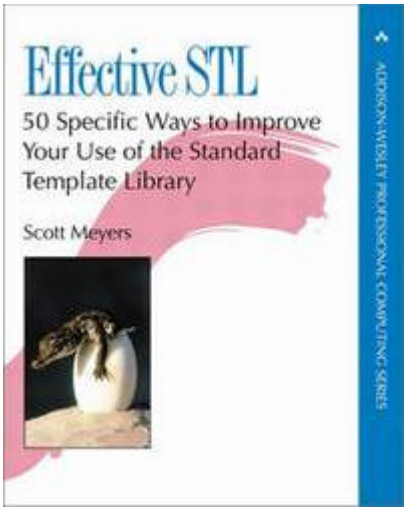
Effective STL



不需要看 Amazon 上的读者打分，也不需要去找 C++专家们的书评。单是似曾相识的封面，似曾相识的书名，还有似曾相识的作者，就足以证明它不会让你失望了。

这是 Scott Meyers 的第三本 C++专著，在以其特有的简练而敏锐的笔锋揭示了如何有效使用 C++语言本身的种种规则之后，Scott 将注意力转移到了 C++标准库中的 STL 部分。

Effective STL 的各章节不同程度的涉及 STL 的六大构成要件。千万不要误以为这是一本 STL 使用手册或者初学者指南，与这一系列的前两本一脉相承，Effective STL 同样是为具备一定 STL 使用经验的程序员准备的。

提供了类型安全、高效而易用特性的 STL 无疑是最值得 C++程序员骄傲的部分。使用 STL 可以使你的代码更为优雅且易于维护，可以让你彻底的从 C 语言风格中解脱出来。然而，STL 秉承了 C++的设计风格：高效甚至正确性仍然依赖于程序员的正确抉择。如何正确使用 STL（尤其是当性能问题至关重要的时候），对广大 C++程序员来说仍是一种挑战。

也许你的 C++代码已经充斥着 vector、string 甚至 multimap，也许你正在为此而自鸣得意，然而，你是否真的了解 STL 呢？你知道如何在各种容器类之间做出正确的选择吗？你知道如何最为有效的从一个 vector 中删除特定值的元素吗？你知道关联容器中“等值”与“等价”的区别吗？先别急着回答，Scott 会让你后悔的，我保证。

有些书是每个 C++程序员都必须拥有的，Effective STL 正是其中的一本。
—— Thomas Becker， C/C++ Users Journal 专栏作家

以下文章（有效使用 STL 迭代器的三条基本原则）是 C/C++ Users Journal 摘录的 Effective STL 的三个独立条款。你可以在 http://www.aristeia.com 找到英文原文。

关于 Scott 的前两本 C++专著：Effective C++和 More Effective C++，可以参见侯捷先生的书评：掷地铿锵的三本 OOP 小书。

It came without ribbons!
It came without tags! It came without packages, boxes or bags!
——Dr. Seuss, *How the Grinch Stole Christmas!*, Random House, 1957

我第一次写关于 Standard Template Library 的东西是在 1995 年，那时，我决定把 More Effective C++的最后一个条款写成一个 STL 的简要概览。我早该更好地了解 STL。不久以后，我开始收到一些 mail，问我什么时候写 Effective STL。

我把这个想法忍耐了几年。一开始，我对 STL 不够熟悉，所以不能给出关于它的建议。但随着时间的推移，我的 STL 的经验丰富了，主要问题出在其他方面。当一个程序库的在效率和可扩展性设计上表现出突破性的时候从来没有出过什么问题，但当开始使用 STL 时，这成了我不能预见的实际问题。迁移到一个几乎最简单的 STL 程序都成了一个挑战，不光是因为库的实现变化多端，而且因为现有的编译器对模板支持有好有坏。STL 的教材很难得到，所以学习“用 STL 方式编程”很难；但即使跨越了这个障碍，找到正确易学的参考文档同样很困难。可能使人畏惧的是，即使最小的 STL 使用错误往往会导致一个编译器诊断的风暴——每一个错误都有上千个字长，而且大多涉及的类，函数或模板在令人厌恶的源代码中并没有被提及——几乎都是难以理解的。虽然我很钦佩 STL 和它背后的英雄们，但我还是觉得把 STL 推荐给在业的程序员并不合适。我不能肯定能有效率地使用 STL。

然后我开始注意到一些让我感到惊奇的事情。尽管有很多小问题，尽管只有令人消沉的文档，尽管编译器的出错信息像无线电信号杂音，但仍然有很多我的咨询客户在使用 STL。而且，他们不只是玩玩而已，他们竟然把 STL 用到了产品的代码中！这是一个革命。我知道 STL 表现出的是一流的设计，但程序员是不会喜欢用“必须忍耐轻微头痛，只有贫乏的文档和天书般的错误信息，但设计得很好”的程序库的。我了解到越来越多的专业程序员都认为即使一个实现得很不好的 STL 也比什么都没有好得多。

此外，我知道关于 STL 的境遇只会越来越好。程序库和编译器对（它们的）标准的兼容性会越来越好，更好的文档将会出现（它已经存在了——请见从 297 页开始的“参考书目”），而且编译器的诊断会渐渐改进（在极大程度上，我们仍然在等待，但条款 49 提供了怎样在其间应付的建议）。因此我决定插嘴，尽一份力量来支持 STL 运动的萌芽。这本书就是结果：改善使用 C++ STL 的 50 个有效做法。

一开始，我计划在 1999 年下半年写这本书。带着这个想法，我组织了一个大纲。但我暂停和改变了进程。我停止了写书的工作，开发了一个介绍性的 STL 训练课程，把它教给几拨不同的程序员。大约一年后，我回到写书的工作中，根据我在训练课程中得到的经验意味深长地修改了大纲。和我的 Effective C++成功的方法一样，它们都是以真正的程序员所面对的问题为基础的。我希望 Effective STL 同样从事于 STL 编程的实践方面——这是对专业开发人员最重要的方面。

我一直在寻找着能让我加深对 C++理解的方法。如果你对 STL 编程有新的建议或者如果你对这本书有什么评论的话，请让我知道。另外，让这本书尽可能的正确是我的继续的目标，所以如果谁挑出了这本书的任何一个错误请务必告诉我——不论是技术、文法、错别字、或任何其他东西——我将在本书再次印刷的时候，把第一位挑出错误的读者大名加到致谢名单中。请将你的建议、见解、批评发至 estl@aristeia.com。

我维护有本书第一次印刷以来的修订记录，其中包括错误更正、文字修润、以及技术更新。这份记录可以从 Effective STL 的勘误表网站 http://www.aristeia.com/BookErrata/estl1e-errata.html 得到。

