Effective STL 笔记

By Ken 2013-04-21

（一） 容器

- ✧ **条款 1：慎重选择容器类型**
  1) 容器类型
     - ■ 标准 STL 序列容器：vector、string、deque 和 list
     - ■ 标准 STL 关联容器：set、multiset、map 和 multimap
     - ■ 非标准容器：slist 和 rope
     - ■ 非标准的关联容器：hash_set、hash_multiset、hash_map 和 hash_multimap
     - ■ 标准的非 STL 容器：数组、bitset、valarray、stack、queue 和 priority_queue
     - ■ vector<char>作为 string 的替代
     - ■ vector 作为标准关联容器的替代
- ✧ **条款 2：不要试图编写独立于容器类型的代码**
  1) 不同类型的序列的容器迭代器、指针和引用无效的规则是不同的
- ✧ **条款 3：确保容器中的对象拷贝正确而高效**
  1) push_back/insert 等向容器中插入对象时，存入容器的是所指定对象的拷贝。next_permutation、previous_permutation、remove、unique、rotate 或 reverse 等都是以 copy 为基础的。copy 对象是 STL 的工作方式
- ✧ **条款 4：调用 empty 而不是检查 size()是否为 0**
  1) list 的 size()是线性时间复杂度。为什么如此设计？如果 size 是常数时间操作，那么 list 的每个成员函数都必须更新它们所操作的链表的大小，当然也包括 splice。可是 splice 更新它所改变的链表的大小的唯一方式是计算所链接的元素的个数，而这会使 splice 不具有你所期望的常数时间操作性能
- ✧ **条款 5：区间成员函数优先于与之对应的单元素成员函数**
  1) 区间成员函数更高效，减少了不必要的 reallocation 和 copy。

     vector v1<int>

     v.assign(v2.begin() + v2.size()/2, v2.end());

     versus

     v1.clear();

     copy(v2.begin() + v2.size()/2, back_inserter(v1));

     versus

     v1.insert(v1.end(), v2.begin() + v2.size()/2, v2.end());
  2) 太多的 STL 程序员滥用了 copy。通过利用插入迭代器的方式来限定目标区间的 copy 调用，几乎都应该被替换为对区间成员函数的调用
  3) 区间操作
     - ■ 区间创建：所有的标准容器都提供如下形式的构造函数

       container::container(InputIterator begin, InputIterator end);
     - ■ 区间插入：所有的标准序列容器都提供了如下形式的 insert

       void container::insert(iterator pos, InputIterator begin, InputIterator end);

关联容器利用比较函数来决定元素改插入何处，它们提供了省去 pos 参数的函数原型

void container::insert(InputIterator begin, InputIterator end);

- 区间删除：所有的标准容器都提供了区间形式的删除 erase 操作

  序列容器提供了这样的形式

  iterator container::erase(iterator begin, iterator end);

  关联容器则提供了如下形式

  void container::erase(iterator begin, iterator end);

- 区间赋值：所有的标准容器都提供了区间形式的 assign

  void container::assign(iterator begin, iterator end);

✧ **条款 6：当心 C++编译器最烦人的分析机制**

1) 把所有的 int 从文件中 copy 到 list 中

   ifstream dataFile("ints.dat");

   list<int> data(istream_iterator<int>(dataFile), istream_iterator<int>());

   编译器把上述表达式解释为一个函数申明：

   一个 data 函数，返回 list<int>，第一个参数是 istream_iterator<int>，第二个参数是一个函数指针，指向一个返回值为 istream_iterator<int>，不接收任何参数的函数。(编译器尽可能地解释为函数声明)

   int g(double (*pf)()) => int g(double pf ()) => int g(double ());

   注意围绕参数名的括号和独立括号的区别，围绕参数名的括号可以省略

   class Widget {…};

   Widget w(); 没有声明一个 w 的 Widget 对象，而是声明了一个名为 w 的函数，该函数不带任何参数，返回一个 Widget

2) 把形式参数的声明用括号括起来是非法的，但给函数参数加上括号是合法的，所以通过加一对括号，强迫编译器按我们的方式来工作：

   list<int> data((istream_iterator<int>(dataFile)), istream_iterator<int>());

   更好的是使用具名 iterator 对象(尽管使用匿名对象是一种趋势)

   istream_iterator<int> dataBegin(dataFile);

   istream_iterator<int> dataEnd;

   list<int> data (dataBegin, dataEnd);

✧ **条款 7：如果容器中包含了通过 new 操作创建的指针，切记在容器对象析构前将指针 delete 掉**

1) 用 shared_ptr

✧ **条款 8：切勿创建包含 auto_ptr 的容器对象**

✧ **条款 9：慎重选择删除元素的方法**

1) 要删除容器中有特定值的所有对象：

   如果容器是 vector、string 或 deque，则使用 erase-remove 习惯用法

   // c 是 vector，string 或 deque，erase-remove 习惯用法是

   // 删除特定值元素的最好办法

   c.erase(remove(c.begin(), c.end(), 1963), c.end());

   如果是 list，这使用 list::remove

   如果容器是一个标准关联容器，则使用它的 erase 成员函数

2) 要删除容器中满足特定判别式(条件)的所有对象:

   如果容器是 vector、string 或 deque，则使用 erase_remove_if 的习惯用法

```
// c 是 vector，string 或 deque，erase-remove-if 习惯用法是
// 删除特定值元素的最好办法
bool badValue(int);
c.erase(remove_if(c.begin(), c.end(), badValue), c.end());
```

