C++ STL简介

容器(vector, deque, list, set, map)

配接器(stack, queue)

迭代器(iterator)

内存管理器(allocator)

函数对象(仿函数)(funtional)

数值运算(numeric)

算法(algorithm)

STL的版本

- **HP STL是所有其它STL实现版本的鼻祖。它是STL之父Alexander Stepanov在惠普的Palo Alto实验室工作时，和Meng Lee共同完成的。**
- **SGI STL http://www.sgi.com，是影响力很大的一个STL版本。**
- **STLport http://www.stlport.org，最初源于俄国人Boris Fomitchev的一个开发项目，主要用于将SGI STL的基本代码移植到其他主流编译器上，对SGI STL做了一些改进。**
- **Rouge Wave STL http://www.rougewave.com，Borland C++，从Builder 6.0**开始被上述版本取代。
- **P.J. Plauger STL http://www.dinkumware.com，VC++使用的STL。**

STL值得深入研究吗？

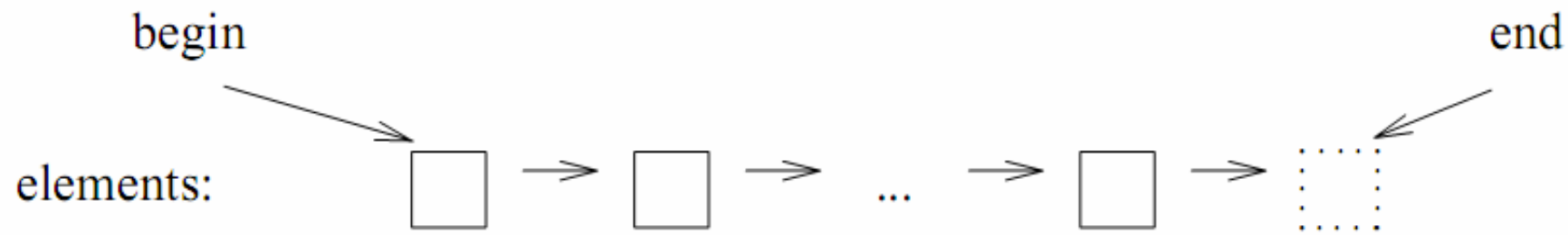
- …我的确认为 99.99 % 的程序员所写的程序，在 SGI STL 面前都是三流水准…

--侯俊杰

- 建议：如果你觉得你的水平远远胜过侯俊杰的话，或者没有多少空闲时间的话，或者对C++编程丝毫没有兴趣的话，大可不必研究任何版本的STL，否则可能看看为好，毕竟STL也就3万行代码，最好也顺便看看boost，后者可就大多了，大约68万行代码，而且还在膨胀过程中。
- 研究STL需要一定的算法和数据结构基础和扎实的C++基本功。
- 研究boost还需要一定的通信协议、线程、随机数产生等基础知识。

”超尾” 和 ”迭代”

超尾：



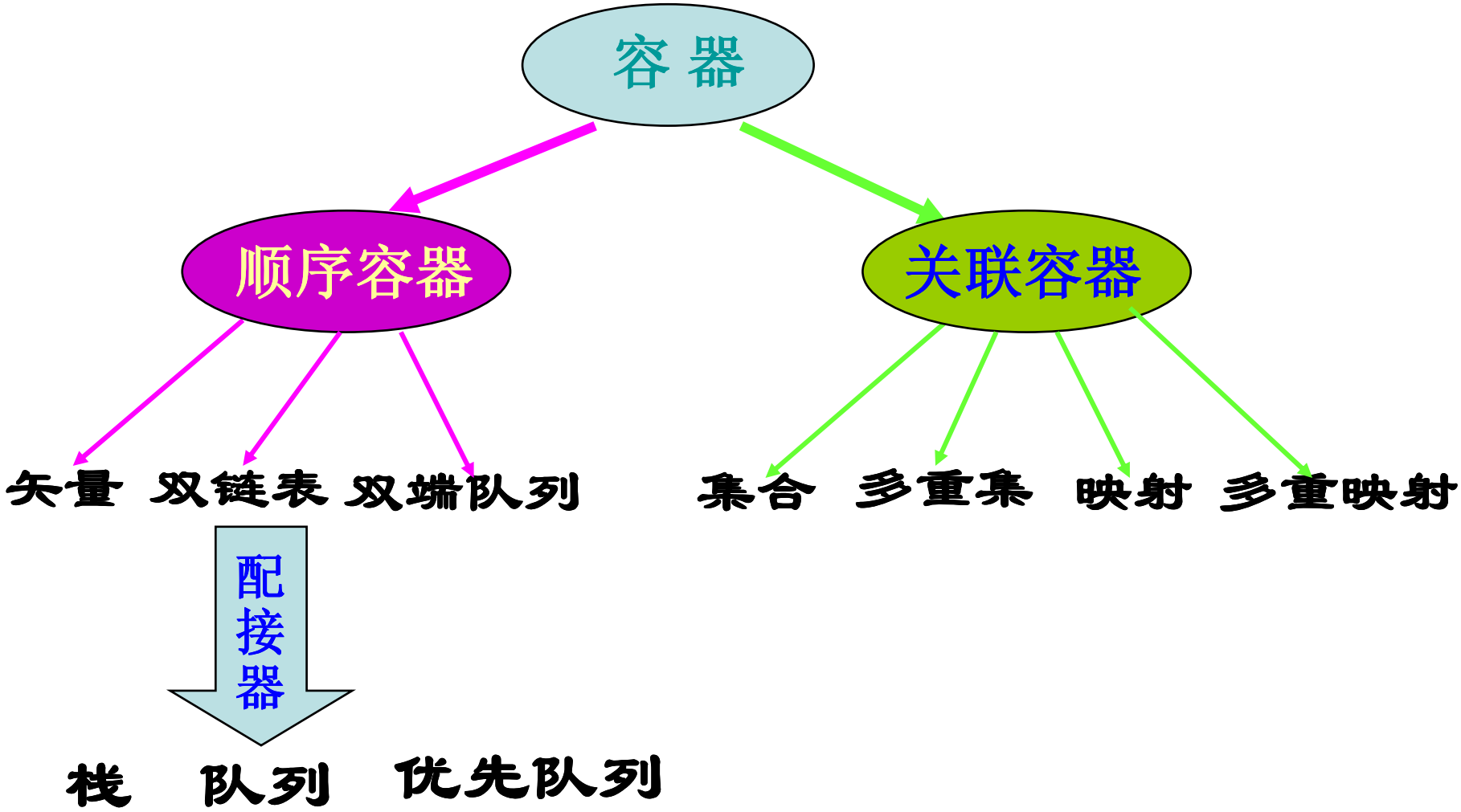
迭代：通俗的说就是++和--，在C语言中，指针前进和后退一步用++和--，它们只能用于数组。在C++ STL中，任何容器和部分算法均支持广义指针(迭代器)的++和--操作，于是就有人发明了个特别的名字“迭代”。

