第 1 章 容器

第 1 条：慎重选择容器类型。

标准 STL 序列容器：vector、string、deque 和 list。

标准 STL 关联容器：set、multiset、map 和 multimap。

非标准序列容器 slist 和 rope。slist 是一个单向链表，rope 本质上是一“重型”string。

非标准的关联容器 hash_set、hase_multiset、hash_map 和 hash_multimap。

vector<char> 作为 string 的替代。(见第 13 条)

vector 作为标准关联容器的替代。(见第 23 条)

几种标准的非 STL 容器，包括数组、bitset、valarray、stack、queue 和 priority_queue。

你是否关心容器中的元素是如何排序的？如果不关心，选择哈希容器.

容器中数据的布局是否需要和 C 兼容？如果需要兼容，就只能选择 vector。(见第 16 条)

元素的查找速度是否是关键的考虑因素？如果是，就要考虑哈希容器、排序的 vector 和标准关联容器－或许这就是优先顺序。

对插入和删除操作，你需要事务语义吗？如果是，只能选择 list。因为在标准容器中，只有 list 对多个元素的插入操作提供了事务语义。

deque 是唯一的、迭代器可能会变为无效（插入操作仅在容器末尾发生时，deque 的迭代器可能会变为无效）而指向数据的指针和引用依然有效的标准 STL 容器。

第 2 条：不要试图编写独立于容器类型的代码。

如果你想编写对大多数的容器都适用的代码，你只能使用它们的功能的交集。不同的容器是不同的，它们有非常明显的优缺点。它们并不是被设计用来交换使用的。

你无法编写独立于容器的代码，但是，它们（指客户代码）可能可以。

第 3 条：确保容器中的对象拷贝正确而高效。

copy in,copy out，是 STL 的工作方式，它总的设计思想是为了避免不必要的拷贝。使拷贝动作高效并且防止剥离问题发生的一个简单办法是使容器包含指针而不是对象。

第 4 条：调用 empty 而不是检查 size()是否为 0。

理由很简单：empty 对所有的标准容器都是常数时间操作，而对一些 list 的实现，size 耗费线性时间。

第 5 条：区间成员函数优先于与之对应的单元素成员函数。

区间成员函数写起来更容易，更能清楚地表达你的意图，而且它们表现出了更高的效率。

第 6 条：当心 C＋＋编译器最烦人的分析机制。

把形参加括号是合法的，把整个形参的声明（包括数据类型和形参名字）用括号括起来是非法的。

第 7 条：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。

STL 很智能，但没有智能到知道是否该删除自己所包含的指针所指向的对象的程度。为了避免资源泄漏，你必须在容器被析构前手工删除其中的每个指针，或使用引用计数形式的智能指针（比如 Boost 的 sharedprt）代替指针。

第 8 条：切勿创建包含 auto_ptr 的容器对象。

拷贝一个 auto_ptr 意味着改变它的值。例如对一个包含 auto_ptr 的 vector 调用 sort 排序，结果是 vector 的几个元素被置为 NULL 而相应的元素被删除了。

第 9 条：慎重选择删除元素的方法。

要删除容器中指定值的所有对象：

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

```
SeqContainer<int> c;
c.erase(remove(c.begin(),c.end(),1963),c.end());
```

如果容器是 list，则使用 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。

要删除容器中满足特定条件的所有对象：

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

如果容器是 list，则使用 list::remove_if。

如果容器是一个标准关联容器，则使用 remove_copy_if 和 swap，或者写一个循环遍历容器的元素，记住当把迭代器传给 erase 时，要对它进行后缀递增。

```
AssocCOntainer<int> c;
...
AssocContainer<int> goodValues;
remove_copy_if(c.begin(), c.end(), inserter(goodValues,
goodValues.end()),badValue);
c.swap(goodValues);
```

或

```
for(AssocContainer<int>::iterator i = c.begin();i !=c.end();/* do nothing */){
if(badValue(*i)) c.erase(i++);
else ++i;
}
```

要在循环内部做某些（除了删除对象之外的）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，记住每次掉用 erase 时，要用它的返回值更新迭代器。

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

第 10 条：了解分配子(allocator)的约定和限制。

第 11 条：理解自定义分配子的合理用法。

第 12 条：切勿对 STL 容器的线程安全性有不切实际的依赖。

对一个 STL 实现你最多只能期望：
多个线程读是安全的。
多个线程对不同的容器写入操作是安全的。
你不能期望 STL 库会把你从手工同步控制中解脱出来，而且你不能依赖于任何线程支持。

第 2 章 vector 和 string

第 13 条：vector 和 string 优先于动态分配的数组。

如果用 new，意味着你要确保后面进行了 delete。

如果你所使用的 string 是以引用计数来实现的，而你又运行在多线程环境中，并认为 string 的引用计数实现会影响效率，那么你至少有三种可行的选择，而且，没有一种选择是舍弃 STL。首先，检查你的库实现，看看是否可以禁用引用计数，通常是通过改变某个预处理变量的值。其次，寻找或开发一个不使用引用计数的 string 实现。第三，考虑使用 vector<char>而不是 string。vector 的实现不允许使用引用计数，所以不会发生隐藏的多线程性能问题。