如果容器是 list，则使用 list::remove_if

如果容器是一个关联容器，则使用 remove_copy_if 和 swap，或者写一个循环来遍历容器中的元素，记住当把迭代器传给 erase 时，对它进行后缀递增

```
AssocContainer<int> c;
…
AssocContainer<int> goodValues;
remove_copy_if(c.begin(), c.end(),
                    inserter(goodValues, goodValues.end()), badValue);
c.swap(goodValues);
OR
for AssocContainer<int>::iterator I = c.begin(); i != c.end();) {
  if (badValue(*i)) c.erase(i++);
  else ++i;
}
```

3) 要在循环内部做某些(除了删除之外的)操作

如果容器是一个标准序列容器，则写一个循环来遍历容器中的元素，记住每次调用 erase 时，要用它的返回值更新迭代器

vector/string/deque 不能这样做，因为对于这类容器，调用 erase 不仅使指向被删除元素的迭代器失效，也会使被删除元素之后的所有迭代器都无效。我们需要另外一种技巧——利用 erase 的返回值。erase 返回指向紧随被删除元素的下一个元素的有效迭代器(list 即可以采用关联容器做法，也可以采用序列容器做法，但一般保持和 vector/string/deque 一致)

```
for (SeqContainer<int>::iterator i = c.begin(); i != c.end(); ) {
  if (badValue(*i)) {
    logFile << "Erasing " << *i << "\n";
    i = c.erase(i);
  }
  else ++i;
}
```

如果容器是一个标准关联容器，则写一个循环遍历容器中的元素，记住当把迭代器传给 erase 时，要对迭代器做后缀递增

```
for AssocContainer<int>::iterator I = c.begin(); i != c.end();) {
  if (badValue(*i)) {
    logFile << "Erasing " << *i << "\n";
    c.erase(i++);
  }
  else ++i;
}
```

✧ **条款 10：** 了解分配子(allocator)的约定和限制

1) 多数标准容器从不向与之关联的分配子(allocator)申请内存

2) 在 C++标准中，一个类型为 T 的对象，它的默认分配子(allocator<T>)提供了两个类型定义，分别为 allocator<T>::pointer 和 allocator<T>::reference，用户定义的分配子也应该提供这些类型定义

3) C++标准说，STL 的实现可以假定所有属于同一种类型的分配子对象都是等价的，并且相互比较的结果总是相等的(allocator 不能带状态)

4) 基于节点的容器从来没有向与之关联的分配子(allocator)申请内存

```
template<typename T, typename Allocator = allocator<T> >
class list {
private:
   Allocator alloc;
   struct ListNoe {
      T data;
      ListNode *prev;
      ListNode *next;
   };
   …
};
```

当新的节点加入到 list 中时，我们需要通过分配子获得内存，但我们并不是需要 T 的内存，而是需要包含 T 的 ListNode 的内存。这使得我们的 Allocator 对象变得毫无用处，因为它不能为 ListNode 分配内存。list 所需要的是这样的一种方式，即如何从它已有的分配子类型到达与 ListNode 相适应的分配子。list 实现这样做：

```
template<typename T>
class allocator {
public:
   template<typename U>
   struct rebind {
      typedef allocator<U> other;
   };
   …
};
```

相应的 ListNode 的分配子的类型是 Allocator::rebind<ListNode>::other

5) 总结

- 你的分配子是一个模板，模板参数 T 代表你为分配内存的对象的类型
- 提供类型定义 pointer 和 reference，但是始终让 pointer 为 T*，reference 为 T&
- 千万别让你的分配子拥有随对象而不同的状态(per-object state)。通常，分配子不应该有非静态的数据成员
- 记住，传给分配子的 allocate 成员函数的是那些要求内存的对象的个数，而不是所需的字节数。同时要记住，这些函数返回 T*指针(通过 pointer 类型定义)，即使尚未有 T 对象被构造出来
- 一定要提供嵌套的 rebind 模板，因为标准容器依赖该模板

✧ **条款 11：理解自定义分配子的合理用法**

1) 案例研究

```
class Heap1 {
public:
  …
  static void *alloc(size_t numBytes, cons void *memryBlockToBeNear);
  static void dealloc(void *ptr);
  …
};

class Heap2 {…}; // has same interfaces as Heap1

template<typename T, typename Heap>
class SpecificHeapAllocator {
public:
  …
  pointer allocate (size_type numObjects, const void *localityHint = 0)
  {
    return static_cast<pointer>(Heap::alloc(numObjects * sizeof(T),
                                            localityHint));
  }

  void dealloc (pointer ptrToMemory, size_type numObjects)
  {
    Heap:dealloc (ptrToMemory);
  }
};

vector<int, SpecialHeapAllocator<int, Heap1> > v;
list<Widget, SpecialHeapAllocator<Widget, Heap2> > l;
```

- ✧ **条款 12：** 切勿对 STL 容器的线程安全性有不切实际的依赖

（二） vector 和 string

- ✧ **条款 13：** vector 和 string 优先动态分配数组
- ✧ **条款 14：** 使用 reserve 来避免不必要的重新分配

1) realloc 操作分 4 步
   - ■ 分配一块大小为当前容量的某个倍数的新内存。一般以 2 的倍数增长
   - ■ 把容器的所有元素从旧的内存拷贝到新内存中
   - ■ 析构掉旧内存中的对象
   - ■ 释放旧内存
2) vector/string resize(n)操作
   - ■ 强迫容器改变包含 n 个元素的状态。
   - ■ 如果 n 比当前 size 要小，则容器尾部的元素将被析构
   - ■ 如果 n 比当前 size 要大，则通过默认构造函数创建的新元素将被添加到容器的末尾
   - ■ 如果 n 比当前容量(capacity)还要大，那么在添加元素之前，将先重新分配内存