容 器

概念和术语

- 容器类是容纳、包含一组元素对象或元素集合的类。
- 基本容器：
  - 矢量(vector)、双端队列(deque)、双向循环链表(list)、集合(set)、多重集合(multiset)、映射(map)和多重映射(multimap)
- 注意：容器可以为空，这点不像基本数组。

容器的分类



顺 序 容 器

容器

- 顺序容器：其中的元素被视为逻辑上线性排列的，有头有尾，有前导有后继，可以用索引值即位置值来随机地访问其中任意一个元素。
- STL 提供三个基本顺序容器：
  - vector　list　deque
- STL 还提供三个容器配接器：
  - stack　queue　priority_queue
- 此外，STL 还提供四个“近容器”(类似容器但没有容器的全部功能)：
  - string　bitset　valarray　数组

顺序容器的常用接口

概念和术语

- 迭代起始点和终点
  - **begin()，end()，rbegin()，rend()**
- 插入方法
  - **push_front()，push_back()，insert()，**
- 迭代访问方法
  - 使用本身内建的迭代器
- 删除方法
  - **pop_front()，pop_back () ，erase()，clear()**
- 其它访问方法
  - **front()，back()，**下标**[ ]**运算符

vector 之 123

数据结构：vector是一个动态数组。

增长速度：它的增长速度是线性的，常数因子一般为2~3，可见它的增长速度很快。

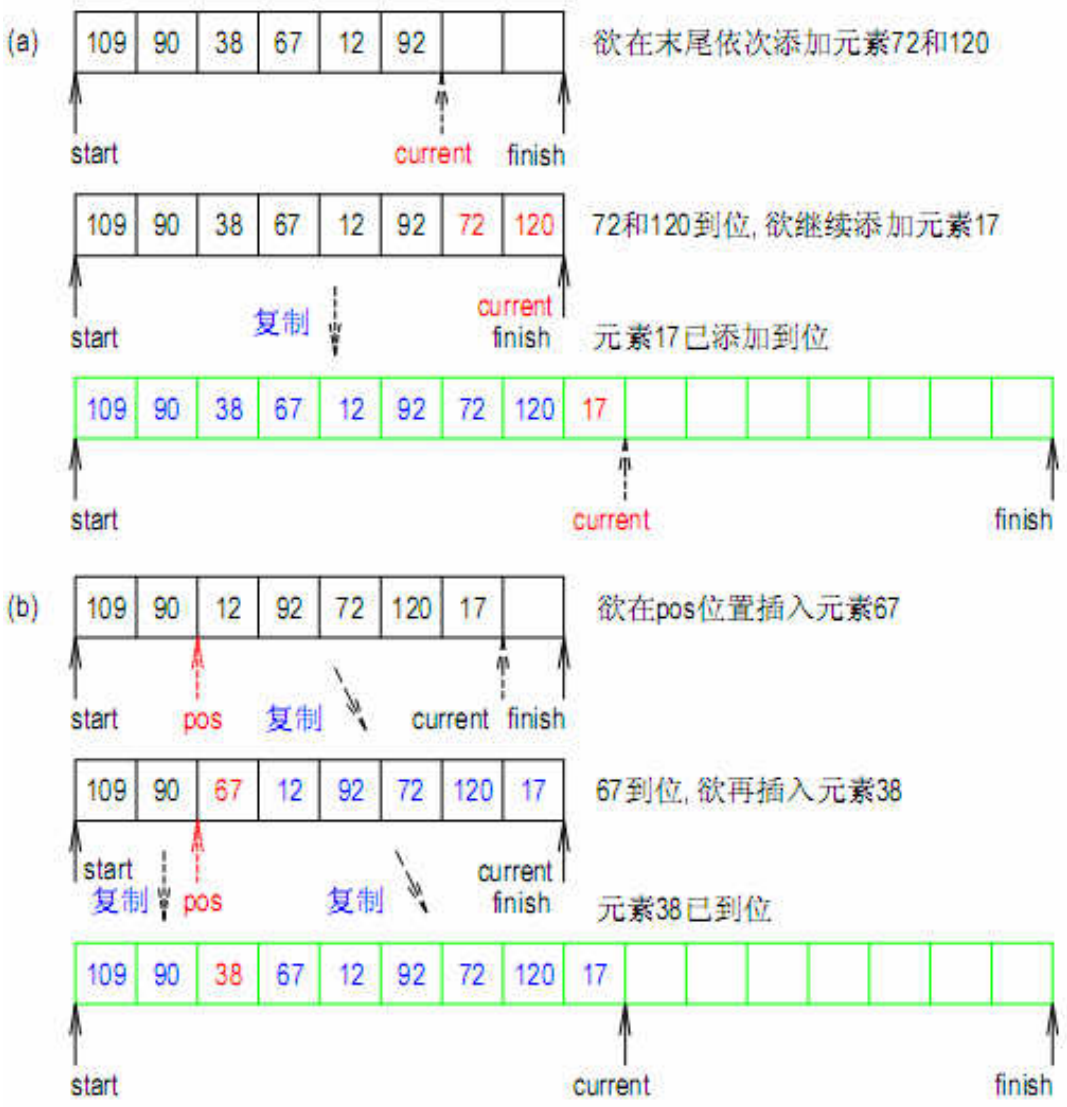
删除元素：删除元素时容量(capacity)不会自动缩小，尽管元素个数size会自动调整。

速度优化：如果能事先估计出元素个数可以使用reserve预留空间。从而可以避免容量膨胀过程中复制旧元素的时间开销。

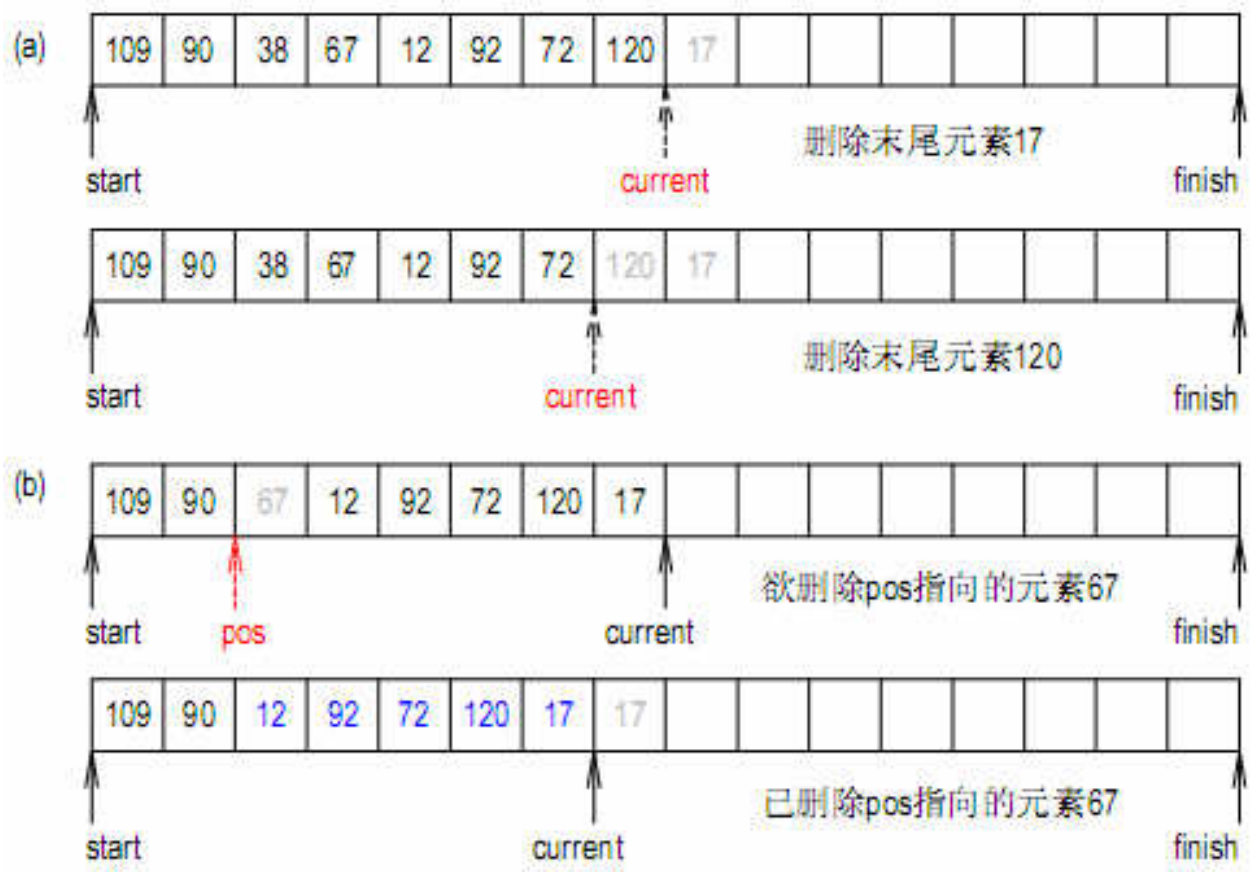
空间优化：当容量capacity远远超过size，可以减肥，减肥的时间复杂度为O(n)。