如果你希望在我对此书作出修改时得到通知，我想你应该加入我的邮件列表。我用这个列表来通知对我的 C++工作感兴趣的人。详情请见 http://www.aristeia.com/MailingList/。

SCOTT DOUGLAS MEYERS
STAFFORD, OREGON
http://www.aristeia.com/
APRIL 2001

条款 2: 小心对“容器无关代码”的幻想

STL 是建立在泛型之上的。数组泛化为容器，参数化了所包含的对象的类型。函数泛化为算法，参数化了所用的迭代器的类型。指针泛化为迭代器，参数化了所指向的对象的类型。

这只是个开始。独立的容器类型泛化为序列或关联容器，而且类似的容器拥有类似的功能。标准的内存相邻容器（参见条款 1）都提供随机访问迭代器，标准的基于节点的容器（再参见条款 1）都提供双向迭代器。序列容器支持 push_front 或 push_back，但关联容器不支持。关联容器提供对数时间复杂度的 lower_bound，upper_bound 和 equal_range 成员函数，但序列容器却没有。

随着泛化的继续，你会自然而然地想加入这个运动。这种做法值得赞扬，而且当你写你自己的容器、迭代器和算法时，你会自然而然地推行它。唉，很多程序员试图把它推行到不同的样式。他们试图在他们的软件中泛化容器的不同，而不是针对容器的特殊性编程，以至于他们可以用，可以说，现在是一个 vector，但以后仍然可以用比如 deque 或者 list 等东西来代替——都可以在不用改变代码的情况下来使用。也就是说，他们努力去写“容器无关代码”。这种可能是出于善意的泛化，却几乎总会造成麻烦。

最热心的“容器无关代码”的鼓吹者很快发现，写既要和序列容器又要和关联容器一起工作的代码并没有什么意义。很多成员函数只存在于其中一类容器中，比如，只有序列容器支持 push_front 或 push_back，只有关联

容器支持 count 和 lower_bound，等等。在不同种类中，甚至连一些如 insert 和 erase 这样简单的操作在名称和语义上也是天差地别的。举个例子，当你把一个对象插入一个序列容器中，它保留在你放置的位置。但如果你把一个对象插入到一个关联容器中，容器会按照的排列顺序把这个对象移到它应该在的位置。举另一个例子，在一个序列容器上用一个迭代器作为参数调用 erase，会返回一个新迭代器，但在关联容器上什么都不返回。（条款 9 给了一个例子来演示这点对你所写的代码的影响。）

假设，然后，你希望写一段可以用在所有常用的序列容器上——vector, deque 和 list——的代码。很显然，你必须使用它们能力的交集来编写，这意味着不能使用 reserve 或 capacity（参见条款 14），因为 deque 和 list 不支持它们。由于 list 的存在意味着你得放弃 operator[]，而且你必须受限于双向迭代器的性能。这意味着你不能使用需要随机访问迭代器的算法，包括 sort，stable_sort，partial_sort 和 nth_element（参见条款 31）。

另一方面，你渴望支持 vector 的规则，不使用 push_front 和 pop_front，而且用 vector 和 deque 都会使接合方式和成员函数方式的 sort 彻底失败。在上面约束的联合下，后者意味着你不能在你的"泛化的序列容器"上调用任何一种 sort。

这是显而易见的。如果你冒犯里其中任何一条限制，你的代码会在至少一个你想要使用的容器配合时发生编译错误。可见这种代码有多阴险。

这里的罪魁祸首是不同的序列容器所对应的不同的迭代器、指针和引用的失效规则。要写能正确地和 vector, deque 和 list 配合的代码，你必须假设任何使那些容器的迭代器，指针或引用失效的操作符真的在你用的容器上起作用了。因此，你必须假设每次调用 insert 都使所有东西失效了，因为 deque::insert 会使所有迭代器失效，而且因为缺少 capacity，vector::insert 也必须假设使所有指针和引用失效。（条款 1 解释了 deque 是唯一一个在使它的迭代取失效的情况下指针和引用仍然有效的东西）类似的理由可以推出一个结论，所有对 erase 的调用必须假设使所有东西失效。

想要知道更多？你不能把容器里的数据传递给 C 风格的界面，因为只有 vector 支持这么做（参见条款 16）。你不能用 bool 作为保存的对象来实例化你的容器，因为——正如条款 18 所阐述的——vector 并非总表现为一个 vector，实际上它并没有真正保存 bool 值。你不能期望享受到 list 的常数时间复杂度的插入和删除，因为 vector 和 deque 的插入和删除操作是线性时间复杂度的。

当这些都说到做到了，你只剩下一个"泛化的序列容器"，你不能调用 reserve，capacity，operator[]，push_front，pop_front，splice 或任何需要随机访问迭代器的算法；调用 insert 和 erase 会有线性时间复杂度而且会使所有迭代器、指针和引用失效；而且不能兼容 C 风格的界面，不能存储 bool。难道这真的是你想要在你的程序里用的那种容器？我想不是吧。

如果你控制住了你的野心，决定愿意放弃对 list 的支持，你仍然放弃了 reserve，capacity，push_front，and pop_front；你仍然必须假设所有对 insert 和 erase 的调用有线性时间复杂度而且会使所有东西失效；你仍然不能兼容 C 风格的布局；而且你仍然不能储存 bool。

如果你放弃了序列容器，把代码改为只能和不同的关联容器配合，这情况并没有什么改善。要同时兼容 set 和 map 几乎是不可能的，因为 set 保存单个对象，而 map 保存对象对。甚至要同时兼容 set 和 multiset（或 map 和 multimap）也是很难的。set/map 的 insert 成员函数只返回一个值，和他们的 multi 兄弟的返回类型不同，而且你必须避免对一个保存在容器中的值的拷贝份数作出任何假设。对于 map 和 multimap，你必须避免使用 operator[]，因为这个成员函数只存在于 map 中。

面对事实吧：这根本没有必要。不同的容器是不同的，而且它们的优点和缺点有重大不同。它们并不被设计成可互换的，而且你做不了什么包装的工作。如果你想试试看，你只不过是在考验命运，但命运并不想被考验。

接着，当天黑以后你认识到你决定使用的容器，嗯，不是最理想的，而且你需要使用一个不同的容器类型。你现在知道当你改变容器类型的时候，不光要修正编译器诊断出来的问题，而且要检查所有使用容器的代码，根据新容器的性能特征和迭代器，指针和引用的失效规则来看看那些需要修改。如果你从 vector 切换到其他东西，你也需要确认你不再依靠 vector 的 C 兼容的内存布局；如果你是切换到一个 vector，你需要保证你不用它来保存 bool。