第 14 条：使用 reserve 来避免不必要的重新分配。

通常有两种方式来使用 reserve 以避免不必要的重新分配。第一种方式是，若能确切知道或大致预计容器中最终会有多少个元素，则此时可使用 reserve。第二种方式是，先预留足够大的空间，然后，当把所有的数据都加入后，再去除多余的容量。

第 15 条：注意 string 实现的多样性。

如果你想有效的使用 STL，那么你需要知道 string 实现的多样性，尤其是当你编写的代码必须要在不同的 STL 平台上运行而你又面临着严格的性能要求的时候。

第 16 条：了解如何把 vector 和 string 数据传给旧的 API。

如果你有个 vector v，而你需要得到一个只想 v 中的数据的指针，从而可把数据作为数组来对才，那么只需要使用＆v［0］就可以了，也可以用＆＊v.begin（），但是不好理解。对于 string s，随应的形式是 s.c_str()。
如果想用来自 C API 的数据来初始化一个 vector，那么你可以利用 vector 和数组的内存布局兼容性，先把数据写入到 vector 中，然后把数据拷贝到期望最终写入的 STL 容器中。

第 17 条：使用“swap 技巧”出去多余的容量。

vector<Contestant>(contestants).swap(contestants);
表达式 vector<Contestant>(contestants)创建一个临时的矢量，它是 contestants 的拷贝：这是由 vector 的拷贝构造函数来完成的。然而，vector 的拷贝构造函数只为所拷贝的元素分配所需要的的内存，所以这个临时矢量没有多余的容量。然后

我们把临时矢量中的数据和 contestants 中的数据作 swap 操作，在这之后，contestants 具有了被去除之后的容量，即原先临时变量的容量，而临时变量的容量则变成了原先 contestants 臃肿的容量。到这时，临时矢量被析构，从而释放了先前为 contestants 所占据的内存。

同样的技巧对 string 也实用：

```
string s;
...
string(s).swap(s);
```

第 18 条：避免使用 vector<bool>。

作为 STL 容器，vector<bool>只有两点不对。首先，它不是一个 STL 容器；其次，它并不存储 bool。除此以外，一切正常。因此最好不要使用它，你可以用 deque<bool>和 bitset 替代。vector<bool>来自一个雄心勃勃的试验，代理对象在 C＋＋软件开发中经常会很有用。C＋＋标准委员会的人很清楚这一点，所以他们决定开发 vector<bool>，以演示 STL 如果支持 “通过代理对象来存取其元素的的容器”。他们说，C＋＋标准中有了这个例子，于是，人们在实现自己的基于代理的容器时就有了一个参考。然而他们却发现，要创建一个基于代理的容器，同时又要求它满足 STL 容器的所有要求是不可能的。由于种种原因，他们失败了的尝试被遗留在标准中。

第 3 章 关联容器

第 19 条：理解相等（equality）和等价（equivalence）的区别。

标准关联容器总是保持排列顺序的，所以每个容器必须有一个比较函数（默认为 less）。等价的定义正是通过该比较函数而确定的。相等一定等价，等价不一定相等。

第 20 条：为包含指针的关联容器指定比较类型。

每当你创建包含指针的关联容器时，容器将会按照指针的值（就是内存地址）进行排序，绝大多数情况下，这不是你所希望的。

第 21 条：总是让比较函数在等值情况下返回 false。

现在我给你演示一个很酷的现象。创建一个 set，用 less_equal 作为它的比较类型，然后把 10 插入到该集合中：

```
set<int, less_equal<int> > s; //s 用"<=" 来排序
s.insert(10);
s.insert(10);
```

对于第二个 insert，集合会检查下面的表达式是否为真：

```
!(10a <= 10b) && !(10b <= 10a); //检查 10a 和 10b 是否等价，结果是!(true) && !(true) 为 false
```

结果集合中有两个 10！

从技术上讲，用于对关联容器排序的比较函数必须为他们所比较的对象定义个“严格的弱序化”（strict weak ordering）。

第 22 条：切勿直接修改 set 或 multiset 中的键。

如果你不关心可移植性，而你想改变 set 或 multiset 中元素的值，并且你的 STL 实现（有的 STL 实现中，比如 set<T>:: iterator 的 operator*总是返回 const T&，就不能修改了）允许你这么做，则请继续做下去。只是注意不要改变元素中的键部分，即元素中能够影响容器有序性的部分。

如果你重视可移植性，就要确保 set 和 multiset 中的元素不能被修改。至少不能未经过强制类型转换（转换到一个引用类型 const_cast<T&>）就修改。