迭代器：vector内建的迭代器是随机迭代器。

vector的插入元素过程



vector的删除元素过程



vector常用的操作

push_back在尾部添加新元素，时间复杂度O(1)

[]用下标访问某个元素，时间复杂度O(1)

empty判断是否空，时间复杂度O(1)

size求当前元素个数，时间复杂度O(1)

reserve预留空间，时间复杂度依赖于何时调用O(1)或O(n)

insert插入一个元素，一般而言时间复杂度O(n)

erase删除一个元素，一般而言时间复杂度为O(n)

clear清空所有元素，时间复杂度O(1)

swap交换两个vector类型的对象，时间复杂度O(1)

// …

禁忌操作!

用指针指向某个位置(非begin非end)，vector的对象经过一系列插入(和删除)操作(包括push_back和insert)后，解析这个指针。

例如：

```
std::vector<int> v;

// 经过一系列的push_back后

std::vector<int>::iterator it = v.begin() + (v.size()>>1);

// 又经过一系列的push_back后

*it = 10;        // 这可能导致程序崩溃或结果莫名其妙
```

怎么给vector对象减肥?

例如：

```
std::vector<int> v;
// 经过一些列的插入和删除操作后v.capacity()远远大于v.size()
// 如果将来不会再添加新元素到v中，则可以考虑减肥节约内存
if(v.size() + (v.size() >> 1) < v.capacity()) // 至少80%空间利用率
{
  std::vector<int> tmp(v.begin(), v.end());//复制现有元素
  tmp.swap(v);
}
```

deque 之 123

容器

- 读音：同deck。
- 物理结构：分段数组。
- 逻辑结构：双端队列(两端都可以快速插入和删除)。
- 一般应用：常用来配接成动态栈或普通队列，很少单独使用。
- 速度优化和空间优化：自动优化，无需干预。
- 常数操作：[ ]以及任何发生在一端的操作均是常数。
- 插入和删除：发生在非两端的插入和删除操作很慢，时间复杂度均O(n)。
- 下标访问：尽管用下标[ ]访问某元素是常数，但比用下标[ ]访问vector中的元素速度慢一点。
- 迭代器：deque内建的迭代器是随机迭代器。

list 之 123

容器

- 数据结构：双向循环链表。
- 常数操作：在任何指定位置添加、删除或解析某元素。
- 时间和空间优化：无需干预，也没有办法干预。
- 特殊操作：排序。
- 空表：空表有一个头结点，但size()为0。
- 禁忌：不要用0 == l.size()判断list对象l是否空，而应该用l.empty()
- 插入和删除：仅仅影响局部，非常安全(对比一下发生在vector上的插入和删除操作会扰乱指向任何元素的所有指针)。
- 留神之处：删除一个元素时迭代器指向的位置。
- 迭代器：内建双向迭代器(不支持用下标[ ]访问某个元素)