既然有了要一次次的改变容器类型的必然性，你可以用这个常用的方法让改变得以简化：使用封装(encapsulating)，封装，再封装。其中一种最简单的方法是通过自由地对容器和迭代器类型使用 typedef。因此，不要这么写

```
class Widget { ... };
vector<Widget> vw;
Widget bestWidget;
...         // 给 bestWidget 一个值
vector<Widget>::iterator i =     // 寻找和 bestWidget 相等的 Widget
 find(vw.begin(), vw.end(), bestWidget);
```

要这么写：

```
class Widget { ... };
typedef vector<Widget> WidgetContainer;
typedef WidgetContainer::iterator WCIterator;
```

```
WidgetContainer vw;
Widget bestWidget;
...
WCIterator i = find(vw.begin(), vw.end(), bestWidget);
```

这是改变容器类型变得容易得多，如果问题的改变是简单的加上用户的 allocator 时特别方便。（一个不影响对迭代器/指针/参考的失效规则的改变）

```
class Widget { ... };
template<typename T>     // 关于为什么这里需要一个 template
SpecialAllocator { ... };    // 请参见条款 10
typedef vector<Widget, SpecialAllocator<Widget> > WidgetContainer;
typedef WidgetContainer::iterator WCIterator;
WidgetContainer vw;    // 仍然能用
Widget bestWidget;
...
WCIterator i = find(vw.begin(), vw.end(), bestWidget); // 仍然能用
```

如果 typedef 带来的代码封装作用对你来说没有任何意义的话，你仍然会称赞它们可以节省许多工作。比如，你有一个如下类型的对象

```
map<string,
 vectorWidget>::iterator,
 CIStringCompare>   // CIStringCompare 是“忽略大小写的字符串比较”
     // 参见条款 19
```

而且你要用 const_iterator 遍历这个map，你真的想不止一次地写下

```
map<string, vectorWidget>::iterator, CIStringCompare>::const_iterator
```

？当你使用 STL 一段时间以后，你会认识到 typedef是你的好朋友。

typedef 只是其它类型的同义字，所以它提供的的封装是纯的词法（译注：不像#define 是在预编译阶段替换的）。typedef 并不能阻止用户使用（或依赖）任何他们不应该用的（或依赖的）。如果你不想暴露出用户对你所决定使用的容器的类型，你需要更大的火力，那就是 class。

要限制如果用一个容器类型替换了另一个容器可能需要修改的代码，就需要在类中隐藏那个容器，而且要通过类的接口限制容器特殊信息可见性的数量。比如，如果你需要建立一个客户列表，请不要直接用 list。取而代之的是，建立一个 CustomerList 类，把 list 隐藏在它的 private 区域：

```
class CustomerList {
private:
 typedef list<Customer> CustomerContainer;
 typedef CustomerContainer::iterator CCIterator;
 CustomerContainer customers;
public:       // 通过这个接口
 ...     // 限制 list 特殊信息的可见性
};
```

一开始，这样做可能有些无聊。毕竟一个 customer list 是一个 list，对吗？哦，可能是。稍后你可能发现从列表的中部插入和删除客户并不像你想象的那么频繁，但你真的需要快速确定客户列表顶部的 20%——一个 nth_element 算法特制的任务（参见条款31）。但 nth_element 需要随机访问迭代器，不能兼容 list。在这种情况下，你的客户"list"可能更应该用 vector 或 deque 来实现。

当你决定作这种更改的时候，你仍然必须检查每个 CustomerList 的成员函数和每个友元，看看他们受影响的程度（根据性能和迭代器/指针/引用失效的情况等等），但如果你做好了对 CustomerList 地实现细节做好封装的话，那对 CustomerList 的客户的影响将会很小。你写不出容器无关性代码，但他们可能可以。

有效使用 STL 迭代器的三条基本原则

译注：原文中将 iterator、const_iterator、reverse_iterator 和 const_reverse_iterator 合称为 iterator。考虑到可能存在的歧义，译文中使用中文“*迭代器*”泛指四种类型，而特定的类型名称保持英文原文。

STL 迭代器的概念看上去似乎已经足够直观了，然而，你会很快发现容器类(Container)实际上提供了四种不同的迭代器类型：iterator、const_iterator、reverse_iterator 和 const_reverse_iterator。进而，你会注意到容器类的 insert 和 erase 方法仅接受这四种类型中的一种作为参数。问题来了：为什么需要四种不同的迭代器呢？它们之间存在何种联系？它们是否可以相互转换？是否可以在 STL 算法(Algorithm)和其他工具函数中混合使用不同类型的迭代器？ 这些迭代器与相应的容器类及其方法之间又是什么关系？

这篇从新近出版的《*Effective STL*》中摘录的文章将会通过迭代器使用的三条基本原则来回答上述问题，它们能够帮助你更为有效的使用 STL 迭代器。

条款 26: 尽量使用 iterator 取代 const_iterator、reverse_iterator 和 const_reverse_iterator

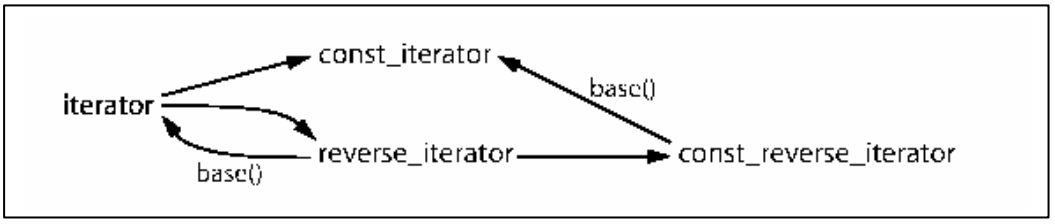
STL 中的所有标准容器类都提供四种不同的迭代器类型。对于容器类 container<T>而言，iterator 的功用相当于 T*，而 const_iterator 则相当于 const T*（可能你也见到过 T const *这样的写法，它们具有相同的语义[2]）。累加一个 iterator 或者 const_iterator 可以由首至尾的遍历一个容器内的所有元素。reverse_iterator 与 const_reverse_iterator 同样分别对应于 T*和 const T*，所不同的是，累加 reverse_iterator 或者 const_reverse_iterator 所产生的是由容器的尾端开始的反向遍历。

让我们先来看一下 vector<T>容器 insert 和 erase 方法的样式：

```
iterator insert(iterator position, const T& x);
iterator erase(iterator position);
iterator erase(iterator rangeBegin, iterator rangeEnd);
```

不同容器的 insert 和 erase 方法虽然可能具有截然不同的返回类型，但它们的参数形式却大都与此类似。需要注意的是：这些方法仅接受 iterator 类型的参数，而不是 const_iterator、reverse_iterator 或者 const_reverse_iterator。虽然容器类支持四种不同的迭代器类型，但其中 iterator 似乎有更为广泛的应用[3]。