如果你想以一种总是可行而且安全的方式来许该 set、multiset、map 和 multimap 中的元素，则可以分 5 个简单步骤来进行：

1. 找到你想修改的容器的元素。如果你不能肯定最好的做法，第 45 条介绍了如何执行一次恰当的搜索来找到特定的元素。
2. 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，你想要改变它。
3. 修改该拷贝，使它具有你期望的值。
4. 把该元素从容器中删除，通常是通过 erase 来进行的（见第 9 条）。
5. 把拷贝插到容器中去。如果按照容器的排列顺序，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“提示”（hint）形式的 insert，以便把插入的效率从对数时间提高到常数时间。把你从第 1 步得来的迭代器作为提示信息。

第 23 条：考虑用排序的 vector 替代关联容器。

标准关联容器通常被实现为平衡的二叉查找树。也就是说，它所适合的那些应用程序首先做一些插入操作，然后做查找，然后可能又插入一些元素，或许接着删掉一些，随后又做查找，等等。这一系列时间的主要特征是插入、删除和超找混在一起。总的来说，没办法预测出针对这颗树的下一个操作是什么。

很多应用程序使用其数据结构的方式并不这么混乱。他们使用其数据结构的过程可以明显地分为三个阶段，总结如下：

1. 设置阶段。创建一个新的数据结构，并插入大量元素。在这个阶段，几乎所有的操作都是插入和删除操作。很少或几乎没有查找操作。
2. 查找操作。查询该数据结构以找到特定的信息。在这个阶段，几乎所有的操作都是查找操作，很少或几乎没有插入和删除操作。
3. 重组阶段。改变该数据结构的内容，或许是删除所有的当前数据，再插入新的数据。在行为上，这个阶段与第 1 阶段类似。当这个阶段结束以后，应用程序又回到第 2 阶段。

第 24 条：当效率至关重要时，请在 map::operator[]与 map::insert 之间谨慎作出选择。

假定我们有一个 Widget 类，它支持默认构造函数，并根据一个 double 值来构造和赋值：

```
class Widget{
public:
Widget();
Widget(double weight);
Widget& operator=(double weight);
```

...
}

map 的 operator[]函数与众不同。它与 vector、deque 和 string 的 operator[]函数无关，与用于数组的内置 operator []也没有关系。相反，map::operator[]的实际目的是为了提供“添加和更新”(add or update)的功能。也就是说，对于下面的例子：
map<int, Widget> m;
m[1] = 1.50;
语句 m[1] = 1.50 相当于
typedef map<int, Widget> IntWidgetMap;
pair<INtWidgetMap::iterator, bool> result =
m.insert(IntWidgetMap::value_type(1.Widget()));//用键值 1 和默认构造的值创建一个新的 map 条目
result.first->second = 1.50;//调用赋值构造函数
我们最好把对 operator[]的调用换成对 insert 的直接调用：
m.insert(IntWidgetMap::value_type(1,1.50));
这里的效果和前面的代码相同，只是它通常会节省三个函数调用：一个用于创建默认构造的临时 Widget 对象，一个用于析构该临时对象，另一个是调用 Widget 的赋值操作符。
请看一下做更新操作时我们的选择：
m[k] = v; //使用 operator[]把 k 的值更新为 v
m.insert(IntWidgetMap::value_type(k,v)).first->second = v; //使用 insert 把 k 的值更新为 v
insert 调用需要一个 IntWidgetMap::value_type 类型的对象，所以当我们调用 insert 时，我们必须构造和西沟一个该类型的对象。这要付出一个 pair 构造函数和一个 pair 析构函数的代价。而这又会导致对 Widget 的构造和析构动作，因为 pair<int, Widget>本身又包含了一个 Widget 对象。而 operator[]不使用 pair 对象，所以它不会构造和析构 pair 或 Widget。
如果要更新一个已有的映射表元素，选择 operator[]；如果要添加一个新的元素，选择 insert。

第 25 条：熟悉非标准的哈希容器。

标准 C++库没有任何哈希容器，每个人认为这是一个遗憾，但是 C++标准委员会认为，把它们加入到标准中所需的工作会拖延标准完成的时间。已经有决定要在标准的下一个版本中包含哈希容器。

第 4 章 迭代器

第 26 条：iterator 优先于 const_iterator、reverse_iterator 以及 const_reverse_iterator。

减少混用不同类型的迭代器的机会，尽量用 iterator 代替 const_iterator。从 const 正确性的角度来看，仅仅为了避免一些可能存在的 STL 实现缺陷而放弃使用 const_iteraor 显得有欠公允。但考虑到在容器类的某些成员函数中指定使用

iterator 的现状，得出 iterator 较之 const_iterator 更为实用的结论也就不足为奇了。更何况，从实践的角度来看，并不总是值得卷入 const_iterator 的麻烦中。