顺序容器的例子 -- list之编码设计

```
1   // dlist.hpp
2   #ifndef DLIST_HPP_
3   #define DLIST_HPP_
4
5   #include <cstddef>          // size_t
6   #include "compare.hpp"      // 同svector实现部分的compare.hpp文件
7
8   template <typename T>
9   struct dnode                // 双向循环链表的节点
10  {
11    dnode* prev, * next;
12    T data;
13    dnode(const T& t = T()) : data(t) {}
14  };  // dnode
15
16  template <typename T>
17  class dlist_base            // dlist的基类
18  {
19  protected:
20    dnode<T>* head;                                          // 链表表头节点
21    dlist_base() : head(new dnode<T>) { head->prev = head->next = head; } // 构造头结点
22    virtual ~dlist_base()                                    // 销毁整个链表, O(n)
23    {
24      clear();
25      delete head;
26    }
27    void clear();
28  };  // dlist_base
```

```
30  template <typename T>
31  inline void dlist_base<T>::clear()                    // 清空除表头外的所有节点，O(n)
32  {
33    dnode<T>* cur = head->next;
34    while(cur != head)
35    {
36      dnode<T>* tmp = cur;
37      cur = cur->next;
38      delete tmp;
39    }
40    head->prev = head->next = head;                     // 指针回指，构造一个环
41  }
42
43  template<typename T, typename R, typename P>
44  struct dlist_iterator       // dlist的双向迭代器
45  {
46    typedef R reference;
47    typedef P pointer;
48    typedef dlist_iterator<T, R, P> self;
49    typedef dlist_iterator<T, T&, T*> iterator;
50    typedef dlist_iterator<T, const T&, const T*> const_iterator;
51
52    dnode<T>* node;
53
54    dlist_iterator() {}
55    dlist_iterator(dnode<T>* x) : node(x) {}
56    dlist_iterator(const iterator& x) : node(x.node) {}
57    dlist_iterator(const const_iterator& x) : node(x.node) {}
58
59    reference operator *() const { return node->data; }
60    pointer operator ->() const { return &**this; }
```

```
61
62      self& operator++()                                  // 假设p指向某节点，模拟++p;
63      {
64        node = node->next;
65        return *this;
66      }
67      self operator++(int)                                // 假设p指向某节点，模拟p++;
68      {
69        self tmp = *this;
70        node = node->next;
71        return tmp;
72      }
73      self& operator--()                                  // 假设p指向某节点，模拟--p;
74      {
75        node = node->prev;
76        return *this;
77      }
78      self operator--(int)                                // 假设p指向某节点，模拟p--;
79      {
80        self tmp = *this;
81        node = node->prev;
82        return tmp;
83      }
84                                                          // 两迭代器是否相等?
85      bool operator ==(const dlist_iterator& x) const { return node == x.node; }
86      bool operator !=(const dlist_iterator& x) const { return node != x.node; }
87    };  // dlist_iterator
88
```

```
89  template <typename T>
90  class dlist : protected dlist_base<T>                    // 带有头结点的双向循环链表
91  {
92  public:
93    typedef T value_type;
94    typedef value_type* pointer;
95    typedef const value_type* const_pointer;
96    typedef value_type& reference;
97    typedef const value_type& const_reference;
98    typedef std::size_t size_type;
99    typedef typename dlist_iterator<value_type, reference, pointer>::iterator iterator;
100   typedef typename dlist_iterator<value_type, const_reference, const_pointer>
101                    ::const_iterator const_iterator;
102 protected:
103   typedef dlist_base<T> base;
104   using base::head;                                       // 使head在dlist中可见
105   void transfer(iterator, iterator, iterator);            // 转移一个序列到新位置, O(1)
106 public:
107   iterator begin() { return iterator(head->next); }       // 指向第一个节点, O(1)
108   const_iterator begin() const { return const_iterator(head->next); }
109   iterator end() { return iterator(head); }               // 指向头结点, O(1)
110   const_iterator end() const { return const_iterator(head); }
111   bool empty() const { return head->next == head; }       // 链表是否为空? O(1)
112   size_type size() const;                                 // 表中有多少个元素? O(n)
113   size_type max_size() const { return size_type(-1); }    // 理论上能存放多少元素? O(1)
114   reference front() { return *begin(); }                  // 第一个元素, O(1)
115   const_reference front() const { return *begin(); }
116   reference back() { return *(--end()); }                 // 最后一个元素, O(1)
117   const reference back() const { return *(--end()); }
```

```
118     void swap(dlist<T>&);                                          // 交换两个链表，O(1)
119     iterator insert(iterator, const_reference x = T());            // 插入一个元素，O(1)
120     void insert(iterator, const_pointer, const_pointer);           // 插入一个元素序列
121     void insert(iterator, const_iterator, const_iterator);         // 插入一个元素序列
122     void insert(iterator, size_type, const_reference);             // 插入多个元素
123     void push_front(const_reference x) { insert(begin(), x); }// 在表头后插入一个元素，O(1)
124     void push_back(const_reference x) { insert(end(), x); }  // 插入一个元素到链表尾，O(1)
125     iterator erase(iterator);                                      // 删除指定节点，O(1)
126     iterator erase(iterator, iterator);                            // 删除一个序列
127     void clear() { base::clear(); }                                // 清空dlist，O(n)
128     void resize(size_type new_sz, const_reference x = T());  // 重新设定元素个数，O(n)
129     void pop_front() { erase(begin()); }                           // 删除第一个节点，O(1)
130     void pop_back() { erase(--end()); }                            // 删除最后一个节点，O(1)
131     dlist() : base() {}
132     dlist(size_type n, const_reference value) : base() { insert(begin(), n, value); }