下图清晰的表明了不同类型的迭代器之间的转换关系：



如图所示，你可以隐式的将 iterator 转换成 const_iterator 或者 reverse_iterator，也可以隐式的将 reverse_iterator 转换成 const_reverse_iterator。并且，reverse_iterator 可以通过调用其 base()成员函数转换为 iterator。const_reverse_iterator 也可以类似的通过 base()转换成为 const_iterator。然而，一个图中无法显示的事实是：通过 base()得到的也许并非你所期待的 iterator，我们将会在原则三中详细讨论这一点。

很显然，我们没有办法从一个 const_iterator 转换得到一个 iterator，也无法从 const_reverse_iterator 得到 reverse_iterator。这一点非常重要，因为这意味着当你仅仅得到一个 const_iterator 或者 const_reverse_iterator，你会在调用容器类的一些成员函数时遇到麻烦。这些成员函数要求 iterator 作为参数，而你无法从 const 类型的迭代器中直接得到 iterator。当需要指出插入或者删除的位置时，const 类型的迭代器总是显得那么力不从心。

千万不要傻乎乎的宣称 const 类型的迭代器一无是处，它们仍然可以在特定的场合使用。比如，const 类型的迭代器可以与 STL 算法默契配合——因为 STL 算法通常只关心迭代器属于何种概念范畴(Category)，而对其类型(Type)没有限制。很多容器类的成员方法也接受 const 类型的迭代器，只有 insert 和 erase 显得有些吹毛求疵。

> 译者注：原文中 Category 与 Type 都可译作类型或种类，而 Category 实际上是指迭代器隶属的概念(Concept)，如双向迭代器(Bidirectional Iterator)、随机访问迭代器(Random Access Iterator)。例如：reverse 算法要求输入可双向遍历的迭代器，而并不关心是 iterator 或者 reverse_iterator。为避免歧义，译文中将 Category 译为*概念范畴*。

即便是在需要插入或删除操作的时候，面对 const 类型的迭代器你也并非走投无路。一般情况下仍然有办法通过 const 类型的迭代器取得一个 iterator，但这种方法并不总是行得通。而且就算可行，它也显得复杂而缺乏效率。原则二中将会提及这种转换的方法。

现在，我们已经有足够的理由相信应该尽量使用 iterator 取代 const 或者 reverse 类型的迭代器：

- **大多数** insert **和** erase **方法要求使用** iterator**。如果你需要调用这些方法，你就必须使用** iterator**。**
- **没有一条简单有效的途径可以将** const_iterator **转换成** iterator**，我们将会在原则二中讨论的技术并不普遍适用，而且效率不彰。**
- **从** reverse_iterator **转换而来的** iterator **在使用之前可能需要相应的调整，在原则三种我们会讨论何时需要调整以及调整的原因。**

由此可见，尽量使用 iterator **而不是** const **或** reverse **类型的迭代器，可以使得容器的使用更为简单而有效，并且可以避免潜在的问题。**

在实践中，你可能会更多的面临 iterator **与** const_iterator **之间的选择，因为** iterator **与** reverse_iterator **之间的选择结果显而易见——依赖于顺序或者逆序的的遍历。而且，即使你选择了** reverse_iterator**，当需要** iterator **的时候，你仍然可以通过** base()**方法得到相应的** iterator**（可能需要一些调整，我们会在条款三中讨论）。**

而在 iterator **和** const_iterator **之间举棋不定的时候，你有更充分的理由选择** iterator**，即使** const_iterator **同样可行，即使你并不需要调用容器类的任何成员函数。其中的缘由包括** iterator **与** const_iterator **之间的比较，如下代码所示：**

```
typedef deque<int>     IntDeque;     //typedef 可以极大的简化
typedef IntDeque::iterator  Iter;    //STL 容器类和 iterator
typedef IntDeque::const_iterator ConstIter;      //的操作。

Iter  i;
ConstIter ci;
...                  //使 ci 和 i 指向同一容器。
if (i == ci) ...     //比较 iterator 和 const_iterator
```

我们所做的只是同一个容器中两个 iterator **之间的比较，这是** STL **中最为简单而常用的动作。唯一的变化是等号的一边是** iterator**，而另一边是** const_iterator**。这应该不是问题，因为** iterator **应该在比较之前隐式的转换成** const_iterator**，真正的比较应该在两个** const_iterator **之间进行。**

对于设计良好的 STL **实现而言，情况确实如此。但对于其它一些实现，这段代码甚至无法通过编译。原因在于，这些** STL **实现将** const_iterator **的等于操作符（**operator==**）作为** const_iterator **的一个成员函数而不是友元函数。而问题的解决之道显得非常有趣：只要交换两个** iterator **的位置，就万事大吉了。**

```
if (ci==i)...                  //问题的解决方法
```

不仅是比较是否相等，只要你在同一个表达式中混用 iterator **和** const_iterator**（或者** reverse_iterator **和** const_reverse_iterator**），这样的问题就会出现。例如，当你试图在两个随机存取迭代器之间进行减法操作时：**

```
if (i-ci >= 3) ...          //i 与 ci 之间有至少三个元素
```

如果迭代器的类型不同，你的完全正确的代码可能会被无理的拒绝。你可以想见解决的方法（交换 i 和 ci 的位置），但这次，不仅仅是互换位置了：

```
if (ci+3 <= i) ...          //问题的解决方法
```

避免类似问题的最简单的方法是减少混用不同类型的迭代器的机会，尽量以 iterator 取代 const_iterator。从 const correctness 的角度来看，仅仅为了避免一些可能存在的 STL 实现的弊端（而且，这些弊端都有较为直接的解决途径）而抛弃 const_iterator 显得有欠公允。但综合考虑到 iterator 与容器类成员函数的粘连关系，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 的麻烦当中去。

> 译者注：const correctness 是 OOP 中非常重要的概念，是设计思想在程序代码中的直接体现方式之一。例如：函数参数为 const 类型的引用标志着函数承诺不会改变其参数的值，const 成员函数则意味着类的设计者承诺该函数不会改变类对象的状态。同时，遵从 const correctness 也为编译器留下了更多的优化空间。参见 Effective C++条款 21。
>
> 如果你需要一个容器类对象作为你所定义的类成员，那么这个类的任何 const 方法就只能访问该容器的 const_iterator。抑或你的函数接受一个 const 类型的容器类的引用，你同样只能选择 const_iterator。（对于关联容器类如 set、map 而言，你仍然可以使用 iterator，这是因为这些类的 iterator 和 const_iterator 具有相同的定义）在这种情况下，也许奉行 const correctness 并且使用 const_iterator 才是正确的选择，这也正是 const_iterator 的设计初衷。