第 27 条：使用 distance 和 advance 将容器的 const_iterator 转换成 iterator。

下面的代码试图把一个 const_iterator 强制转换为 iterator：

```
typedef deque<int> IntDeque; //类型定义，简化代码
typedef IntDeque::iterator Iter;
typedeef IntDeque:;const_iterator ConstIter;

ConstIter ci; //ci 是一个 const_iterator
...
Iter i(ci); //编译错误！从 const_iterator 到 iterator 没有隐式转换途径

Iter i(const_cast<Iter>(ci)); //仍然是编译错误！不能将 const_iterator 强制转换为 iterator
```

包含显式类型转换的代码不能通过编译的原因在于，对于这些容器类型，iterator 和 const_iterator 是完全不同的类，他们之间的关系甚至比 string 和 complex<double>之间的关系还要远。

下面是这种方案的本质。

```
typedef deque<int> IntDeque; //类型定义，简化代码
typedef IntDeque::iterator Iter;
typedeef IntDeque:;const_iterator ConstIter;

IntDeque d;
ConstIter ci; //ci 是一个 const_iterator
... //使 ci 指向 d
Iter i(d.begin());//使 i 指向 d 的起始位置
advance(i,distance<ConstIter>(i,ci));//移动 i，使它指向 ci 所指的位置
```

这中方法看上去非常简单和直接，也很令人吃惊。为了得到一个与 const_iterator 指向同一位置的 iterator，首先创建一个新的 iterator，将它指向容器的起始位置，然后取得 const_iterator 距离容器起始位置的偏移量，并将 iterator 向前移动相同的偏移量即可。这项技术的效率取决于你所使用的迭代起，对于随机迭代器，它是常数时间的操作；对于双向迭代器，以及某些哈希容器，它是线性时间的操作。

第 28 条：正确理解由 reverse_iterator 的 base()成员函数所产生的 iterator 的用法。

如果要在一个 reverse_iterator ri 指定的位置上插入元素，则只需在 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。

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()前一个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。

我们还是有必要来看一看执行这样一个删除操作的实际代码，其中暗藏着惊奇之处：

```
vector<int> v;
```

... //同上，插入 1 到 5
vector<int>::reverse_iterator ri = find(v.rbegin(),v.rend(),3);//使 ri 指向 3
v.erase(--ri.base()); //试图删除 ri.base()前面的元素，对于 vector，往往编译通不过
对于 vector 和 string，这段代码也许能工作，但对于 vector 和 string 的许多实现，它无法通过编译。这是因为在这样的实现中， iterator(和 vconst_iterator)是以内置指针的方式实现的，所以 ri.base()的结果是一个指针。C 和 C＋＋都规定了从函数返回的指针不应该被修改，所以所以编译不能通过。
既然不能对 base()的结果做递减操作，那么只要先递增 reverse_iterator，然后再调用 base()函数即可！
...
v.erase((++ri).base()); //删除 ri 所指的元素，这下编译没问题了！

第 29 条：对于逐个字符的输入请考虑使用 istreambuf_iterator。

假如你想把一个文本文件的内容拷贝到一个 string 对象中，以下的代码看上去是一种合理的解决方案：
ifstream inputFile("interestingData.txt");
inputFIle.unsetf(ios::skipws);//istream_iterator 使用 operator>>函数来完成实际的读操作，而默认情况下 operator>>函数会跳过空白字符
string fileData((istream_iterator<char> (inputFIle)),istream_iterator<char>());
然而，你可能会发现整个拷贝过程远不及你希望的那般快。istream_iterator 内部使用的 operator>>实际上执行了格式化的输入，但如果你只是想从输入流中读出下一个字符的话，它就显得有点多余了。
有一种更为有效的途径，那就是使用 STL 中最为神秘的法宝之一：istreambuf_iterator。 istreambuf_iterator<char>对象使用方法与 istream_iterator<char>大致相同，但是 istreambuf_iterator<char>直接从流的缓冲区读取下一个字符。（更为特殊的是， istreambuf_iterator<char>对象从一个输入流 istream s 中读取下一个字符的操作是通过 s.rdbuf()->sgetc()来完成的。）
ifstream inputFile("interestingData.txt");
string fileData((istreambuf_iterator<char>(inputFile)),istreambuf_iterator<char>());
这次我们用不着清楚输入流的 skipws 标志，因为 istreambuf_iterator 不会跳过任何字符。
同样的，对于非格式化的逐个字符输出过程，你也应该考虑使用 ostreambuf_iterator。

第 5 章 算法

第 30 条：确保目标区间足够大。

当程序员希望向容器中添加新的对象，这里有一个例子：
int transmogrify(int x); //该函数根据 x 生成一个新的值
vector<int> values;
vector<int> results;
transform(values.begin(),values.end(),back_inserter(results),transmogrify);

back_inserter 返回的迭代起将使得 push_back 被调用，所以 back_inserter 可适用于所有提供了 push_back 方法的容器。同理，front_inserter 仅适用于那些提供了 push_front 成员函数的容器（如 deque 和 list）。

