# 容器概观与分类

* 研究数据的特定排列方式；
* 目的：利于搜寻、排序或其他特殊目的，该学科称为数据结构；
* 常用的数据结构：array、vector、heap（以算法形式呈现: xxx\_heap）、list（list、slist）、tree（AVL树、RB树）、stack、queue（普通队列、优先级队列两种）、hash table（ 衍生hash\_set、hash\_map、hash\_multiset、hash\_multimap ）、set（set、mutilset两种）、map（map、multimap两种）;
* 根据“数据在容器中的排列”特性，这些数据结构分为“序列式 和 关联式 （两种）”
* 数据结构底层分析：

1. heap : 底层实现 vector;
2. priority-queue : 底层实现 heap;
3. stack和queue : 底层实现 deque;
4. set/map/multiset/multimap : 底层实现 RB-tree ;
5. hash\_x :底层实现 hash table;

* 容器源码查看方式：

1. 第一部分：容器迭代器
2. 第二部分：容器基类的两种实现方式，使用 “\_\_STL\_USE\_STD\_ALLOCATORS”该宏区分；（默认使用第二种实现方式）
3. 第三部分：容器类，继承于容器基类
4. STL源码

1.先调用空间配置器，配置空间；

2.在调用构造函数（使用定位表达式），为该空间赋值；

1. 查看源码规则

* 配置空间、构造对象、析构对象、归还空间；
* 增删查改

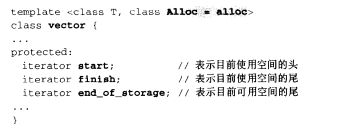
# vector

1. #include <vector>,但SGI STL将vector实现于更底层的<stl\_vector.h>
2. 查看源码之前，研究“内存处理函数”(见“第2章”)；
3. array静态空间，vector动态空间；
4. vector实现技术，关键在于对“空间大小的配置”及“空间重新配置时的数据移动“效率,空间不足时，扩充空间为原来的两倍，然后复制元素进入新空间，在释放旧空间
5. vector支持随机存取，它的迭代器为普通指针（RamdonAccess Iterator）
6. iterator start ; //表示目前使用空间的头

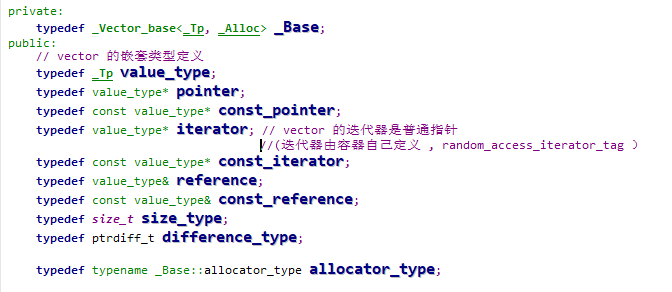
iterator finish ; //表示目前使用空间的尾

iterator end\_of\_storage; //表示目前可用空间的尾；

## vector数据结构



## vector 迭代器



## vector 构造与内存管理

* vector容量（capacity）和大小（size）是存在区别的：

capacity：start -> end\_of\_storage 的距离；

size : start -> finish 的距离；

* vector 配置空间：

vector ::vector() : 构造时，调用 allocator<T> 空间配置器 ，配置空间；

* vector 构造对象：

push\_back ()/ insert() : 插入元素时，调用全局consturct 构造器，给“迭代器所指的空间”添加值；

* 旧空间被占满时：

\_M\_insert\_aux() ： 重新分配原来2倍的空间，将旧空间的值uninitialized\_copy 到新空间中，然后释放旧空间；

因此，对于vecotr来说，一旦引起空间重新配置，指向原vector的所有迭代器就都失效了

## vector 元素操作(查看测试程序)

//获取开始迭代器

Iterator begin() ;

//获取结束迭代器

Iterator end() ;

// 检查容器是否为空

bool empty() const ;

//返回当前元素个数；

size\_type size ();