条款 27: 使用 distance 和 advance 将 const_iterator 转换成 iterator

原则一中指出容器类的部分方法仅接受 iterator 作为参数，而不是 const_iterator。那么，如何在一个 const_iterator 指出的容器位置上插入新元素呢？换言之，如何通过一个 const_iterator 得到指向容器中相同位置的 iterator 呢？由于并不存在从 const_iterator 到 iterator 之间的隐式转换，你必须自己找到一条转换的途径。

嗨，我知道你在想什么！“每当无路可走的时候，就祭起强制类型转换的大旗！”。在 C++的世界里，强制类型转换似乎总是最后的杀手锏。老实说，这恐怕算不上什么好主意——真不知道你是哪儿学来的。

让我们看看当你想把一个 const_iterator 强制转换为 iterator 时会发生什么：

```
typedef deque<int> IntDeque;  //typedef, 简化代码。
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

ConstIter ci;                 //const_iterator
Iter i(ci);                   //编译错误！
                              //没有隐式转换途径！
Iter i(const_cast<Iter>(ci)); //编译错误！
                              //转换不能成立！
```

这里只是以的 deque 为例，但是用其它容器类（list, set, multiset, map, multimap 甚至非标准 STL 的基于 hash 表的容器[4]）产生的代码大同小异。也许 vector 或 string 类的代码能够顺利编译，但这是非常特殊的情形。

包含显式类型转换的代码不能通过编译的原因在于，对于这些容器而言，iterator 和 const_iterator 是完全不同的类。它们之间并不比 string 和 complex<float>具有更多的血缘关系。编译器无法在两个毫无关联的类之间进行 const_cast 转换。reinterpret_cast、static_cast 甚至 C 语言风格的类型转换也不能胜任。

不过，对于 vector 和 string 容器来说，包含 const_cast 的代码也许能够通过编译。因为通常情况下 STL 的大多数实现都会采用真实的指针作为 vector 和 string 容器的迭代器。就这种实现而言，vector<T>::iterator 和 vector<T>::const_iterator 分别被定义为 T*和 const T*，string::iterator 和 string::const_iterator 则被定义为 char*和 const char*。因此，const_iterator 与 iterator 之间的 const_cast 转换被最终解释成 const T*到 T*的转换。当然，这样的类型转换没有问题。但是，即便在这种 STL 的实现当中，reverse_iterator 和 const_reverse_iterator 仍然是实在的类，所以仍然不能直接将 const_reverse_iterator 强制转换成 reverse_iterator。而且，这些 STL 实现为了方便调试通常只会在 Release 模式时才使用指针表示 vector 和 string 的迭代器[5]。所有这些事实表明，出于可移植性的考虑，这样的强制类型转换即使是对 vector 和 string 来说也不可取。

> 译者注：这种强制类型转换不能成立的根本原因在于 const_iterator 与 const iterator 具有完全不同的语义！const_iterator 暗示你无法通过此迭代器修改器所指向的容器的内容，但仍然可以用于遍历；而 const iterator 则表示你根本就不能修改该迭代器的值，换言之，不能调用它的累加方法遍历容器。

如果你得到一个 const_iterator 并且可以访问它所指向的容器，那么这里有一条安全、可移植的途径将它转换成 iterator，而且，用不着搅乱类型系统的强制转换。下面是基本的解决思路，称之为“思路”是因为还要稍作修改才能让这段代码通过编译。

```
typedef deque<int> IntDeque;
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;

IntDeque  d;
ConstIter ci;
...                                 //ci 指向 d。
Iter i(d.begin());                  //指向 d 的起始位置。
advance(i, distance(i, ci));        //调整 i，指向 ci 位置。
```

这种方法看上去非常简单，也很直观：要得到与 const_iterator 指向同一位置的 iterator，首先将 iterator 指向容器的起始位置，并且取得 const_iterator 距离容器起始位置的偏移量，然后将 iterator 向后移动相同的偏移即可。这些动作是通过<iterator>中声明的两个算法函数实现的：distance 用以取得两个指向同一个容器的 iterator 之间的距离；advance 则用于将一个 iterator 移动制定的距离。如果 ci 和 i 指向同一个容器，并且 i 指向容器的器时位置，表达式 advance(i, distance(i, ci))会将 i 移动到与 ci 相同的位置上。

如果这段代码能够通过编译，它就能完成这种转换任务。但似乎事情并不那么顺利。先来看看 distance 的定义：

```
template<typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

不必为长达 56 个字符的返回类型操心，也不用理会 difference_type 是什么东西。仔细看看调用的参数！编译器需要推断出它所遇到的 distance 调用的参数类型。再来看看我们的 distance 调用：

```
advance(i, distance(i, ci));      //调整 i，指向 ci 位置。
```

i 和 ci 分别是 distance 函数的两个参数，它们的类型分别是 deque<int>::iterator 和 deque<int>::const_iterator，而 distance 函数需要一种确定的参数类型，所以调用失败。也许相应的错误信息会告诉你编译器无法推断出 InputIterator 的类型。

要顺利的通过编译，你需要排除 distance 调用的歧义性。最简单的办法就是显式的指明 distance 调用的模版参数类型，从而避免编译器为此大伤脑筋。

```
advance(i, distance<ConstIter>(i, ci));
```

译者注：如果只是希望改变 const_iterator 所指向的元素的值，并不需要得到相应的 iterator 的话，这里有一种较为简单有效的方法：

```
IntDeque d;                          //typedef 同上
IntDeque::const_iterator ci;
...
const_cast<IntDeque::reference>(*ci) = 5;
```

这样，我们就可以通过 advance 和 distance 将一个 const_iterator 转换成 iterator 了。但另一个值的考虑的问题是，这样做的效率如何？答案在于你所转换的究竟是什么样的迭代器。对于随机存取的迭代器（如 vector, string 和 deque）而言，这种转换只需要一个与容器中元素个数无关的常数时间；对于双向迭代器（其它容器，包括基于 hash 表的非标准容器的一些实现[6]）而言，这种转换是与容器元素个数相关的线性时间操作。

译者注：这是因为双向迭代器无法通过简单的减法操作（ci – i)得出两个迭代器之间的 distance，而且也不能通过简单的加法操作将 i 后移若干元素。双向迭代器必须采用步进的方式进行上述计算。

这种从 const_iterator 到 iterator 的转换可能需要线性的时间代价，并且需要访问 const_iterator 所属的容器。从这个角度出发，也许你需要重新审视你的设计：是否真的需要从 const_iterator 到 iterator 的转换呢？

条款 28: 正确理解如何使用通过 base()得到的 iterator

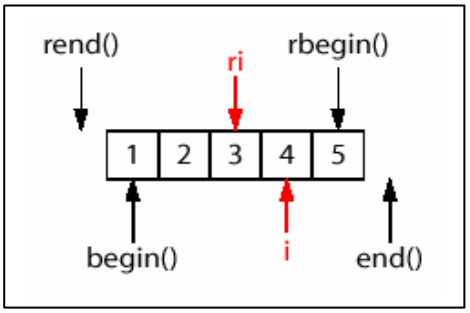
调用 reverse_iterator 的 base()方法可以得到“与之相对应的” iterator**。这句话也许有些辞不达意。还是先来看一下这段代码，我们首先把从 1 到 5 的 5 个数放进一个 vector 中，然后产生一个指向 3 的** reverse_iterator**，并且通过其 base()函数取得一个 iterator。**