当是使用 reserver 提高一个序列插入操作的效率的时候，切记 reserve 只是增加了容器的容量，而容器的大小并未改变。当一个算法需要向 vector 或者 string 中加入新的元素，即使已经调用了 reserve，你也必须使用插入型的迭代器。如下代码给出了一种错误的方式：

```
vector<int> values;
vector<int> results;
...
results.reserve(results.size() + values.size());
transform(values.begin(), values.end(), results.end(), transmogrify);//变换的结果会写入到尚未初始化的内存，结果将是不确定的
```

在以上代码中 transform 欣然接受了在 results 尾部未初始化的内存中进行复制操作的任务。由于赋值操作重视在两个对象之间而不是在一个对象与一个未初始化的内存块之间进行，所以一般情况下，这段代码在运行时会失败。

假设希望 transform 覆盖 results 容器中已有的元素，那么就需要确保 results 中已有的元素至少和 values 中的元素一样多。否则，就必须使用 resize 来保证这一点。

```
vector<int> values;
vector<int> results;
...
if(results.size() < values.size()){
results.resize(values.size());
}
transform(values.begin(),values.end(),results.begin(),transmogrify);
```

或者，也可以先清空 results，然后按通常的方式使用一个插入型迭代起：

```
...
results.clear();
results.reserve(values.size());
transform(values.begin(),values.end(),back_inserter(results),transmogrify);
```

第 31 条：了解各种与排序有关的选择。

sort(stable_sort)、partial_sort 和 nth_element 算法都要求随即访问迭代器，所以这些算法只能被应用于 vector、string、deque 和数组。partion（stable_partion）只要求双向迭代器就能完成工作。对于标准关联容器中的元素进行排序并没有实际意义，因为它们总是使用比较函数来维护内部元素的有效性。list 是唯一需要排序却无法使用这些排序算法的容器，为此，list 特别提供了 sort 成员函数（有趣的是，list::sort 执行的是稳定排序）。如果希望希望一个 list 进行完全排序，可以用 sort 成员函数；但是，如果需要对 list 使用 partial_sort 或者 nth_element 算法的话，你就只能通过间接途径来完成了。一种间接做法是，将 list 中的元素拷贝到一个提供随即访问迭代器的容器中，然后对该容器执行你所期望的算法；另一种简介做法是，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；第三中方法是利用一个包含迭代器的有序容器的信息，通过反复地调用 splice 成员函数，将 list 中的元素调整到期望的目标位置。可以看到，你会有很多中选择。

第 32 条：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 2 3 99 5 99 7 8 9 99
调用 remove(v.begin(),v.end(),99)；后变成
1 2 3 5 7 8 9 8 9 99
remove 无法从迭代器推知对应的容器类型，所以就无法调用容器的成员函数 erase，因此就无法真正删除元素。其他两个算法 remove_if 和 unique 也类似。不过 list::remove 和 list::unique 会真正删除元素（比用 erase-remove 和 erase-unique 更为高效），这是 STL 中一个不一致的地方。

第 33 条：对包含指针的容器使用 remove 这一类算法时要特别小心。

无论你如何处理那些存放动态分配的指针的容器，你总是可以这样来进行：或者调用 remove 类算法之前先手工删除指针并将它们置为空，或者通过引用计数的智能指针(如 boost::shared_ptr)，或者你自己发明的其他某项技术。
下面的代码利用第一种方式：

```
void delAndNullifyUncertified(Widget*& pWidget)
{
if(!pWidget->isCertified())
{
delete pWidget;
pWidget = 0;
}
}
for_each(v.begin(),v.end(),delAndNullifyUndertified);
v.erase(vemove(v.begin(),v.end(),static_cast<Widget*>(0)),v.end());
```

下面的的代码使用第二中方式：

```
template<typename T> //RSCP = "Reference Counting Smart Pointer"
class RCSP{...};
tpedef RCSP<Widget> RCSPW;
vector<RCSPW> v;
...
v.push_back(RCSPW(new Widget));
...
v.erase(remove_if(v.begin(),v.end(),not1(mem_fun(&Widget::isCertified))),v.end());
```