3) vector/string reserve(n)操作

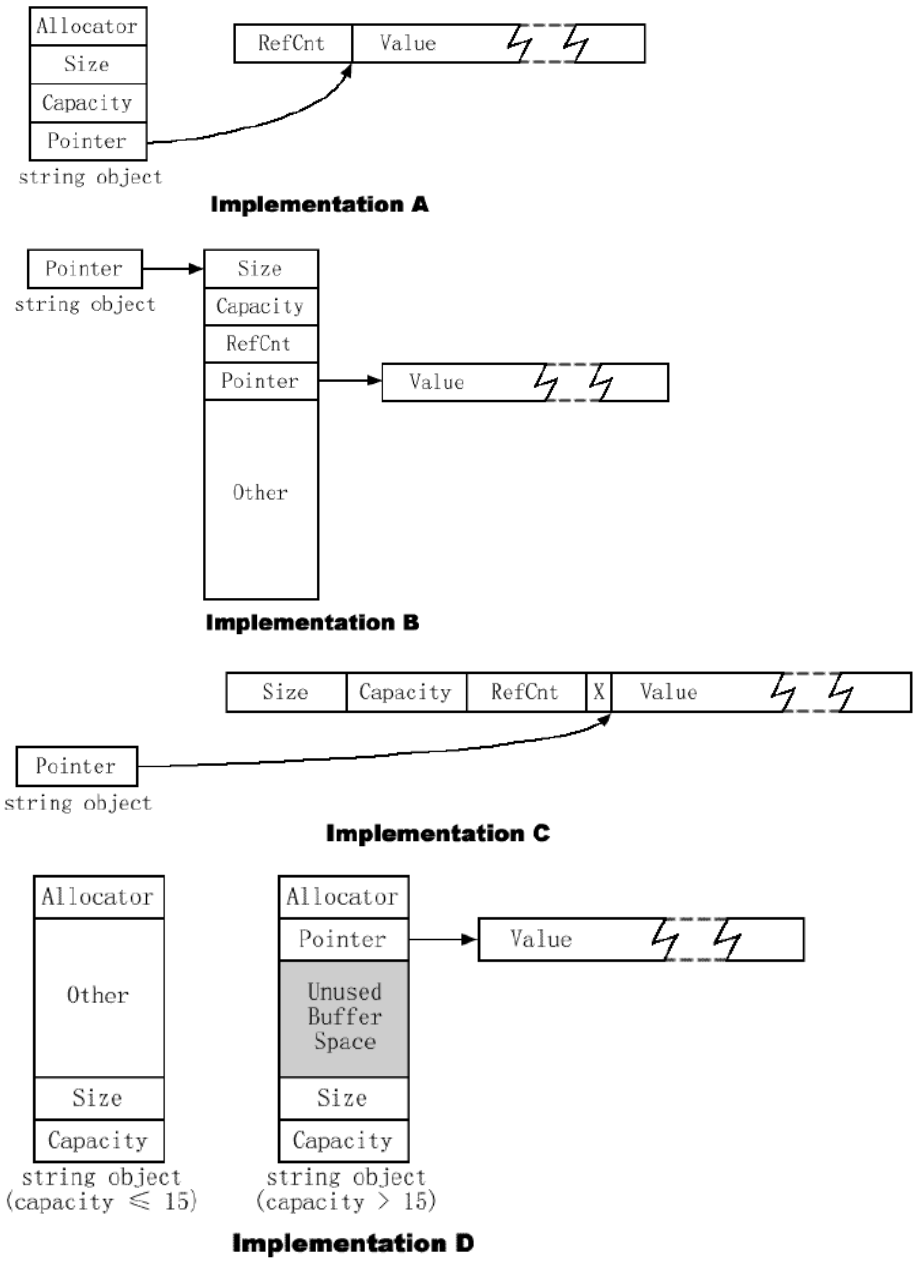
- 如果 n 比当前容量(capacity)小，则 vector 忽略该调用，什么也不做；而 string 则可能把自己的容量(capacity)减少为 size()和 n 中的最大值，但是 string 的 size 肯定保持不变
- 如果 n 比当前容量大，重新分配内存

4) 有两种方法是用 reserve 以避免不必要的重新分配

- 第一种是，若能确切知道或大致预计容器中最终会有多少元素，则此时使用 reserve。
- 第二种是，先预留足够大的空间(根据需要而定)，然后，当把所有的数据都加入后，再去除多余的容量(条款 17)

✧ **条款 15：**注意 string 实现的多样性

1) 可能的实现



✧ **条款 16：**了解如何把 vector 和 string 数据传给旧的 API

1) void doSomething(int *parr, int size); // C API
vector<int> v;
if (!v.empty()) {
  doSomthing(&v[0], v.size());
}
2) void doSomthing(const char* pStr); // C API
doSomething(s.c_str());
3) string 中的数据不一定存储在连续的内存中；string 的内部表示不一定是以空字符结尾。c_str()返回一个指向字符串的指针
4) 当其他的容器需要和 C API 交互时，先把他们挪到 vector 或 string 中，然后再和 C API 交互

