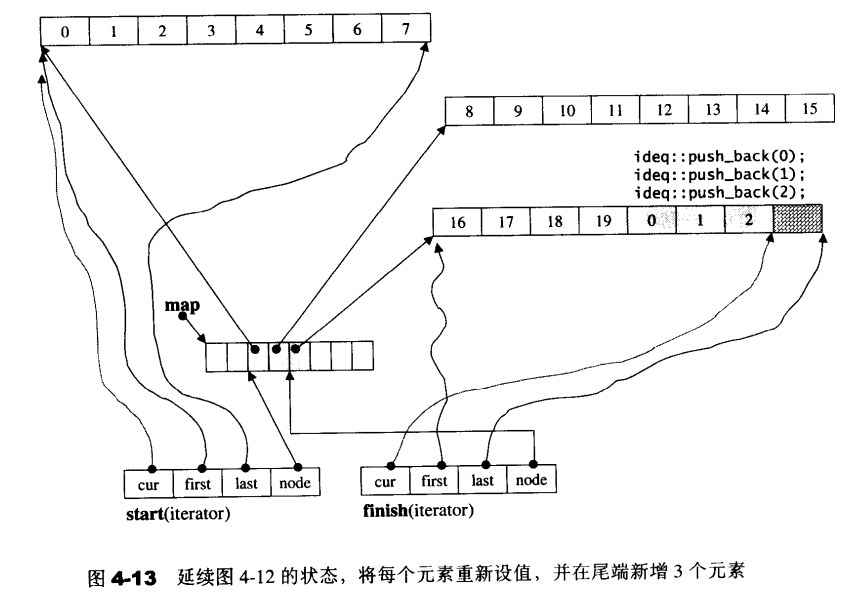
中控器

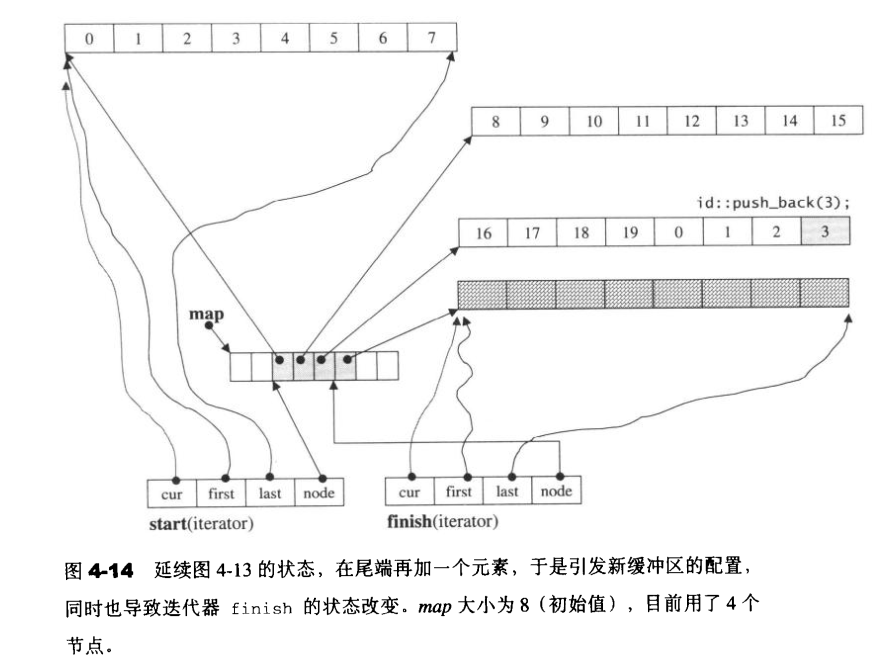


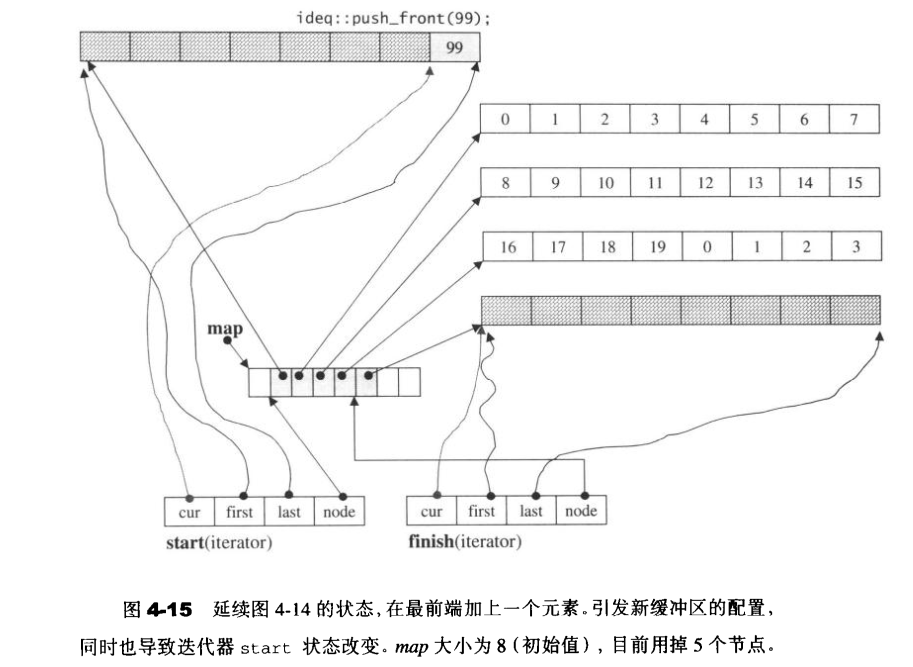
如果一个迭代器到达当前node 节点所指向缓冲区的头端或尾端, 那node 节点就需要向前或向后移动一个节点.