133     explicit dlist(size_type n) : base() { insert(begin(), n, T()); }
134
135     template <typename InputIterator>
136     dlist(InputIterator first, InputIterator last) : base()
137     { insert(begin(), first, last); }
138
139     dlist(const dlist<T>& x) : base() { insert(begin(), x.begin(), x.end()); }
140     dlist<T>& operator =(const dlist<T>&);                         // 复制一个dlist，O(n)
141
```

```
142     void assign(size_type, const_reference);                    // 重新构造dlist, O(n)
143     void assign(iterator, iterator);
144     void splice(iterator, dlist&);                              // 嫁接整个链表, O(1)
145     void splice(iterator, dlist&, iterator);                    // 嫁接一个节点, O(1)
146     void splice(iterator, dlist&, iterator, iterator);          // 嫁接一个序列, O(1)
147
148     void remove(const_reference);                               // 删除某元素, O(n)
149     void unique();                                              // 删除相邻的重复元素, O(n)
150     void merge(dlist&);                                         // 合并两个有序链表, O(n)
151     void reverse();                                             // 反向链表所有节点, O(n)
152     void sort();                                                // 归并排序整个链表, O(nlogn)
153
154     template <typename Predicate> void remove_if(Predicate);
155     template <typename BinaryPredicate> void unique(BinaryPredicate);
156     template <typename StrictWeakOrdering> void merge(dlist&, StrictWeakOrdering);
157     template <typename StrictWeakOrdering> void sort(StrictWeakOrdering);
158 };  // dlist
159
160 template <typename T>       // 把[first, last)区间的所有节点转移到pos前, O(1)
161 inline void dlist<T>::transfer(iterator pos, iterator first, iterator last)
162 {
163   if(pos != last)
164   {
165     last.node->prev->next = pos.node;
166     first.node->prev->next = last.node;
167     pos.node->prev->next = first.node;
168     dnode<T>* tmp = pos.node->prev;
169     pos.node->prev = last.node->prev;
170     last.node->prev = first.node->prev;
171     first.node->prev = tmp;
172   }
173 }
```

```
174
175   template <typename T>      // 表中有多少个元素? O(n)
176   inline typename dlist<T>::size_type dlist<T>::size() const
177   {
178     size_type sz = 0;
179     for(const_iterator it = begin(); it != end(); ++it)
180       ++sz;
181     return sz;
182   }
183
184   template <typename T>      // 交换两个链表, O(1)
185   inline void dlist<T>::swap(dlist<T>& x)
186   {
187     dnode<T>* tmp = head;
188     head = x.head;
189     x.head = tmp;
190   }
191
192   template <typename T>      // 在pos前插入一个元素, O(1)
193   inline typename dlist<T>::iterator dlist<T>::insert(iterator pos, const_reference x)
194   {
195     dnode<T>* tmp = new dnode<T>(x);
196     tmp->next = pos.node;
197     tmp->prev = pos.node->prev;
198     pos.node->prev->next = tmp;
199     pos.node->prev = tmp;
200     return tmp;
201   }
202
203   template <typename T>      // 插入[first, last)区间的所有元素到pos前
204   void dlist<T>::insert(iterator pos, const_pointer first, const_pointer last)
205   {
206     for(; first != last; ++first)
207       insert(pos, *first);
208   }
```

```
209
210   template <typename T>       // 插入[first, last)区间的所有元素到pos前
211   void dlist<T>::insert(iterator pos, const_iterator first, const_iterator last)
212   {
213     for(; first != last; ++first)
214       insert(pos, *first);
215   }
216
217   template <typename T>       // 插入k个元素到pos前，时间复杂度O(k)
218   void dlist<T>::insert(iterator pos, size_type k, const_reference x)
219   {
220     for(; k > 0; --k)
221       insert(pos, x);
222   }
223
224   template <typename T>       // 嫁接整个链表到pos前，O(1)
225   inline void dlist<T>::splice(iterator pos, dlist& x)
226   {
227     if(!x.empty())
228       transfer(pos, x.begin(), x.end());
229   }
231   template <typename T>       // 嫁接一个节点到pos前，O(1)
232   inline void dlist<T>::splice(iterator pos, dlist&, iterator i)
233   {
234     iterator j = i;
235     ++j;
236     if(pos == i || pos == j)
237       return;
238     transfer(pos, i, j);
239   }
240
241   template <typename T>       // 嫁接[first, last)区间的所有节点到pos前，O(1)
242   inline void dlist<T>::splice(iterator pos, dlist&, iterator first, iterator last)
243   {
244     if(first != last)
245       transfer(pos, first, last);
246   }
```

```
248   template <typename T>       // 重设dlist大小, O(n)
249   void dlist<T>::resize(size_type new_sz, const_reference x)
250   {
251     iterator i = begin();
252     size_type len = 0;
253     for(; i != end() && len < new_sz; ++i, ++len);
254     if(len == new_sz)
255       erase(i, end());
256     else
257       insert(end(), new_sz - len, x);
258   }
259
260   template <typename T>       // 复制一个dlist, O(n)
261   dlist<T>& dlist<T>::operator =(const dlist<T>& x)
262   {
263     if(this != &x)
264     {
265       iterator first1 = begin();
266       iterator last1 = end();
267       const_iterator first2 = x.begin();
268       const_iterator last2 = x.end();
269       while(first1 != last1 && first2 != last2)
270         *first1++ = *first2++;
271       if(first2 == last2)
272         erase(first1, last1);
273       else
274         insert(last1, first2, last2);
275     }
276     return *this;
277   }
```

```
279   template <typename T>       // 用n个x重建dlist, O(n)
280   inline void dlist<T>::assign(size_type n, const_reference x)
281   {
282     iterator i = begin();
283     for(; i != end() && n > 0; ++i, --n)
284       *i = x;
285     if(n > 0)
286       insert(end(), n, x);
287     else
288       erase(i, end());
289   }
290
291   template <typename T>       // 用[first, last)区间的元素重建dlist, O(n)
292   inline void dlist<T>::assign(iterator first, iterator last)