✧ **条款 17：使用“swap 技巧”去除多余的容量**

1) vector<Contestant>(contestants).swap(contestants); // shrink-to-fit
vector<Contestant>(contestants)创建一个临时变量，它是 contestants 的一个 copy；由 vector 的 copy 构造函数完成。然而，vector 的 copy 构造函数只为所拷贝的元素分配所需要的内存，所以这个临时变量没有多余的容量。然后把临时变量中的数据 contestants 中的数据做 swap 操作，在这之后，contestants 具有了被去除之后的容量，而临时变量的容量则变成了原先 contestants 臃肿的容量。在语句结尾时，临时变量被析构，从而释放了先前为 contestants 所占据的内存
2) 在做 swap 的时候，不仅两个容器的内容被交换，同时它们的迭代器、指针和引用也将被交换(string 除外)。在 swap 发生之后，原先指向某个容器中元素的迭代器、指针和引用依然有效，并指向同样的元素——但是这些元素已经在另外一个容器中了

✧ **条款 18：避免使用 vector<bool>**

1) vector<bool>不是一个 STL 容器，而且它并不存储 bool(存储的是 bit)
2) 替代品：deque<bool>或 bitset

（三） 关联容器

✧ **条款 19：理解相等(equality)和等价(equivalence)的区别**

1) 关联容器是基于等价而不是相等的，一般以 operator <为基础
2) find 算法是基于相等的，一般以 operator == 为基础

✧ **条款 20：为包含指针的关联容器指定比较类型**

1) 不然的话，就是以指针作为排序的 key，或许这不是我们想要的

✧ **条款 21：总是让比较函数在等值情况下返回 false**

1) 案例研究
set<int, less_equal<int> > s;
s.insert(10);
s.insert(10);
less_equal => “operator <=”，集合检查一下表达式是否为真：
$!(10_A <= 10_B)$ && $!(10_B <= 10_A)$ // 检查 10_A 和 10_B 的等价性
！(true) && !(true) => false && false => false
最终导致重复值插入到 set 中
struct StringPtrGreater: public binary_function<const string*, const string*, bool>
{

```
    bool operator() (const string* lhs, const string* rhs) const
    {
        return !(*lhs < *rhs);    // 错误
    }
};
struct StringPtrGreater: public binary_function<const string*, const string*, bool>
{
    bool operator() (const string* lhs, const string* rhs) const
    {
        return *rhs < *lhs);    // 正确
    }
};
```

2) 对关联容器所使用的比较函数总是要对相等的值返回 false。相等的值不会有前后顺序，所以总是返回 false。

3) 对于 multiset 和 multimap 也是一样。

```
multiset<int, less_equal<int> > s
s.insert(10);
s.insert(10);
```

现在 s 中有两个 10，我们期望 equal_range 操作返回一对迭代器，包含这两个值的区间(equal_range 返回一个等价的区间)。但这是不可能的，因为在这里 10_A 和 10_B 是不等价的

4) 从技术上来说，用于对关联容器排序的比较函数必须为它们所比较的对象定义一个严格的弱序化(strict week ordering)。任何定义了“严格弱序化”的函数必须对相同值的两个拷贝返回 false

✧ **条款 22：切勿直接修改 set 或 multiset 中的键**

1) map 和 multimap 的 key 部分是 const 的，所以无法直接改变

2) set 和 multiset 的值(和 key)不是 const 的，可以直接改变，但改变之后，容器就得以破坏

3) 试图修改 set 或 multiset 中元素的代码将是不可移植的

```
EmpIdSet se;
Employee selectedID;
…
EmpIDSet::iterator i = se.find(selectedID);
if (i != se.end()) {
    i->setTitle (“Corporate Deity”); // 有些 STL 实现将认为这个不合法，
```

```
                                        // 因为*i 是 const
}
```

修改 set 或 multiset 中元素的非键部分是合理的，一个可行的作法是：

```
if (i != se.end()) {
   const_cast<Employee&>(*i).setTitle("Corporate Deity");
}
// the following one is wrong
if (i != se.end()) {
   static_cast<Employee>(*i).setTitle("Corporate Deity"); // temp object is created
}
```

4) 执行下面 5 个步骤来进行对 set/multiset/map/multimap 中的元素做修改

- 找到你想修改的容器的元素
- 为将要被修改的元素做一份拷贝。在 map 和 multimap 的情况下，请记住，不要把该拷贝的第一个部分声明为 const。毕竟，你想改变它
- 修改该拷贝，使它具有你期望它在容器中的值
- 把改元素从容器中删除，通常是通过调用 erase 来进行的
- 把新的值插入到容器中。如果按照容器排列顺序，新元素的位置可能与被删除元素的位置相同或紧邻，则使用 hint 形式的 insert，以便把插入的效率从对数事件提高到常数时间。把你第一步得来的迭代器作为提示信息

```
EmpIdSet se;
Employee selectedID;
…
EmpIDSet::iterator i = se.find(selectedID);    // 1)
if (i != se.end()) {
   Employee e(*i);                             // 2)
   e.setTitle ("Corporate Deity");             // 3)
   se.erase(i++);                              // 4)
   se.insert(i, e);                            // 5)
}
```

✧ **条款 23：**考虑用排序的 vector 替代关联容器

1) 标准关联容器的典型实现是平衡二叉查找树。一个平衡二叉查找树是一个对插入、删除和查找的混合操作优化的数据结构。换句话说，它被设计为应用于进行一些插入，然后一些查找，然后可能再进行一些插入，然后也许一些删除，然后再来一些查找，然后更多的插入或删除，然后更多的查找等。这个事件序列的关键特征是插入、删除和查找都是混合在一起的。

2) 在很多应用中，使用数据结构并没有那么混乱。它们对数据结构的使用可以总结为这样的三个截然不同的阶段。以这种方式使用它们的数据结构的应用来说，一个 vector 可能比一个关联容器能提供更高的性能(时间和空间上都是)。但不是任意的 vector 都会，只有有序 vector。因为只有有序容器才能正确地使用查找算法_binary_search、lower_bound、equal_range 等(更好的 locality，page in/page out，关联容器占用跟多的内存)