//返回vector的容量

size\_type capacity() ；

//获取第一个元素

reference front()；

//获取最后一个元素

reference back()；

//从尾部插入一个元素

void push\_back(const \_Tp& \_\_x)；

//指定位置插入一个元素

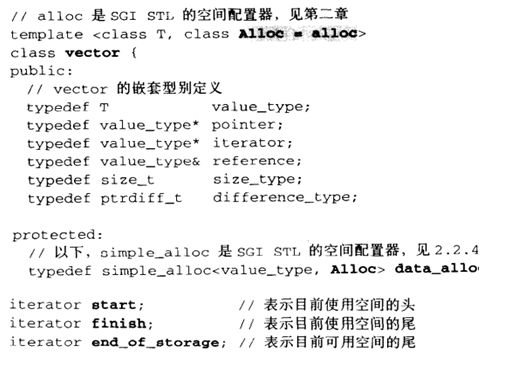
iterator insert(iterator \_\_position, const \_Tp& \_\_x);

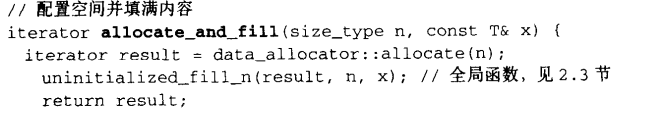
//删除某个位置上的元素

iterator erase(iterator \_\_position)；

//删除[first , last )区间内的所有元素

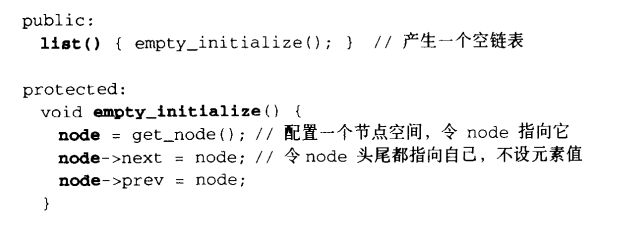
iterator erase(iterator \_\_first, iterator \_\_last)；





# list

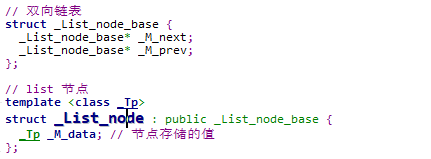
1. #include <list>,但SGI STL将list实现于更底层的<stl\_list.h>
2. SGI List是一个环形双向链表，链式结构
3. list迭代器：Bidirectional Iterator（双向迭代器）;
4. list不能够像vector一样以普通指针作为迭代器，因为其节点不保证在存储空间中连续存在；list迭代器必须具有指向List节点的能力，并有能力进行++ / -- / &等操作；
5. list的空节点



1. 查看list容器源码4步：

* 配置空间；
* 构造对象；
* 析构对象；
* 归还空间；

## List节点



## List数据结构

// 默认使用

template <class \_Tp, class \_Alloc>

class \_List\_base

{

protected:

\_List\_node<\_Tp>\* \_M\_node; // 只要一个指针，便可表示整个环状双向链表，空白节点

};

## List迭代器

struct \_List\_iterator

{

struct \_List\_iterator : public \_List\_iterator\_base

{

typedef \_List\_iterator<\_Tp , \_Tp& , \_Tp\*> iterator;

typedef \_List\_iterator<\_Tp , const \_Tp& , const \_Tp\*> const\_iterator;

typedef \_List\_iterator<\_Tp , \_Ref , \_Ptr> \_Self;

typedef \_Tp value\_type;

typedef \_Ptr pointer;

typedef \_Ref reference;

typedef \_List\_node<\_Tp> \_Node;

//\_List\_iterator\_base

typedef size\_t size\_type;

typedef ptrdiff\_t difference\_type;

typedef bidirectional\_iterator\_tag iterator\_category; // 双向移动迭代器

\_List\_node\_base\* \_M\_node; // 迭代器内部当然要有一个普通指针，指向 list 的节点

\_List\_iterator(\_Node\* \_\_x) : \_List\_iterator\_base(\_\_x) {}

\_List\_iterator() {}

\_List\_iterator(const iterator& \_\_x) : \_List\_iterator\_base(\_\_x.\_M\_node) {}

// 重载 \*，返回链表节点的值

reference operator\*() const ;

pointer operator->() const;

// 迭代器先前进一个节点

\_Self& operator++();

// 迭代器后增

\_Self operator++(int) ;

// 迭代器先后退一个节点

\_Self& operator--() ;

// 迭代器后减

\_Self operator--(int) ;

};

## List构造与内存管理

双向环形链表：

\_List\_base(const allocator\_type&)

{

\_M\_node = this->\_M\_get\_node();