第 34 条：了解哪些算法要求使用排序的区间作为参数。

下面的代码要求排序的区间：
binary_search lower_bound
upper_bound equal_range

set_union set_intersection
set_difference set_symmetric_difference

merge inplace_merge

includes

下面的算法并不一定需要排序的区间：
unique unique_copy

第 35 条：通过 mismatch 或 lexicographical_compare 实现简单的忽略大小写的字符串比较。

用 mistatch 实现：

```
//此函数判断两个字母是否相同，而忽略它们的大小写
int ciCharCompare(char c1, char c2)
{
    int lc1 = tolower(static_cast<unsigned_char>(c1));
    int lc2 = tolower(static_cast<unsigned_char>(c2));

    if(lc1 < lc2) return -1;
    if(lc1 > lc2) return 1;
    return 0;
}
/* 此函数保证传递给 ciStringCompareImpl 的 s1 比 s2 短，如果 s1 和 s2 相同，返回 0；如果 s1 比 s2 短，返回-1；如果 s1 比 s2 长，返回 1。*/
int ciStringCompare(const string& s1, const string& s2)
{
    if(s1.size() <= s2.size()) return ciStringCompareImpl(s1, s2);
    else return – ciStringCompareImpl(s2, s1);
}
//如果 s1 和 s2 相同，返回 0；如果 s1 比 s2 短，返回-1；如果 s1 和 s2 都是在非结尾处发生不匹配，有开始不匹配的那个字符决定。
int ciStringCompareImpl(const string &s1, const string &c2)
{
    typedef pair<string::const_iterator,string::const_iterator> PSCI;
    PSCI p =
mismatch(s1.begin(),s1.end(),s2.begin(),not2(ptr_fun(ciCharCompare)));
    if(p.first == s1.end()){
    if(p.second == s2.end()) return 0;
    else return -1;
    }
    return ciCharCompair(*p.first, *p.second);
}
```

用 lexicographical_compare 实现：

```
bool ciCharLess(char c1, char c2)
{
return tolower(static_cast<unsigned char>(c1)) < tolower(static_cast<unsigned
char>(c2));
}
bool ciStringCompare(const string &s1,const string &s2)
{
return lexicographical_compare(s1.begin(), s1.end(), s2.begin(), s2.end(),
ciCharLess);
}
```

第 36 条：理解 copy_if 算法的正确实现。

STL 中没有 copy_if 的算法，下面是一个实现，但是不够完美：

```
template<typename INputIterator,typename OUtputIterator,tpename Predicate>
OutputIterator copy_if(INputIterator begin,INputIterator end,OutputIterator
destBegin,Predicate p)
{
return remove_copy_if(begin,end,destBegin, not1(0));
}
copy_if(widgets.begin(), widgets.end(), ostream_iterator<Widget>(cerr,
"\n"),isDefective);//编译错误
```

因为 not1 不能被直接应用到一个函数指针上（见 41 条），函数指针必须先用 ptr_fun 进行转换。为了调用 copy_if 的这个实现，你传入的不仅是一个函数对象，而且还应该是一个可配接（adaptable）的函数对象。虽然这很容易做到，但是要想成为 STL 算法，它不能给客户这样的负担。

下面是 copy_if 的正确实现：

```
template<typename INputIterator,typename OUtputIterator,typename Predicate>
OutputIterator copy_if(INputIterator begin,INputIterator end,OutputIterator
destBegin,Predicate p)
{
while(begin != end){
if(p(*begin)) *destBegin++ = *begin;
++begin;
}
return destBegin;
}
```

第 37 条：使用 accumulate 或者 for_each 进行区间统计。

确保 accumulate 的返回类西和初始值类型相同。for_each 返回的是一个函数对象。accumulate 不允许副作用而 for_each 允许。（这是一个深层次的问题，也是一个涉及 STL 核心的问题，待解）

第 6 章 函数子、函数子类、函数及其他

第 38 条：遵循按值传递的原则来设计函数子类。

在 STL 中，函数对象在函数之间来回传递的时候也是像函数指针那样按值传递的。因此，你的函数对象必须尽可能的小，否则拷贝的开销会很大；其次，函数对象必须是单态的，也就是说，它们不得使用虚函数。这是因为，如果参数的类型是基类类型，而实参是派生类对象，那么在传递过程中会产生剥离问题（slicing problem）：在对象拷贝过程中，派生部分可能会被去掉，而仅保留了基类部分（见第 3 条）。

试图禁止多态的函数子同样也是不实际的。所以必须找到一种两全其美的办法，既允许函数对象可以很大并且 / 或保留多态性，又可以与 STL 所采用的按值传递函数子的习惯保持一致。这个办法就是：将所需要的数据和虚函数从函数子中分离出来，放到一个新的类中，然后在函数子中设一个指针，指向这个新类。

第 39 条：确保判别式是“纯函数”。

一个判别式（predicate）是一个返回值为 bool 类型的函数。一个纯函数（pure function）是指返回值仅仅依赖于其参数的函数。

因为接受函数子的 STL 算法可能会先创建函数子对象的拷贝，然后使用这个拷贝，因此这一特性的直接反映就是判别式函数必须是纯函数。

```
template<typename FwdIterator,typename Predicata>
FwdIterator remove_if(FwdIterator begin, FwdIterator end, Predicate p)
{
begin = find_if(begin, end, p);//可能是 p 的拷贝
if(begin == end return begin;
else{
FwdIterator next = begin;
return remove_copy_if(++next, end, begin, p);//可能是 p 的另一个拷贝
}
}
```