- 建立。通过插入很多元素建立一个新的数据结构。在这个阶段，几乎所有的操作都是插入和删除。几乎没有或根本没有查找。
- 查找。在数据结构中查找指定的信息片。在这个阶段，几乎所有的操作都

是查找。几乎没有或根本没有插入和删除。

- 重组。修改数据结构的内容，也许通过删除所有现有数据和在原地插入新数据。从动作上说，这个阶段等价于阶段 1。一旦这个阶段完成，应用程序返回阶段 2。

✧ **条款 24：**当效率至关重要时，请在 map::operator[]和 map::insert 之间慎重做出选择

1) map::operator[]的设计目的是为了提供“添加和更新”的功能。它的工作原理是 operator[]返回一个与 k 关联的值对象的引用。然后 v 赋值给所引用(从 operator[]返回的)的对象。

- 当要更新一个已存在的键的关联值时很直接，因为已经有 operator[]可以用来返回引用的值对象。(当 update 已存在的对象是 operator[]优于 insert)
- 如果 k 还不在 map 里，operator[]就没有可以引用的值对象。那样的话，它使用值类型的默认构造函数从头开始建立一个，然后 operator[]返回这个新建立对象的引用。(当 insert 一个不存在的对象是，insert 优于 operator[])

```
template<typename MapType, // map 的类型
         typename KeyArgType, // KeyArgType 和 ValueArgtype
         typename ValueArgtype> //  是类型参数
typename MapType::iterator
efficientAddOrUpdate(MapType& m,
                     const KeyArgType& k,
                     const ValueArgtype& v)
{
  typename MapType::iterator lb = m.lower_bound(k);
  if (lb != m.end() && !(m.key_comp()(k, lb->first))) {
    lb->second = v;
    return lb;
  }
  else {
    typedef typename MapType::value_type MVT;
    return m.insert(lb, MVT(k, v));   // 常数时间
  }
}
```

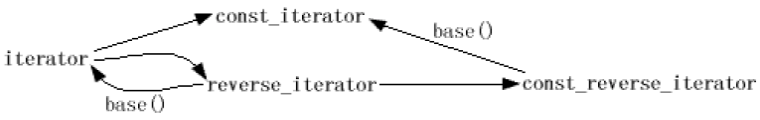
✧ **条款 25：**熟悉非标准的哈希容器

1) hash_set/hash_multiset/hash_map/hash_multimap

（四） 迭代器

✧ **条款 26：**iterator 优先于 const_iterator，reverse_iterator 以及 const_reverse_iterator

1) 各种 iterator 之间的关系。通过 base()得到的迭代器也许并非你所期望的迭代器。也没办法从 const_iterator 转换得到 iterator，或从 const_reverse_iterator 得到 reverse_iterator。这意味着，如果你得到一个 const_iterator 或 const_reverse_iterator，你就会发现很难将这些迭代器与容器的某些成员函数一起使用。这些成员函数要求 iterator 作为参数。如果你利用迭代器来指定插入或删除元素的位置，则常量的迭代器往往是没有用处的



2) 减少混用不同类型(const 和 non-const)迭代器的机会

3) vector<T> insert 和 erase 都接收非常量迭代器

✧ **条款 27：使用 distance 和 advance 将容器的 const_iterator 转换成 iterator**

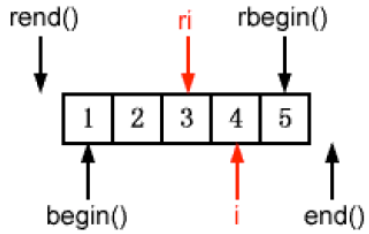
1) typedef deque<int> IntDeque;
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;
IntDeque d;
ConstIter ci;
… // ci point to d

Iter i(ci); // wrong
Iter i (const_cast<Iter>(ci)); // wrong again

Iter i (d.begin());
advance(I, distance<ConstIter>(i, ci)); // Correct

✧ **条款 28：正确理解由 reverse_iterator 的 base()成员函数所产生的 iterator 的用法**

1) 注意 off-by-one
vector<int>::reverse_iterator ri = find(v.rbegin(), v.rend(), 3);
vector<int>::iterator i(ri.base()); // 使 i 和 ri 的 base 一样



2) 如果要在 reverse_iterator ri 指定的位置上插入新元素，则只需要在 ri.base()位置处插入元素即可。对于插入操作而言，ri 和 ri.base()是等价的，ri.base()是真正与 ri 对应的 iterator

3) 如果要在一个 reverse_iterator ri 指定的位置上删除一个元素，则需要在 ri.base()前面的位置上执行删除操作。对于删除操作而言，ri 和 ri.base()是不等价的，ri.base()不是与 ri 对应的 iterator
vecot<int>::reverse_iterator ri = find(v.rbegin(), v.rend(), 3); // 同上，ri 指向 3
v.erase(--ri.base()); // 尝试删除 ri.base()前面的元素；
// 对于 vector/string，一般来说编译不通过
// C/C++都规定了从函数返回的指针不应该被修改
v.erase((++ri).base()); // 删除 ri 指向的元素；这下没问题了

✧ **条款 29：对于逐个字符的输入请考虑使用 istreambuf_iterator**

1) istreambuf_iterator/ostreambuf_iterator 不执行格式化，更高效
// 跳过了空白字符
ifstream inputFile(“interestingData.txt”);

```
string fileData((istream_iterator<char>(inputFile)), istream_iterator<char>());
// 没有跳过空白字符
ifstream inputFile("interestingData.txt");
inputFile.unsetf(ios::skipws);
string fileData((istream_iterator<char>(inputFile)), istream_iterator<char>());

// efficient way
ifstream inputFile("interestingData.txt");
string fileData((istreambuf_iterator<char>(inputFile)),
                istreambuf_iterator<char>());
```