\_M\_node->\_M\_next = \_M\_node;

\_M\_node->\_M\_prev = \_M\_node;

}

* List配置空间：

list::list() : 构造时，调用空间配置器构造一个空节点；

insert() : 插入时，调用\_M\_create\_node()，配置空间并构造对象给其赋值；

* List构造对象：

insert() : 插入时，调用\_M\_create\_node()，配置空间并构造对象给其赋值；

## List元素操作

list::push\_front()

list::push\_back()

list::erase()

list::pop\_front()

list::pop\_back()

list::clear()

list::remove()

list::unique()

list::splice()

list::merge()

list::reverse()

list::sort()

# deque

1. #include <deque>,但SGI STL将deque实现于更底层的<stl\_deque.h>
2. vector 是单向开口的连续线性空间，deque是一种双向开口的连续线性空间；
3. 双向开口:可以在头尾两端分别做元素插入和删除操作；vector从技术上讲，也可以进行头尾两端操作，但其头部操作，效率奇差；
4. deque和vector差异：
   1. deque允许常数时间内对头端进行元素的插入或移除操作；
   2. deque没有容量（capacity）观念，因为它是动态的以分段连续空间组合而成，随时可以增加一段新的空间并链接起来。deque不会像vector那样：“因旧空间不足而重新配置一块更大的空间，然后复制元素到新空间，在释放就空间”；
5. deque也提供RamdonAccess Iterator，但并不是普通指针; 因此，deque复杂度比vector更复杂，当然影响了各方面的运算，因此：除非必要，尽可能选择vector而非deque；
6. deque中控器:
7. 分段连续空间，就必须有中央控制，而为了整体连续的假象，数据结构及迭代器设计颇为复杂；
8. deque采用一块“map”作为主控，map是一小块连续空间，其中每个元素（称为一个节点，node）都是指针，指向另一段（较大的）连续线性空间，称为缓冲区；缓冲区才是deque的存储空间主体；

## deque中控器

\_Tp\*\* \_M\_map; //中控器,map是一小块连续空间，其内的每个元素都是一个指针（节点），指向一个缓冲区

## deque数据结构

inline size\_t \_\_deque\_buf\_size(size\_t \_\_size) ; //确定缓冲区大小；

template <class \_Tp, class \_Alloc>

class \_Deque\_base

{

protected:

typedef simple\_alloc<\_Tp, \_Alloc> \_Node\_alloc\_type; // 专属的空间配置器，每次配置一个元素大小

typedef simple\_alloc<\_Tp\*, \_Alloc> \_Map\_alloc\_type;

\_Tp\*\* \_M\_map; //中控器,map是一小块连续空间，其内的每个元素都是一个指针（节点），指向一个缓冲区

size\_t \_M\_map\_size; //map内可以容纳多个指针；当map节点不足时，可以 重新配置一块更大的map

iterator \_M\_start; //指向第一个缓冲区的第一个元素；

iterator \_M\_finish; //指向最后一个缓冲区的最后一个元素

};

template <class \_Tp, class \_Alloc = \_\_STL\_DEFAULT\_ALLOCATOR(\_Tp) >

class deque : protected \_Deque\_base<\_Tp, \_Alloc>

{

};

## deque迭代器

deque是整体连续空间，维持整体连续的假象，落在了迭代器operator++ , operator—身上；

迭代器设计方案：

* 1. 指出缓冲区在哪？
  2. 判断自己是否处于缓冲区边缘？是，一旦前进或后退跳入下一个缓冲区；
  3. 如何保证正确跳跃？deque必须掌控管控中心（map）

struct \_Deque\_iterator

{

typedef \_Deque\_iterator<\_Tp, \_Tp&, \_Tp\*> iterator;

typedef \_Deque\_iterator<\_Tp, const \_Tp&, const \_Tp\*> const\_iterator;

typedef random\_access\_iterator\_tag iterator\_category; // Random access iterator

typedef \_Tp value\_type;

typedef \_Ptr pointer;

typedef \_Ref reference;

typedef size\_t size\_type;

typedef ptrdiff\_t difference\_type;

typedef \_Tp\*\* \_Map\_pointer;

typedef \_Deque\_iterator \_Self;