293   {
294     const_iterator i = begin(), j = first;
295     for(; i != end() && j != last; ++i, ++j)
296       *i = *j;
297     if(i == end() && j != last)
298       insert(end(), j, last);
299     else
300       erase(i, end());
301   }
302
303   template <typename T>       // 删除pos指向的节点, O(1), 返回指向下一个节点的迭代器
304   inline typename dlist<T>::iterator dlist<T>::erase(iterator pos)
305   {
306     dnode<T>* next_node = pos.node->next;
307     dnode<T>* prev_node = pos.node->prev;
308     dnode<T>* n = pos.node;
309     prev_node->next = next_node;
310     next_node->prev = prev_node;
311     delete n;
312     return iterator(next_node);
313   }
```

```
315  template <typename T>       // 合并两个有序链表，需遍历每个节点以比较元素大小，O(n)
316  void dlist<T>::merge(dlist<T>& x)
317  {
318    iterator first1 = begin(), last1 = end();
319    iterator first2 = x.begin(), last2 = x.end();
320    while(first1 != last1 && first2 != last2)
321      if(*first2 < *first1)
322      {
323        iterator next = first2;
324        transfer(first1, first2, ++next);
325        first2 = next;
326      }
327      else
328        ++first1;
329    if(first2 != last2)
330      transfer(last1, first2, last2);
331  }
332
333  template <typename T>       // 翻转整个循环链表，需要遍历整个链表，O(n)
334  inline void dlist<T>::reverse()
335  {
336    dnode<T>* tmp = head;
337    do
338    {
339      dnode<T>* p = tmp->next;
340      tmp->next = tmp->prev;
341      tmp->prev = p;
342      tmp = tmp->prev;
343    } while(tmp != head);
344  }
345
```

```
346   template <typename T>        // 归并排序整个链表，时间复杂度O(nlogn)
347   void dlist<T>::sort()
348   {
349     if(head->next != head && head->next->next != head)
350     {
351       dlist<T> carry, counter[32];
352       size_type fill = 0;
353       while(!empty())
354       {
355         carry.splice(carry.begin(), *this, begin());
356         size_type i = 0;
357         while(i < fill && !counter[i].empty())
358         {
359           counter[i].merge(carry);
360           carry.swap(counter[i++]);
361         }
362         carry.swap(counter[i]);
363         if(i == fill) ++fill;
364       }
365
366       for(size_type i = 1; i < fill; ++i)
367         counter[i].merge(counter[i-1]);
368       swap(counter[fill-1]);
369     }
370   }
371
```

```
372   template <typename T>
373     template <typename Predicate>            // 删除符合谓词条件的元素，O(n)
374   void dlist<T>::remove_if(Predicate pred)
375   {
376     iterator first = begin(), last = end();
377     while(first != last)
378     {
379       iterator next = first;
380       ++next;
381       if(pred(*first))
382         erase(first);
383       first = next;
384     }
385   }
386
387   template <typename T>
388     template <typename BinaryPredicate>      // 相邻重复元素只保留第一个，O(n)
389   void dlist<T>::unique(BinaryPredicate binary_pred)
390   {
391     iterator first = begin(), last = end();
392     if(first == last) return;
393     for(iterator next = first; ++next != last; next = first)
394       if(binary_pred(*first, *next))
395         erase(next);
396       else
397         first = next;
398   }
```

```
400   template <typename T>
401     template <typename StrictWeakOrdering>  // 归并排序整个链表, O(nlogn)
402   void dlist<T>::sort(StrictWeakOrdering comp)
403   {
404     if(head->next != head && head->next->next != head)
405     {
406       dlist<T> carry;
407       dlist<T> counter[32];
408       size_type fill = 0;
409       while(!empty())
410       {
411         carry.splice(carry.begin(), *this, begin());
412         size_type i = 0;
413         while(i < fill && !counter[i].empty())
414         {
415           counter[i].merge(carry, comp);
416           carry.swap(counter[i++]);
417         }
418         carry.swap(counter[i]);
419         if(i == fill) ++fill;
420       }
421
422       for(size_type i = 1; i < fill; ++i)
423         counter[i].merge(counter[i-1], comp);
424       swap(counter[fill-1]);
425     }
426   }
```

```
428  template <typename T>
429    template <typename StrictWeakOrdering>  // 合并两个有序链表，O(n)
430  void dlist<T>::merge(dlist<T>& x, StrictWeakOrdering comp)
431  {
432    iterator first1 = begin(), last1 = end();
433    iterator first2 = x.begin(), last2 = x.end();
434    while(first1 != last1 && first2 != last2)
435      if(comp(*first2, *first1))
436      {
437        iterator next = first2;
438        transfer(first1, first2, ++next);
439        first2 = next;
440      }
441      else
442        ++first1;
443    if(first2 != last2) transfer(last1, first2, last2);
444  }
445
```

```
446  template <typename T>      // 两个链表相等吗？可能逐一比较各个元素，O(n)
447  inline bool operator ==(const dlist<T>& x, const dlist<T>& y)
448  {
449    typedef typename dlist<T>::const_iterator const_iterator;
450    const_iterator end1 = x.end(), end2 = y.end();
451    const_iterator i1 = x.begin(), i2 = y.begin();
452    while(i1 != end1 && i2 != end2 && *i1 == *i2)
453      ++i1, ++i2;
454    return i1 == end1 && i2 == end2;
455  }
```

```
457   template <typename T>       // 两个链表中前者较小吗？可能逐一比较各个元素，O(n)
458   inline bool operator <(const dlist<T>& x, const dlist<T>& y)
459   {
460     typedef typename dlist<T>::const_iterator const_iterator;
461     const_iterator end1 = x.end(), end2 = y.end();
462     const_iterator i1 = x.begin(), i2 = y.begin();
463     while(i1 != end1 && i2 != end2 && *i1 == *i2)
464       ++i1, ++i2;
465     return (i1 != end1 && i2 != end2 && *i1 < *i2) || (i1 == end1 && i2 != end2);
466   }
467
468   template <typename T>       // 交换两个链表，O(1)
469   inline void swap(dlist<T>& x, dlist<T>& y) { x.swap(y); }
470
471   #endif // DLIST_HPP_
472
```