（五） 算法

✧ 条款 30：确保目标区间足够大

1)

```
int transmogrify(int x);                    // 这个函数从x
                                            // 产生一些新值
vector<int> values;
...                                         // 把数据放入values
vector<int> results;                        // 把transmogrify应用于
transform(values.begin(), values.end(),     // values中的每个对象
          results.end(),                    // 把这个返回的values
          transmogrify);                    // 附加到results
                                            // 这段代码有bug!
```

// back_inserter calls push_back internally (vector/string/deque/list)

```
vector<int> results;                        // 把transmogrify应用于
transform(values.begin(), values.end(),     // values中的每个对象，
          back_inserter(results),           // 在results的结尾
          transmogrify);                    // 插入返回的values
```

// front_insert calls push_front internally (deque/list)

```
...                                         // 同上
list<int> results;                          // results现在是list
transform(values.begin(), values.end(),     // 在results前端
          front_inserter(results),          // 以反序
          transmogrify);                    // 插入transform的结果
```

// inserter 到特定的位置

```
vector<int> values;                         // 同上
vector<int> results;
...
results.reserve(results.size() + values.size());    // 确定results至少
                                            // 还能装得下
                                            // values.size()个元素
transform(values.begin(), values.end(),     // 同上，
    inserter(results, results.begin() + results.size() / 2),// 但results
    transmogrify);                          // 没有任何重新分配操作
```

✧ **条款 31：了解各种与排序有关的选择**

1) 总结
   - ■ 如果你需要在 vector、string、deque 或数组上进行完全排序，你可以使用 sort 或 stable_sort
   - ■ 如果你有一个 vector、string、deque 或数组，你只需要排序前 n 个元素，应该用 partial_sort
   - ■ 如果你有一个 vector、string、deque 或数组，你需要鉴别出第 n 个元素或你需要鉴别出最前的 n 个元素，而不用知道它们的顺序，nth_element 是你应该注意和调用的
   - ■ 如果你需要把标准序列容器的元素或数组分隔为满足和不满足某个标准，你大概就要找 partition 或 stable_partition
   - ■ 如果你的数据是在 list 中，你可以直接使用 partition 和 stable_partition，你可以使用 list 的 sort 来代替 sort 和 stable_sort。如果你需要 partial_sort 或 nth_element 提供的效果，你就必须间接完成这个任务：(1) 将 list 的元素拷贝到一个提供随机访问迭代器的容器中，然后对该容器执行期望的算法；(2) 先创建一个 list::iterator 的容器，再对该容器执行相应的算法，然后通过其中的迭代器访问 list 的元素；(3) 利用一个包含迭代器的有序容器中的信息，通过反复调用 splice 成员函数，将 list 中的元素调整到期望的目标位置
2) priority_queue
3) 对于在这些排序算法之间作选择的建议是让你的选择基于你需要完成的任务上，而不是考虑性能。

✧ **条款 32：如果确实需要删除元素，则需要在 remove 这一类算法之后调用 erase**

1) remove/remove_if/unique
2) v.erase(remove(v.begin(), v.end(), 99), v.end());

✧ **条款 33：对包含指针的容器使用 remove 这一类算法时要特别小心**

1) 因为一些指针被覆盖从而导致资源泄漏

✧ **条款 34：了解哪些算法要求使用排序的区间作为参数**

1) 要求排序区间的 STL 算法

binary_search	lower_bound
upper_bound	equal_range
set_union	set_intersection
set_difference	set_symmetric_difference
merge	inplace_merge
includes	

unique 和 unique_copy 不要求排序区间，但通常会与排序区间一起使用

2) 如果你为一个算法提供了一个排序的区间，而这个算法也带一个比较函数作为参数，那么你一定要保证你传递的比较函数与这个排序区间所使用的比较函数有一致的行为

不一致的例子：

```
vector<int> v;
…
```

```
sort(v.begin.(). v.end(), greater<int>());
…
// 错误，默认情况下 binary_search 采用<排序
bool a5Exists = binary_search(v.begin(), v.end(), 5);
// 正确
bool a5Exists = binary_search(v.begin(), v.end(), 5, greater<int>());
```

3) 所有要求排序区间的算法均使用等价性来判断两个对象是否“相同”。unique/unique_copy 在默认情况下使用“相等”来判断两个对象是否“相同”，当然可以改变这个默认行为

✧ **条款 35：通过 mismatch 或 lexicographical_compare 实现忽略大小写的字符串比较**

1) mismatch

```
int ciCharCompare(char c1, char c2)                    // 忽略大小写比较字符
{                                                      // c1和c2，如果c1 < c2返回-1，
                                                       // 如果c1==c2返回0，如果c1 > c2返回1
    int Ic1 = tolower(static_cast<unsigned char>(c1));// 这些语句的解释
    int Ic2 = tolower(static_cast<unsigned char>(c2));// 看下文

    if (Ic1 < Ic2) return -1;
    if (lc1 > Ic2) return 1;
    return 0;
}
```

```
int ciStringCompareImpl(const string& si, const strings s2)
{
    typedef pair<string::const_iterator,               // PSCI = “pair of
            string::const_iterator> PSCI;  // string::const_iterator”
    PSCI p = mismatch(                                 // 下文解释了
            s1.begin(), s1.end(),        // 为什么我们
            s2.begin(),                  // 需要not2；参见
            not2(ptr_fun(ciCharCompare))); // 条款41解释了为什么
                                         // 我们需要ptr_fun
    if (p.first== s1.end()) {                          // 如果为真，s1等于
        if (p.second == s2.end()) return 0;            // s2或s1比s2短
        else return -1;
    }
    return ciCharCompare(*p.first, *p.second);         // 两个字符串的关系
}                                                      // 和不匹配的字符一样
```

```
int ciStringCompare(const string& s1, const string& s2)
{
    if (s1.size() <= s2.size()) return ciStringCompareImpl(s1, s2);
    else return -ciStringCompareImpl(s2, s1);
}
```