\_Tp\* \_M\_cur; // 迭代器指向缓冲区的当前元素

\_Tp\* \_M\_first; // 迭代器指向缓冲区的头部

\_Tp\* \_M\_last; // 迭代器指向缓冲区的尾部

\_Map\_pointer \_M\_node; // 迭代器指向 map 的 node （保证缓冲区可正确跳跃）

}

## deque构造与内存管理

map节点不足时，扩充map的规则？？？？

template <class \_Tp, class \_Alloc>

class \_Deque\_base

{

public:

\_Deque\_base( const allocator\_type&, size\_t \_\_num\_elements )

: \_M\_map(0) ,

\_M\_map\_size(0) ,

\_M\_start() ,

\_M\_finish()

{

\_M\_initialize\_map( \_\_num\_elements );

}

protected:

/\*\*

\* \_M\_initialize\_map map 主控初始化

\* 作用：

\* 1.为map，分配内存

\* 2.为缓冲区，分配内存

\* 3.为迭代器，找到正确位置

\*/

void \_Deque\_base<\_Tp,\_Alloc>::\_M\_initialize\_map( size\_t \_\_num\_elements )

}

template <class \_Tp, class \_Alloc = \_\_STL\_DEFAULT\_ALLOCATOR(\_Tp) >

class deque : protected \_Deque\_base<\_Tp, \_Alloc>

{

explicit deque ( const allocator\_type& \_\_a = allocator\_type() ) : \_Base(\_\_a, 0) { }

deque ( size\_type \_\_n ,

const value\_type& \_\_value,

const allocator\_type& \_\_a = allocator\_type() )

: \_Base(\_\_a, \_\_n)

{

\_M\_fill\_initialize(\_\_value);

}

}

## deque元素操作

# stack

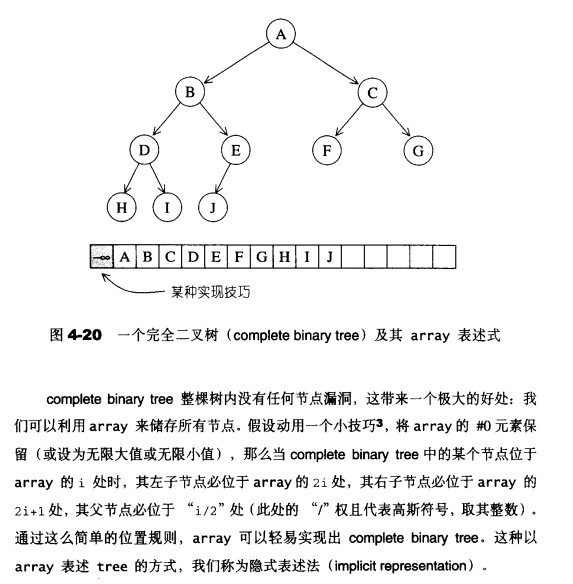
1. #include <stack>,但SGI STL将stack实现于更底层的<stl\_stack.h>
2. stack是一种“先进后出”的数据结构
3. stack底层默认使用deque作为容器；
4. stack是deque的一种配接器（修改其接口，使其形成另一种风貌，称为配接器）

# queue

1. #include <queue>,但SGI STL将queue实现于更底层的<stl\_queue.h>
2. queue是一种“先进先出”的数据结构；
3. queue底层默认使用deque作为容器；
4. queue是deque的一种配接器（修改其接口，使其形成另一种风貌，称为配接器）

# heap

1. #include <heap>,但将heap实现于更底层的<stl\_heap.h>
2. heap不归属于STL容器组件，扮演priority queue 的助手。顾名思义，priority queue 允许用户以任何次序将任何元素放入容器内；但取出时一定是优先权最高（也就是数值最高）的元素开始取；priority queue底层机制：binary max heap。
3. list和binary search tree 作为 priority queue的底层实现分析；
4. binary heap就是一种完全二叉树，也就是整棵树binary tree除了最底层的叶节点（s）之外，都是是填满的；而最底层的叶子节点，由左至右由不得有空隙；



1. 根据元素排列方式，heap可分为max-heap 和 min-heap两种，
2. max-heap 每个节点的key >= 其子节点的key ;
3. min-heap 每个节点的key <= 其子节点的key;