第 40 条：若一个类是函数子，则应使它可配接。

4 个标准的函数配接器(not1、not2、bind1st 和 bind2nd）都要求一些特殊的类型定义。提供了这些必要的类型定义（argument_type、first_argument_type、second_argument_type 以及 result_type）的函数对象被称为可配接的（adaptable）函数对象，反之，如果函数对象缺少这些类型定义，则称为不可配接的。可配接的函数对象能够与其他 STL 组件更为默契地协同工作。不过不同种类的函数子类所需要提供的类型定义也不尽相同，除非你要编写自定义的配接器，否则你并不需要知道有关这些类型定义的细节。这是因为，提供这些类型定义最简便的办法是让函数子从特定的基类继承，或者更准确的说，如果函数子类的 operator()只有一个形参，那么它应该从 std::unary_function 模板的一个实例继承；如果函数子类的 operator()有两个形参，那么它应该从 std:: binary_function 继承。

对于 unary_function，你必须指定函数子类 operator()所带的参数的类型，以及返回类型；对于 binary_function，你必须指定三个类型：operator()的第一个和第二个参数的类型，以及 operator()的返回类型。以下是两个例子：

```
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;
};
```

你可能已经注意到 MeetsThreshold 是一个类，而 WidgetNameCompare 是一个结构。这是因为 MeetsThreshold 包含了状态信息（数据成员 threshold），而类是封装状态信息的一种逻辑方式；与此相反，WidgetNameCompare 并不包含状态信息，因而不需要任何私有成员。如果一个函数子的所有成员都是公有的，那么通常会将其声明为结构而不是类。究竟是选择结构还是类来定义函数子纯属个人编码风格，但是如果你正在改进自己的编码风格，并希望自己的风格更加专业一点的话，你就应该注意到，STL 中所有无状态的函数子类（如 less<T>、 plus<T>等）一般都定义成结构。

我们在看一下 WidgetNameCompare：

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

虽然 operator()的参数类型都是 const Widget&，但我们传递给 binary_function 的类型却是 Widget。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（不要问其中的原因，如果你有兴趣，可以访问 boost.org，卡可能看他们在调用特性(traits)和函数对象配接器方面的工作）。

如果 operator()带有指针参数，规则又有不同了。下面是 WidgetNameCOmpare 函数子的另一个版本，所不同的是，这次以 Widget*指针作为参数：

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

第 41 条：理解 ptr_fun、mem_fun 和 mem_fun_ref 的来由。

如果有一个函数 f 和一个对象 x，现在希望在 x 上调用 f，而我们在 x 的成员函数之外，那么为了执行这个调用，C＋＋提供了三种不同的语法：

```
f(x); //语法#1：f 是一个非成员函数
x.f(); //语法#2：f 是一个成员函数，并且 x 是一个对象或一个对象引用
p->f(); //语法#3：f 是成员函数，并且 p 是一个指向对象 x 的指针
```

现在假设有个可用于测试 Widget 对象的函数：

```
void test(Widget& w);
```

另有一个存放 Widget 对象的容器：

```
vector<Widget> vw;
```

为了测试 vw 中的每一个 Widget 对象，自然可以用如下的方式来调用 for_each：

```
for_each(vw.begin(), vw.end(), test); //调用#1 (可以通过编译)
```

但是，加入 test 是 Widget 的成员函数，即 Widget 支持自测：

```
class Widget{
public:
...
void test();
....
};
```

那么在理想情况下，应该也可以用 for_each 在 vw 中的每个对象上调用 Widget::test 成员函数：
for_each(vw.begin(), vw.end(), &Widget::test);//调用#2（不能通过编译）
实际上，如果真的很理想的话，那么对于一个存放 Widget* 指针的容器，应该也可以通过 for_each 来调用 Widget::test：
list<Widget*> lpw;
for_each(lpw.begin(), lpw.end(), &Widget::test);//调用#3（也不能通过编译）
这是因为 STL 中一种和普遍的惯例：函数或函数对象在被调用的时候，总是使用非成员函数的语法形式（即#1）。
现在 mem_fun 和 mem_fun_ref 之所以必须存在已经很清楚了——它们被用来调整（一般是#2 和#3）成员函数，使之能够通过语法#1 被调用。 mem_fun、mem_fun_ref 的做法其实很简单，只要看一看其中任意一个函数的声明就清楚了。它们是真正的函数模板，针对它们所配接的成员函数的圆形的不同，有几种变化形式。我们来看其中一个声明，以便了解它是如何工作的：
template<typename R, typename C> //该 mem_fun 声明针对不带参数的非 const 成员函数，C 是类，R 是所指向的成员函数的返回类型
mem_fun_t<R,C>
mem_fun(R(C::*pmf) ());
mem_fun 带一个指向某个成员函数的指针参数 pmf，并且返回一个 mem_fun_t 类型的对象。mem_fun_t 是一个函数子类，它拥有该成员函数的指针，并提供了 operator()函数，在 operator()中调用了通过参数传递进来的对象上的该成员函数。例如，请看下面一段代码：
list<Widget*> lpw;
...
for_each(lpw.begin(),lpw.end(),mem_fun(&Widget::test));//现在可以通过编译了
for_each 接受到一个类型为 mem_fun_t 的对象，该对象中保存了一个指向 Widget::test 的指针。对于 lpw 中的每一个 Widget*指针，for_each 将会使用语法#1 来调用 mem_fun_t 对象，然后，该对象立即用语法#3 调用 Widget*指针的 Widget::test()。
（ptr_fun 是多余的吗？）mem_fun 是针对成员函数的配接器，mem_fun_ref 是针对对象容器的配接器。