```
vector<int> v;

for(int i = 0;i < 5; ++ i) {        //插入 1 到 5
    v.push_back(i);
}

vector<int>::reverse_iterator ri =      //使 ri 指向 3
    find(v.rbegin(), v.rend(), 3);
vector<int>::iterator i(ri.base());     //取得 i.
```

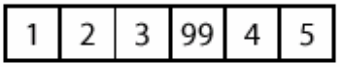
下图表示了执行上述代码之后的 i 和 ci 迭代器位置：



如图所示，rbegin()与 end()、rend()与 begin()之间存在一个元素的偏移，并且 ri 与 i 之间仍然保留了这种偏移。但这还远远不够，针对你所要进行的特定操作，你还需要知道一些技术细节。

原则一指出容器类的部分成员函数仅接受 iterator 类型的参数。上面的例子中，你并不能直接在 ri 所指出的位置上插入元素，因为 insert 方法不接受 reverse_iterator 作为参数。如果你要删除 ri 位置上的元素，erase 方法也有同样的问题。为了完成这种插入/删除操作，你必须首先用 base 方法将 reverse_iterator 转换成 iterator，然后用 iterator 调用 insert 或 erase 方法。

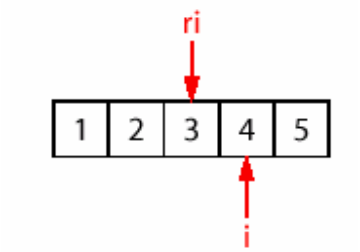
先让我们假设你要在 ri 指出的位置上进行插入操作，并且假设你要插入的值是 99。由于 ri 遍历 vector 的顺序是自右向左，而 insert 操作会将新元素插入到 ri 位置，并且将原先 ri 位置的元素移到遍历过程的“下一个”位置，插入操作之后，3 应该出现在 99 的左侧，如下图所示：



当然，这些只是我们的假设而已。insert 实际上并不接受 reverse_iterator 类型的参数，所以我们必须用 i 取代 ri。如上所述，在插入操作之前，ri 指向元素 3 而通过 base()得到的 i 指向元素 4。考虑到 insert 与遍历方向的关系，直接使用 i 进行 insert 操作，我们会得到与上述假设完全相同的结果。

- 如果要在一个 reverse_iterator 指出的位置上插入新元素，只需通过 base()得到相应的 iterator 然后调用 insert 方法即可。对于 insert 操作而言，reverse_iterator 与通过其 base()方法得到的 iterator 是完全等价的。

现在再来考虑删除元素的情况，先来回顾一下最初（没有进行 insert 操作）的 vector 的状态以及 i 与 ri 的位置：



如果你要删除 ri 指向的元素，你恐怕不能直接使用 i 了。这时 i 与 ri 分别指向不同的位置，因此，你需要删除的是 i 所指向元素的前一个元素。

- 如果要删除一个 reverse_iterator 指向的元素，需要通过 base()得到相应的 iterator，并且删除此 iterator 所指向的前一位值的元素。对于 erase 操作而言，reverse_iterator 与通过其 base()方法得到的 iterator 并非完全等价。

我们还是有必要看看 erase 操作的实际代码：

```
vector<int> v;
...                                          //插入 1 到 5，同上
vecot<int>::reverse_iterator ri =
     find(v.rbegin(), v.rend(), 3);          //ri 指向 3
v.erase(--ri.base());                        //vector 编译不通过。
```

这段代码并不存在什么设计问题，表达式--ri.base()确实能够指出我们需要删除的元素。而且，它们能够处理除了 vector 和 string 之外的其他所有容器。但问题是，对于 vector 和 string，--ri.base()可能会无法通过编译。这是因为 vector 和 string 容器的 iterator 和 const_iterator 通常会采用真实的指针来实现，而 ri.base()会返回一个内建指针。

C 和 C++都加强了对指针类型返回值的限制，你不能直接修改函数返回的指针。如果 vector 和 string 选用真实的指针作为 iterator，你也就不能修改 base()的返回值，所以--ri.base()无法通过编译。出于通用性和可移植性的考虑，应该避免直接修改 base()的返回值。当然，我们只需要先增加 ri 的值，然后再调用 base()方法：

```
...                                  //同上
v.erase((++ri).base());              //这下编译没问题了！
```

当你需要删除一个由 reverse_iterator 指出的元素时，应该首选这种更为通用而可移植的方法。

> 译者注：事实上 C 与 C++对所有的内建类型都加强了返回值限制，例如：
>
> ```
> int foo1();          //返回 int，内建类型
> foo1() ++;           //编译错误！foo1 返回 lvalue
> Widget foo2();       //返回 Widget，用户定义类型
> foo2() ++;           //只要 Widget 实现++，编译通过
> ```

由此可见，通过 base()方法可以取得一个与 reverse_iterator“ 相对应的 ”iterator 的说法并不准确。对于 insert 而言，这种对应关系确实存在；但是对于 erase 操作，情况并非如此简单。当你需要将 reverse_iterator 转换成 iterator 的时候，你必须根据所要进行的操作对 base()的返回值进行相应的调整。

总结：

STL 容器提供了四种不同的迭代器类型，但在实践当中，iterator 比其它三种迭代器更为实用。如果你有一个 const_iterator，并且可以访问它所指向的容器，你就可以通过 advance 和 distance 得到与值相应的 iterator。如果你有一个 reverse_iterator，你可以通过 base()方法得到“ 相对应的 ” iterator，但在你进行删除操作之前，必须首先调整 iterator 的值。

条款 16: 如何将 vector 和 string 的数据传给传统的 API

因为 C++语言已经于 1998 年被标准化，C++的中坚分子在努力推动程序员从数组转到 vector 时就没什么顾虑了。同样显然的情况也发生于尝试使开发者从 char*指针迁移到 string 对象的过程中。有很好的理由来做这些转变，包括可以消除常见的编程错误（参见条款 13），而且有机会获得 STL 算法的全部强大能力（比如参见条款 31）。

但是，障碍还是有的，最常见的一个就是已经存在的传统 C 风格的 API 接受的是数组和 char*指针，而不是 vector 和 string 对象。这样的 API 函数还将会存在很长时间，如果我们要高效使用 STL 的话，就必须和它们和平共处。

幸运的是，这很容易。如果你有一个 vector 对象 v，而你需要得到一个指向 v 中数据的指针，以使得它可以被当作一个数组，只要使用&v[0]就可以了。对于 string 对象 s，相应的咒语是简单的 s.c_str()。但是是只读的。如广告中难懂的条文时常指出的，必然会有几个限制。

给定一个

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

表达式 v[0]生产一个指向 vector 中第一个元素的引用，所以，&v[0]是指向那个首元素的指针。vector 中的元素被 C++标准限定为存储在连续内存中，就像是一个数组，所以，如果我们想要传递 v 给这样的 C 风格的 API：