2) lexicographical_compare

```
bool ciCharLess(char c1, char c2)                    // 返回在忽略大小写
{                                                    // 的情况下c1是否
                                                     // 在c2前面;
    tolower(static_cast<unsigned char>(c1)) <        // 条款46解释了为什么
        tolower(static_cast<unsigned char>(c2));     // 一个函数对象可能
}                                                    // 比函数好

bool ciStringCompare(const string& s1, const string& s2)
{
    return lexicographical_compare(s1.begin(), s1.end(),      // 关于这个
                                   s2.begin(), s2.end(),  // 算法调用的
                                   ciCharLess);       // 讨论在下文
}
```

3) stricmp (most efficient)，但是不是 standard C++ lib

```
int ciStringCompare(const string& s1, const string& s2)
{
    return stricmp(s1.c_str(), s2.c_str());          // 你的系统上的
}                                                    // 函数名可能
                                                     // 不是stricmp
```

✧ **条款 36：理解 copy_if 算法的正确实现**

1) 正确实现

```
template<typename InputIterator,                     // 一个copy_if的
         typename OutputIterator,                    // 正确实现
         typename Predicate>
OutputIterator copy_if(InputIterator begin,
                       InputIterator end,
                       OutputIterator destBegin,
                       Predicate p) {
    while (begin != end) {
        if (p(*begin))*destBegin++ = *begin;
        ++begin;
    }

    return destBegin;
}
```

✧ **条款 37：使用 accumulate 或者 for_each 进行区间统计**

1) 传递给 accumulate 的函数不允许有副作用；传给 for_each 的函数可以有副作用

```
class PointAverage:
        public binary_function<Point, Point, Point> {        // 参见条款40
public:
    PointAverage(): numPoints(0), xSum(0), ySum(0) {}
    const Point operator()(const Point& avgSoFar, const Point& p) {
        ++numPoints;
        xSum += p.x;
        ySum += p.y;
        return Point(xSum/numPoints, ySum/numPoints);
    }

private:
    size_t numPoints;
    double xSum;
    double ySum;
};
```

```
list<Point> lp;
...
Point avg =                          // 对lp中的point求平均值
    accumulate(lp.begin(), lp.end(),
               Point(0, 0), PointAverage());
```

上述实现结果其实是未定义的，但实际上都能 work 很好

```
struct Point {...};                          // 同上
class PointAverage:
        public unary_function<Point, void> {        // 参见条款40
public:
    PointAverage(): xSum(0), ySum(0), numPoints(0) {}
    void operator()(const Point& p)
    {
        ++numPoints;
        xSum += p.x;
        ySum += p.y;
    }
    Point result() const
    {
        return Point(xSum/numPoints, ySum/numPoints);
    }

private:
    size_t numPoints;
    double xSum;
    double ySum;
};

list<Point> lp;
...
Point avg = for_each(lp.begin(), lp.end(), PointAverage()).result;
```

2) accumulate 多用于计算出一个区间的统计信息，for_each 是对一个区间的每个元素做一个操作。很多时候两者可以替换使用。区别就是在什么样的环境下谁更是代码清晰

（六） 函数子、函数子类、函数及其他

✧ 条款 38：遵循按值传递的原则来设计函数子类

1) 将函数对象按引用来传递，有些 STL 算法的某些实现甚至不能编译通过
2) 由于函数对象总是按值传递和返回，所以必须确保函数对象尽可能小，函数对象单态。如果函数对象很大的话和需要多态的话，可以考虑使用 pimpl 方式实现函数对象

```
template<typename T>                    // 用于修改的BPFC
class BPFCImpl
    public unary_function<T, void> {    // 的新实现类
private:
    Widget w;                           // 以前在BPFC里的所有数据
    int x;                              // 现在在这里
    ...
    virtual ~BPFCImpl();                // 多态类需要
                                        // 虚析构函数
    virtual void operator()(const T& val) const;
    friend class BPFC<T>;               // 让BPFC可以访问这些数据
};

template<typename T>
class BPFC:                             // 小的，单态版的BPFC
    public unary_function<T, void> {
private:
    BPFCImpl<T> *pImpl;                 // 这是BPFC唯一的数据

public:
    void operator()(const T& val) const     // 现在非虚；
    {                                   // 调用BPFCImpl的
        pImpl->operator() (val);
    }
    ...
};
```

✧ **条款 39：**确保判别式是“纯函数”