因此：max-heap 最大值为根节点，总位于array或vector的起头处；

min-heap 最小值为根节点，总位于array或vector的起头处；

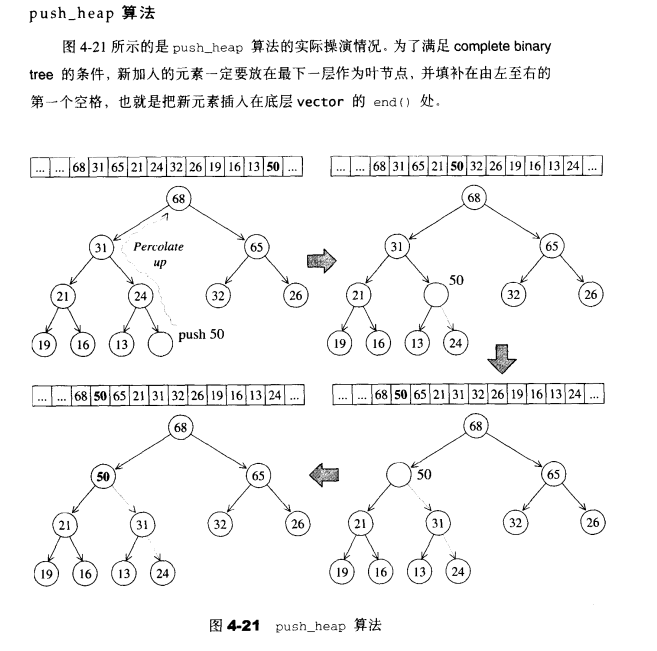
STL供应的为max-heap ,下面的heap都为max-heap;

1. heap的所有元素都必须遵循特别的（complete binary tree）排列规则，所以heap不提供遍历功能，也不提供迭代器；
2. 查看heap规则：
3. 先查看make\_heap；
4. 再查看push\_heap/pop\_heap/sort\_heap；

## push\_heap算法

插入新值时：

* + 1. 将新加入的元素一定要放在最下一层作为叶节点，并填补在由左至右的第一个空格，也就是把新元素插入在底层vector的end()处。
    2. 将新节点拿来与其父节点比较，如果其键值比父节点大，就父子对换位置。如此一直执行，直到不需要对换或直到根节点为止；