第 42 条：确保 less<T>与 operator<具有相同的含义。

operator<不仅仅是 less 的默认实现方式，它也是程序员期望 less 所做的事情。让 less 不调用 operator<而去坐别的事情，这会无端地违背程序员的意愿，这与“少”带给人惊奇的原则（the principle of least astonishment）完全背道而驰。这是很不好的，你应该尽量避免这样做。
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，它的名字不能是 less。

第 7 章 在程序中使用 STL

第 43 条：算法调用优于手写的循环。

有三个理由：
效率：算法通常比程序员自己写的循环效率更高。
STL 实现者可以针对具体的容器对算法进行优化；几乎所有的 STL 算法都使用了复杂的计算机科学算法，有些科学算法非常复杂，并非一般的 C＋＋程序员所能够到达。
正确性：自己写的循环比使用算法容易出错。
比如迭代器可能会在插入元素后失效。
可维护性：使用算法的代码通常比手写循环的代码更加简介明了。
算法的名称表明了它的功能，而 for、while 和 do 却不能，每一位专业的 C＋＋程序员都应该知道每一个算法所做的事情，看到一个算法就可以知道这段代码的功能，而对于循环只能继续往下看具体的代码才能懂代码的意图。

第 44 条：容器的成员函数优先于同名的算法。

第一：成员函数往往速度快；第二，成员函数通常与容器（特别是关联容器）结合得更紧密（相等和等价的差别，比如对于关联容器，count 只能使用相等测试）。

第 45 条：正确区分 count、find、binary_search、lower_bound、upper_bound 和 equal_range。

想知道什么	使用算法		使用成员函数	
	对未排序的区间	对排序的区间	对 set 或 map	对 multiset 或 multimap
特定的值存在吗	find	binary_search	count	find
特定的值存在吗？如果有，第一个在哪里	find	equal_range	find	find 或 lower_bound
第一个不超过特定值的对象在哪里	find_if	lower_bound	lower_bound	lower_bound
第一个超过某个特定值的对象在哪里	find_if	upper_bound	upper_bound	upper_bound
具有特定值的对象有多少个	count	equal_range (然后 distance)	count	count
具有特定值的对象都在哪里	find(反复调用)	equal_range	equal_range	equal_range

第 46 条：考虑使用函数对象而不是函数指针作为 STL 算法的参数。

函数指针抑制了内联机制，而函数对象可以被编译器优化为内联。

另一个理由是，这样做有助于避免一些微妙的、语言本身的缺陷。在偶然的情况下，有些看似合理的代码会被编译器以一些合法但又含糊不清的理由而拒绝。例如，当一个函数模板的实例化名称并不完全等同于一个函数的名称时，就可能会出现这样的问题。下面是一个例子：

```
template<typename FPType>
FPType average(FPType val1, FPType val2)//返回两个浮点的平均值
{
return (val1 + val2) / 2;
}
template<typename InputIter1,typename InputIter2>
void writeAverages(InputIter1 begin1, INputIter1 end1, InputIter2 begin2,ostream& s)
//将两个序列的值按顺序对应取平均，然后写到一个流中
{
transform(begin1, end1, begin2,
ostream_iterator<typename iterator_trais<InputIterl>::value_type(s,"\n")>,
average<typename iterator_traits<InputIterl>::value_type>   //错误？
);
}
```

许多编译器接受这段代码，但是 C ＋＋标准却不认同这样的代码。原因在于，理论上存在另一个名为 average 的函数模板，它也只带一个类型参数。如果这样的话，表达式 average<typename iterator_traits<InputIterl>::value_type>就会有二义性，因为编译器无法分辨到底应该实例化哪一个模板。换成函数对象就可以了。

第 47 条：避免产生“直写型”（write-only）的代码。

代码被阅读的次数远远大于它被编写的次数。

第 48 条：总是包含（#include）正确的头文件。

几乎所有的标准 STL 容器都被声明在与之同名的头文件中。

除了 4 个 STL 算法外，其他所有的算法都被声明在<algorithm>中，这 4 个算法是 accumulate、 inner_product、adjacent_difference 和 partial_sum，它们都被声明在头文件<numeric> 中。

特殊类型的迭代器，包括 istream_iterator 和 istreambuf_iterator（见第 29 条），被声明在<iterator>中。

标准的函数子（比如 less<T>）和函数子配接器（比如 not1、bind2nd）被声明在头文件<functional>中。

第 49 条：学会分析与 STL 相关的编译器诊断信息。

用文本替换（例如用 string 替换掉 basic_string<char,struct std::char_traits<char>,class std::allocator<char> >）。