顺序容器之思考

- 栈：vector, deque, list都可以配接成栈，应该用哪个更好？
- 队列： deque, list都可以配接成普通队列，应该用哪个更好？
- 当频繁的插入和删除操作发生在序列的中间某位置时，用vector, deque, list哪个更快？
- 用vector, deque, list存储同样多的元素，哪个更节约内存？
- 在序列尾部依次添加同样多的元素至vector, deque, list之中，哪个速度更快？

容器

关联容器之123

set, multiset, map, multimap的底层数据结构：
红黑树，可见存储在关联容器中的元素都是有序存储的。
速度代价：插入、删除、或查找任何一个元素的时间复杂度均为**O(logn)**。
内存代价：每插入一个元素需要的辅助空间是三个指针外加一个标记位，由于**padding**原因，通常用三个指针加一个整数的空间。
迭代器：内建双向迭代器。
元素数序存储的元素本身都有序。

配接器 之 123

配

接

器

- 配接器是一种接口类，是原有类的“二次封装”。其存在的理由就是改造别的类，改造的常用手段是组合。
  - 为已有的类保留适用的接口，去掉不适用的接口，提供新的接口。
  - 目的是简化、约束、使之安全、隐藏或改变被修改类提供的服务集合。
  - 狭义的配接器指stack, queue, priority_queue，通常它们都不能算做容器，因为它们没有迭代器。
  - 栈和队列通常用deque来组合而成， priority_queue则通常用vector和堆排序算法来实现。

例如，队列的定义：

```
template <class T, class C = deque<T> > class std::queue {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit queue(const C& a = C()) : c(a) { }

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    value_type& front() { return c.front(); }
    const value_type& front() const { return c.front(); }

    value_type& back() { return c.back(); }
    const value_type& back() const { return c.back(); }

    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};
```

什么是迭代器

- 迭代器是广义的指针，它们提供了访问容器或序列每个元素的方法。

迭代器

指针可以指向内存中的一个地址。

迭代器可以指向容器或序列中的一个位置。

各容器支持的迭代器种类

迭代器

- vector　随机迭代器
- deque　随机迭代器
- list　双向迭代器
- set　双向迭代器
- multiset　双向迭代器
- map　双向迭代器
- multimap　双向迭代器
- stack　没有迭代器
- queue　没有迭代器
- priority_queue　没有迭代器

标准C++库中的算法

- 大约70个算法，都用函数模板实现的。

算法

- 分类：

不可变序列算法（Non-mutating algorithms）

– 不直接修改所操作的容器内容的算法。

可变序列算法（Mutating algorithms）

– 可以修改它们所操作的容器的元素的算法。

典 型 算 法

算法

find 返回找到的首个元素
count 统计所含指定值的元素个数
equal 比较两容器元素，若全相等则 **true**
search 搜索一容器中与另一容器中相符的元素
copy 复制一序列的元素到其他位置
swap 交换两位置上的元素
fill 将某值填充到某位置
sort 按指定顺序将容器中的元素排序
partial_sort 部分排序，通常用于筛选
accumulate 返回指定范围内元素个数
for_each 对容器中的每个元素执行指定操作
includes 前一个排序序列是否包含后一个排序序列
nth_element 找第**nth**个元素
upper_bound 在一个排序序列中找某元素的上界
// ...

函 数 对 象

- 函数对象又称为仿函数，有几十个。
- 例如：

```
template <typename T>
struct greater
{
  bool operator()(const T& x, const T& y) const { return y < x; }
};
std::vector<int> v;
// 存入一些整数到v中，然后排序
std::sort(v.begin(), v.end(), greater<int>());
```

课后习题

- **1. 用vector和堆排序相关算法设计和编写优先队列的类。**
- **2. 单链表slist(见第二次课习题)改为模板实现，并添加前向迭代器。**
- **3. 容器map虽然插入和删除速度快，但是耗费内存较多，用vector，二分插入排序(自己编写)和upper_bound等函数组合设计一个类来模拟容器map的提供各成员函数。编程完毕后，测试find, insert, erase的速度跟std::map的相关成员函数对比一下，看看相差多少。**
- **4. 查阅文档，参考STLport设计编写一个hash_map并提供必要的接口。**

Thanks!