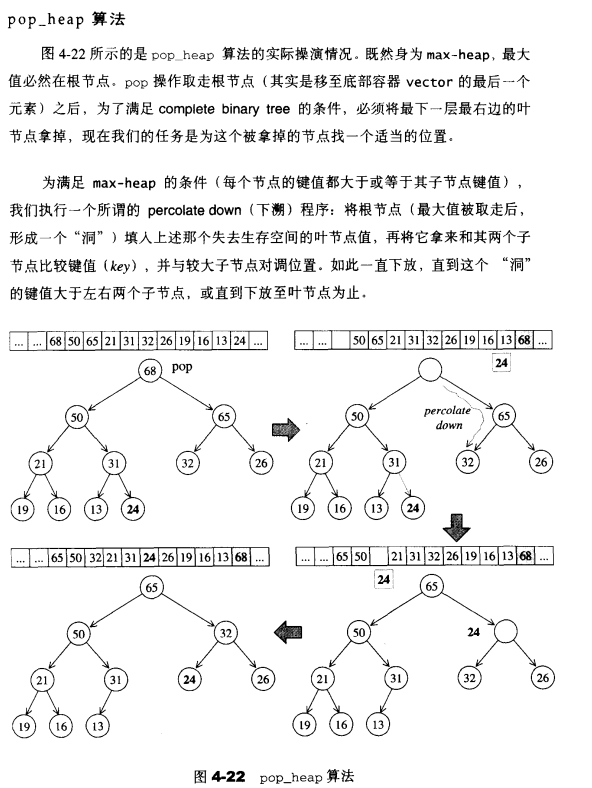


## pop\_heap算法

取出新值时：

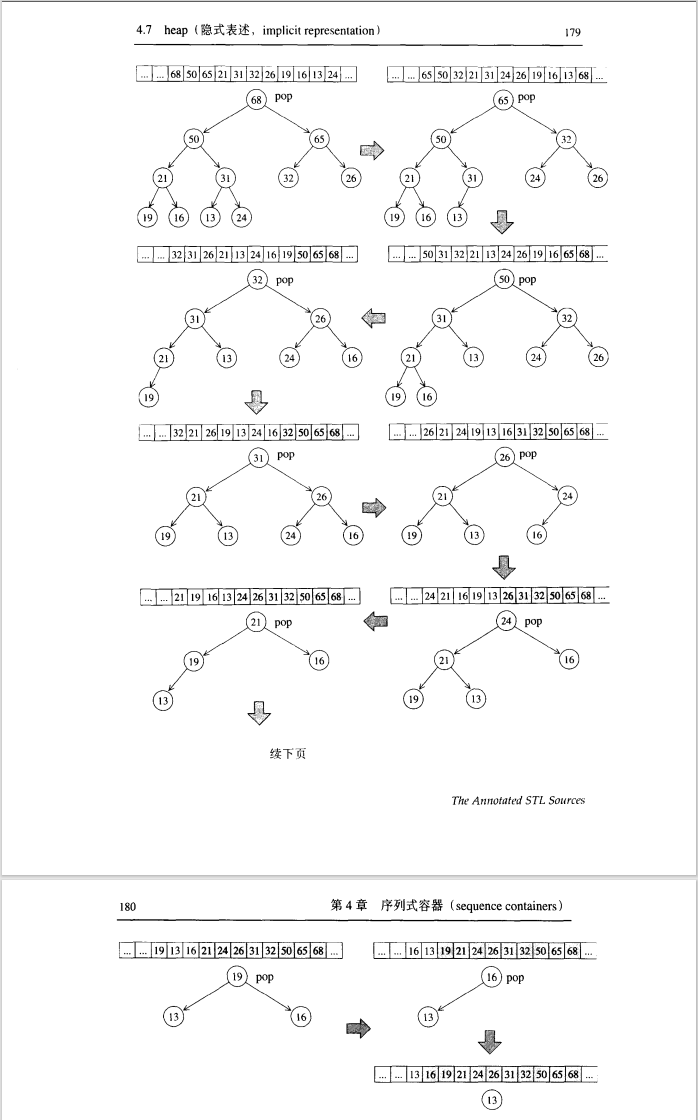
1. pop取走根节点（ 其实是移至底部容器vector的end位置 ）之后，为了满足complete binary tree的条件，必须将最下一层，最右边的叶节点拿掉；
2. 此时的任务：为这个被拿掉的节点找一个适当的位置
3. 将“vector的end节点” 与 “根节点” 交换位置，然后将它（“vector的end节点”）拿来和其两个子节点比较键值，并于较大子节点对调位置，如此一直下放，直到它（“vector的end节点”）大于左右两个子节点，或直到下放至叶节点为止；

注意：pop\_heap之后，最大元素只是被放置于底部容器（vector）最尾端，尚未被取走。取出：使用back(),移除:使用pop\_back();



## sort\_heap算法

1. sort\_heap算法：依赖于pop\_heap算法；
2. pop\_heap可获得heap中键值最大的元素，如果持续对整个heap做pop\_heap操作，每次将操作范围“从后向前”缩减一个元素 （因此，pop\_heap会把键值最大的元素放在底部容器的最尾端），当整个程序执行完毕时，我们便有了一个递增序列；
3. 排序过后，heap就不再是一个“合法的heap”了；



## make\_heap算法

1. 该算法用来将一组现有的数据转化为一个heap。其主要依据就是complete binary tree的**隐式表述**；

# priority\_queue

1. #include <queue>,但将heap实现于更底层的<stl\_queue.h>
2. Priority\_queue底层使用vector（heap的处理规则）；
3. Priority\_queue 是一个拥有权值（类似于优先级）观念的queue；
4. Priority\_queue内的元素，并非依照被推入的次序排列，而是自动依照元素的权值排列（通常权值以实值（数的大小）表示。权值最高者，排在最前面）；
5. 默认情况下，priority\_queue利用一个max-heap完成（complete binary tree）；max-heap可以满足priority\_queue所需要的“依权值高低自动递增排序”的特性；
6. 无迭代器，priority\_queue的所有元素，进出都有一定规则，只有queue顶端的元素（权值最高），才有机会被外界取用，priority\_queue不提供遍历功能，所以无迭代器；

# slist

1. #include <slist>，但将slist实现于更底层的<stl\_slist.h>
2. slist是一个单向链表；
3. slist与list的区别：
4. slist 是单向链表，list是双向链表;
5. slist 的迭代器是属于单向的Forward Iterator ;

list 的迭代器是属于双向的Bidirectional Iterator ;