```
void doSomething(const int* pInts, size_t numInts);
```

我们可以这么做：

```
doSomething(&v[0], v.size());
```

也许吧。可能吧。唯一的问题就是，如果 v 是空的。如果这样的话，v.size()是 0，而&v[0]试图产生一个指向根本就不存在的东西的指针。这不是件好事。其结果未定义。一个较安全的方法是这样：

```
if (!v.empty()) {
 doSomething(&v[0], v.size());
}
```

如果走错路了，你可能会碰到一些半吊子的人物，他们会告诉你说可以用 v.begin()代替&v[0]，因为（这些讨厌的家伙将会告诉你）begin 返回指向 vector 内部的迭代器，而对于 vector，其迭代器实际上是指针。那经常是正确的，但正如条款 50 所说，并不总是如此，你不该依赖于此。begin 的返回类型是 iterator，而不是一个指针，当你需要一个指向 vector 内部数据的指针时绝不该使用 begin。如果你基于某些原因决定键入 v.begin()，就应该键入&*v.begin()，因为这将会产生和&v[0]相同的指针，这样可以让你有更多的打字机会而且让其他要弄懂你代码得人感觉到更晦涩。坦白地说，如果你正在和告诉你使用 v.begin()代替&v[0]的人打交道的话，你该重新考虑一下你的社交圈了。（译注：在 VC6 中，如果用 v.begin()代替&v[0]，编译器不会说什么，但在 VC7 和 GCC 中这么做的话，就会引发一个编译错误）

类似从 vector 上获取指向内部数据的指针的方法，对 string 不是可靠的，因为（1）string 中的数据并没有承诺被存储在连续内存中，（2）string 的内部表示形式并没承诺以一个 null 字符结束。这解释了 string 的成员函数 c_str 存在的原因，它返回一个按 C 风格设计指针，指向 string 的值。因此我们可以这样传递一个 string 对象 s 给这个函数，

```
void doSomething(const char *pString);
```

象这样：

```
doSomething(s.c_str());
```

即使是字符串的长度为 0，它都能工作。在那种情况下，c_str 将返回一个指向 null 字符的指针。即使字符串内部自己内含 null 时，它同样能工作。但是，如果真的这样，doSomething 很可能将第一个内含的 null 解释为字符串结束。string 对象不在意是否容纳了结束符，但基于 char*的 C 风格 API 在意。

再看一下 doSomething 的声明：

```
void doSomething(const int* pInts, size_t numInts);
void doSomething(const char *pString);
```

在两种形式下，指针都被传递为指向 const 的指针。vector 和 string 的数据只能传给只读取而不修改它的 API。这到目前为止都是最安全的事情。对于 string，这也是唯一可做的，因为没有承诺说 c_str 产生的指针指在 string 数据的内部表示形式上；它可以返回一个指针指向数据的一个不可修改的拷贝，这个拷贝满足 C 风格 API 对格式的要求。（如果这个恐吓令你毛骨悚然的话，还请放心吧，因为它也许不成立。我没听说目前哪个库的实现使用了这个自由权的。）

对于 vector，有更多一点点灵活性。如果你将 v 传给一个修改其元素的 C 风格 API 的话，典型情况都是没问题，但被调用的函数绝不能试图改变 vector 中元素的个数。比如，它绝不能试图在 vector 还未使用的容量上“创建”新的元素。如果这么干了，v 将会变得内部状态不一致，因为它再也不知道自己的正确大小了。v.size()将会得到一个不正确的结果。并且，如果被调用的函数试图在一个大小和容量（参见条款 14）相等的 vector 上追加数据的话，真的会发生灾难性事件。我甚至根本就不愿去想象它。实在太可怕了。

你注意到我在前面的“典型情况都是没问题”那句话用的是“典型地”一词吗？你当然注意到了。有些 vector 对其数据有些额外的限制，而如果你把一个 vector 传递给需要修改 vector 数据的 API，你一定要确保这些额外限制继续被满足。举个例子，条款 23 解释了排序的 vector 往往是一种可行的选择来实现关联容器，但对这些 vector 而言，保持顺序非常重要。如果你将一个排序的 vector 传给一个可能修改其数据的 API 函数，你需要重视 vector 在调用返回后不再保持顺序的情况。

如果你想用 C 风格 API 返回的元素初始化一个 vector，你可以利用 vector 和数组潜在的内存分布兼容性将存储 vecotr 的元素的空间传给 API 函数：

```
// C API：此函数需要一个指向数组的指针，数组最多有 arraySize 个 double
// 而且会对数组写入数据。它返回写入的 double 数，不会大于 maxNumDoubles
size_t fillArray(double *pArray, size_t arraySize);
vector<double> vd(maxNumDoubles);    // 建立一个 vector，
      // 它的大小是 maxNumDoubles
vd.resize(fillArray(&vd[0], vd.size()));  // 让 fillArray 把数据
      // 写入 vd，然后调整 vd 的大小
      // 为 fillArray 写入的元素个数
```

这个技巧只能工作于 vector，因为只有 vector 承诺了与数组具有相同的潜在内存分布。但是，如果你想用来自 C 风格 API 的数据初始化 string 对象，也很简单。只要让 API 将数据放入一个 vector<char>，然后从 vector 中将数据拷到 string：

```
// C API：此函数需要一个指向数组的指针，数组最多有 arraySize 个 char
```

```
// 而且会对数组写入数据。它返回写入的 char 数，不会大于 maxNumChars
size_t fillString(char *pArray, size_t arraySize);
vector<char> vc(maxNumChars);   // 建立一个 vector，
      // 它的大小是 maxNumChars
size_t charsWritten = fillString(&vc[0], vc.size()); // 让 fillString 把
数据写入 vc
string s(vc.begin(), vc.begin()+charsWritten); // 从 vc 通过范围构造函数
      // 拷贝数据到 s（参见条款 5）
```

事实上，让 C 风格 API 将数据放入一个 vector，然后拷到你实际想要的 STL 容器中的主意总是有效的：