1) 请将判别式类中的 operator()声明为 const 且是纯函数
2) 案例研究，删除第三个 widget 的错误实现。它不仅删除了容器中的第三个元素，而且同时还删除了第六个

```
class BadPredicate:                 // 关于这个基类的更多信息
    public unary_function<Widget, bool> {  // 请参见条款40
public:
    BadPredicate(): timesCalled(0) {}       // 把timesCalled初始化为0
    bool operator()(const Widget&)
    {
        return ++timesCalled == 3;
    }

private:
    size_t timesCalled;
};

vector<Widget> vw;                  // 建立vector，然后
                                    // 放一些Widgets进去
vw.erase(remove_if(vw.begin(),      // 去掉第三个Widget;
                   vw.end(),        // 关于erase和remove_if的关系
                   BadPredicate()),     // 请参见条款32
                   vw.end());
```

remove_if 的一个可能的实现，从这个实现可以了解为什么第六个元素也被删除了

```
template <typename FwdIterator, typename Predicate>
FwdIterator remove_if(FwdIterator begin, FwdIterator end, Predicate p)
{
    begin = find_if(begin, end, p);
    if (begin == end) return begin;
    else {
        FwdIterator next = begin;
        return remove_copy_if(++next, end. begin, p);
    }
}
```

✧ **条款 40：若一个类是函数子，则应使它可配接**

1) 如果函数子类的 operator()只有一个参数，那么它应该 std::unary_function 继承；如果函数子类有两个参数，那么应该从 std::binary_function 继承。标准的 std::unary_/binary_function 定义了一些特殊的类型，这样可以和 not1、not2、bind1st 和 bind2nd 配接
2) 一般情况下，传递给 unary_function 或 binary_function 的非指针类型需要去掉 const 和引用(&)部分；如果 operator()带有指针参数，则传递给 unary_function 或 binary_function 的类型与 operator()的参数和返回类型一致。

```
template<typename T>
class MeetsThreshold: public std::unary_function<Widget, bool>{
private:
    const T threshold;

public:
    MeetsThreshold(const T& threshold);
    bool operator()(const Widget&) const;
    ...
};
struct WidgetNameCompare:
    public std::binary_function<Widget, Widget, bool>{
    bool operator()(const Widget& lhs, const Widget& rhs) const;
};
```

```
struct PtrWidgetNameCompare:
    public std::binary_function<const Widget*, const Widget*, bool> {
    bool operator()(const Widget* lhs, const Widget* rhs) const;
};
```

3) STL 总是假设每个函数子类只有一个 operator()成员函数

✧ **条款 41：理解 ptr_fun，mem_fun 和 mem_fun_ref 的来由**

1) ptr_fun：定义一些类型，使得函数或函数子可配接
2) mem_fun：返回 mem_fun_t，用于配接“指针”(容器)
3) mem_fun_ref：返回 mem_fun_ref_t，用于配接“对象”(容器)

✧ **条款 42：确保 less<T>与 operator<具有相同的语义**

1) less<T>默认情况下调用 operator <。应该尽量避免修改 less 的行为，因为这样

做可能会误导其它的程序员。如果你希望以一种特殊的方式来排序对象，那么最好创建一个特殊的函数子类

（七） 在程序中使用 STL

✧ 条款 43：算法调用优先于手写的循环

1) 效率、正确性和可维护性，算法在绝大部分情况下都高于手写循环
2) 关于代码清晰度的底线是：这完全取决于你想在循环里做的是什么。如果你要做的是算法已经提供了的，或者非常接近于它提供的，调用泛型算法更清晰。如果循环里要做的事非常简单，但调用算法时却需要使用绑定和适配器或者需要独立的仿函数类，你恐怕还是写循环比较好。最后，如果你在循环里做的事相当长或相当复杂，天平再次倾向于算法。因为长的、复杂的通常总应该封装入独立的函数。只要将循环体一封装入独立函数，你几乎总能找到方法将这个函数传给一个算法（通常是 for_each），以使得最终代码直截了当。

✧ 条款 44：容器的成员函数优先同名的算法

1) 成员函数有更好的效率

✧ 条款 45：正确区分 count、find、binary_search、lower_bound、upper_bound 和 equal_range

1) 总结

你想知道的	使用的算法		使用的成员函数	
	在无序区间	在有序区间	在set或map上	在multiset或multimap上
期望值是否存在？	find	binary_search	count	find
期望值是否存在？如果有，第一个等于这个值的对象在哪里？	find	equal_range	find	find或lower_bound（参见下面）
第一个不在期望值之前的对象在哪里？	find_if	lower_bound	lower_bound	lower_bound
第一个在期望值之后的对象在哪里？	find_if	upper_bound	upper_bound	upper_bound
有多少对象等于期望值？	count	equal_range，然后distance	count	count
等于期望值的所有对象在哪里？	find（迭代）	equal_range	equal_range	equal_range

✧ 条款 46：考虑使用函数对象而不是函数作为 STL 算法的参数

1) 编译器可以内联函数对象，但不会内联函数指针。
2) C++的 sort 算法总是由于 C 的 qsort 算法，C++以内联方式调用 operator()
3) StringSize 可能性能更好。mem_fun_ref(&string::size)肯定不能内联

```
set<string> s;
...
transform(s.begin(), s.end(),
          ostream_iterator<string::size_type>(cout, "\n"),
          mem_fun_ref(&string::size));
struct StringSize:
    public unary_function<string, string::size_type>{     // 参见条款40
    string::size_type operator()(const string& s) const
    {
        return s.size();
    }
};

transform(s.begin(), s.end(),
          ostream_iterator<string::size_type>(cout, "\n"),
          StringSize());
```

- ✧ **条款 47：**避免产生“直写型”(write-only)的代码
- ✧ **条款 48：**总是包含正确的头文件
  1) C++和 C 标准不同，它没有定义标准库中头文件之间的相互包含关系
- ✧ **条款 49：**学会分析与 STL 相关的编译器诊断信息
- ✧ **条款 50：**熟悉与 STL 相关的 Web 站点
  1) SGI STL: http://www.sig.com/tech/stl
  2) STLport: http//www.stlport.org
  3) Boost: http://www.boost.org