Deque还有start 和 finish 两个迭代器, 分别指向第一缓冲区的第一个元素和最后缓冲区的最后一个元素(的下一个位置).

如果 map 的大小所提供的节点不足, 就必须重新配置一块更大的.







4.stack

Stack 是一种先进后出(FILO)的数据结构, 只有一个出口(栈顶). 运行新增, 删除, 取得最顶端元素, 但是不能存取其他位置元素.

Stack 以 deque 作为底部结构.

Template <calss T, class Sequence = deque<T> >

Calss stack {

Public:

Typedef typename Sequence::value\_type value\_type;

Typedef typename Sequence::size\_type size\_type;

Typedef typename Sequence::reference reference;

Typedef typename Sequence::const\_reference const\_reference;

Protected:

Sequence c; //底层容器

Public:

//完全利用 Sequence c 的操作完成

Bool empty() const { return c.empty(); }

Size\_type size() const { return c.size(); }

Reference top() { return c.back(); }

Const\_reference top() { return c.back(); }

Void push(const value\_type &x) { c.push\_back(x); }

Void pop() { c.pop\_back(); }

}

5.queue

queue 是一种先进先出(FIFO)的数据结构, 只有一个出口(列首). 运行新增, 删除, 取得最前端元素, 但是不能存取其他位置元素.

queue 以 deque 作为底部结构.

Template <calss T, class Sequence = deque<T> >

Calss stack {

Public:

Typedef typename Sequence::value\_type value\_type;

Typedef typename Sequence::size\_type size\_type;

Typedef typename Sequence::reference reference;

Typedef typename Sequence::const\_reference const\_reference;

Protected:

Sequence c; //底层容器

Public:

//完全利用 Sequence c 的操作完成

Bool empty() const { return c.empty(); }

Size\_type size() const { return c.size(); }

Reference front() { return c.front(); }

Const\_reference front() { return c.front(); }

Reference back() { return c.back(); }

Const\_reference back() { return c.back(); }

Void push(const value\_type &x) { c.push\_back(x); }

Void pop() { c.pop\_front(); }

}

6. heap 算法

STL 提供max\_head(最大堆), 最大值总是在根节点, 且总是位于数组的第一个位置.

Make\_heap(beg, end); #以区间[beg, end) 内的元素建立最大堆

Pop\_heap(beg, end); #交换第一个元素和最后一个元素, 把最大值放到尾端, 同时操作范围减1(但未删除)调整堆, 使其保持最大堆

Push\_headp(beg, end); #插入一个新元素之后, 调整堆使其保持最大堆

Sort\_heap(beg, end); #对最大堆进行排序

7. priority\_queue

一个拥有权值的queue, 内部的元素不是按照加入的顺序排序, 而是按照元素的权值进行排列. 权值越高, 排在最前面. Max-heap 可以满足优先队列按照”权值高低自动递增排序”的要求.

可以用 heap 泛型算法实现优先队列

Template <calss T, calss Sequence = vector<T> >

Class priority\_queue {

Public:

Typedef typename Sequence::value\_type value\_type;

Typedef typename Sequence::size\_type size\_type;

Typedef typename Sequence::reference reference;

Typedef typename Sequence::const\_reference const\_reference;

Protected:

Sequence c; //底层容器

Public:

Priority\_queue() : c() {}

Template <class iter>

Priority\_queue(iter first, iter last) : c(first, end) { make\_heap(c.begin(), c.end()); }

Bool empty() const { return c.enpty(); }

Size\_type size() const { return c.size(); }

Const\_reference top() const { return c.front(); }

Void push(const value\_type &x) {

c.push\_back(x);

push\_heap(c.begin(), c.end()); #插入一个元素后, 调用该算法保持最大堆

}

Void pop() {

Pop\_heap(c.begin(), c.end()); #把最大值放到最后一个位置, 并调整堆

c.pop\_back(); #将最后一个位置元素删除

}

}

可以为模板指定比较类, 从而调整队列顺序.

8. set 和 map

Set 和 map 的底层实现都是红黑树(RB Tree), **红黑树的特性**:  
**（1）每个节点不是黑色，就是红色;**  
**（2）根节点是黑色;**  
**（3）如果一个节点是红色的，则它的子节点必须是黑色的;**  
**（4）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。**

**注意：如果一个节点的叶子节点为空(NULL), 可以看成为黑节点.**

**根据规则5, 可以保证没有一条路径会比其他路径长出两倍. 因此红黑树是相对接近平衡的二叉树.**

**性质:**

**查找, 插入, 删除等操作时间复杂度为O(lgn);**

**在最坏情况下, 基本的动态几何操作时间为O(lgn);**

**应用:**

**主要用于存储有序数据,**

**结构:**

**Color 颜色**

**Value 值**

**Parent 父节点**

**Left 左子节点**

**Right 右子节点**

**根据规则4, 插入一个节点时, 这个节点必须是红色; 插入这个节点之后, 可能会破坏规则(3)**