```
size_t fillArray(double *pArray, size_t arraySize); // 同上
vector<double> vd(maxNumDoubles);   // 一样同上
vd.resize(fillArray(&vd[0], vd.size());
deque<double> d(vd.begin(), vd.end());  // 拷贝数据到 deque
list<double> l(vd.begin(), vd.end());  // 拷贝数据到 list
set<double> s(vd.begin(), vd.end());   // 拷贝数据到 set
```

此外，这也提示了 vector 和 string 以外的 STL 容器如何将它们的数据传给 C 风格 API。只要将容器的每个数据拷到 vector，然后将它们传给 API：

```
void doSomething(const int* pInts, size_t numInts); // C API (同上)
set<int> intSet;     // 保存要传递给 API 数据的 set
...
vector<int> v(intSet.begin(), intSet.end());  // 拷贝 set 数据到 vector
if (!v.empty()) doSomething(&v[0], v.size());  // 传递数据到 API
```

你也可以将数据拷进一个数组，然后将数组传给 C 风格的 API，但你为什么想这样做？除非你在编译期就知道容器的大小，否则你不得不分配动态数组，而条款 13 解释了为什么你应该总是使用 vector 来取代动态分配的数组

条款 21: 永远让比较函数对相等的值返回 false

让我向你展示一些比较酷的东西。建立一个 set，比较类型用 less_equal，然后插入一个 10：

```
set<int, less_equal<int> > s;     // s 以“ <= ”排序
```

```
s.insert(10);       // 插入 10
```

现在尝试再插入一次 10：

```
s.insert(10);
```

对于这个 insert 的调用，set 必须先要判断出 10 是否已经位于其中了。 我们知道它是，但 set 可是木头木脑的，它必须执行检查。为了便于弄明白发生了什么，我们将一开始已经在 set 中的 10 称为 10A，而正试图插入的那个 10 叫 10B。

set 遍历它的内部数据结构以查找哪儿适合插入 10B。最终，它总要检查 10B 是否与 10A 相同。关联容器对“相同”的定义是等价(equivalence)（参见条款 19），因此 set 测试 10B 是否等价于 10A。当执行这个测试时，它自然是使用 set 的比较函数。在这一例子里，是 operator<=，因为我们指定 set 的比较函数为 less_equal，而 less_equal 意思就是 operator<=。于是，set 将计算这个表达式是否为真：

```
!(10A <= 10B) && !(10B <= 10A)     // 测试 10A 和 10B 是否等价
```

哦，10A 和 10B 都是 10，因此，10A <= 10B 肯定为真。同样清楚的是，10B <= 10A。于是上述的表达式简化为

```
!(true) && !(true)
```

再简化就是

```
false && false
```

结果当然是 false。 也就是说，set 得出的结论是 10A 与 10B 不等价，因此不一样，于是它将 10B 插入容器中 10A 的旁边。在技术上而言，这个做法导致未定义的行为，但是通常的结果是 set 以拥有了两个为 10 的值的拷贝而告终，也就是说它不再是一个 set 了。通过使用 less_equal 作为我们的比较类型，我们破坏了容器！此外，任何对相等的值返回 true 的比较函数都会做同样的事情。根据定义，相等的值却不是等价的！是不是很酷？

OK，也许你对酷的定义和我不一样。就算这样，你仍然需要确保你用在关联容器上的比较函数总是对相等的值返回 false。但是，你需要保持警惕。对这条规则的违反容易达到令人吃惊的后果。

举个例子，条款 20 描述了该如何写一个比较函数以使得容纳 string*指针的容器根据 string 的值排序，而不是对指针的值排序。那个比较函数是按升序排序的，但我们现在假设你需要 string*指针的容器的降序排序的比较函数。自然是抓现成

的代码来修改了。如果你不细心，可能会这么干，我已经加亮了对条款 20 中代码作了改变的部分：

```
struct StringPtrGreater:     // 高亮显示
public binary_function<const string*,  // 这段代码和 89 页的改变
const string*,      // 当心，这代码是有瑕疵的！
bool> {
 bool operator()(const string *ps1, const string *ps2) const
 {
  return !(*ps1 < *ps2);    // 只是相反了旧的测试；
 }       // 这是不对的！
};
```

这里的想法是通过将比较函数内部结果取反来达到反序的结果。很不幸，取反“<”不会给你（你所期望的）“>”，它给你的是“>=”。而你现在知道，因为它将对相等的值返回 true，对关联容器来说，它是一个无效的比较函数。

你真正需要的比较类型是这个：

```
struct StringPtrGreater:      // 对关联容器来说
public binary_function<const string*,   // 这是有效的比较类型
const string*,
bool> {
 bool operator()(const string *ps1, const string *ps2) const
 {
  return *ps2 < *ps1;    // 返回*ps2 是否
 }       // 大于*ps1（也就是
};        // 交换操作数的顺序）
```

要避免掉入这个陷阱，你所要记住的就是比较函数的返回值表明的是在此函数定义的排序方式下，一个值是否大于另一个。相等的值绝不该一个大于另一个，所以比较函数总应该对相等的值返回 false。

唉。

我知道你在想什么。你正在想，“当然，这对 set 和 map 很有意义，因为这些容器不能容纳复本。但是 multiset 和 multimap 怎么样呢？那些容器可以容纳复本，那些容器可能包含副本，因此，如果容器认为两个值相等的对象不等价，我需要注意些什么？它将会把两个都存储进去的，这正是 multi 系列容器的所要支持的事情。没有问题，对吧？”

错。想知道为什么，让我们返回头去看最初的例子，但这次使用的是一个 mulitset：

```
multiset<int, less_equal<int> > s;   // s 仍然以“<=”排序
s.insert(10);      // 插入 10A
s.insert(10);      // 插入 10B
```

现在，s 里有两个 10 的拷贝，因此我们期望如果我们在它上面做一个 equal_range，我们将会得到一对指出包含这两个拷贝的范围的迭代器。但那是不可能的。equal_range，虽然叫这个名字，但不是指示出相等的值的范围，而是等价的值的范围。在这个例子中，s 的比较函数说 10A 和 10B 是不等价的，所以不可能让它们同时出现在 equal_range 所指示的范围内。

你明白了吗？除非你的比较函数总是为相等的值返回 false，你将会打破所有的标准关联型容器，不管它们是否允许存储复本。

从技术上说，用于排序关联容器的比较函数必须在它们所比较的对象上定义一个“严格的弱序化(strict weak ordering)”。（传给 sort 等算法（参见条款 31）的比较函数也有同样的限制）。如果你对严格的弱序化含义的细节感兴趣，可在很多全面的 STL 参考书中找到，比如 Josuttis 的《The C++ Standard Library》[3]（译注：中译本《C++标准程序库》P176），Austern 的《Generic Programming and the STL》（译注：中译本《泛型程序设计与 STL》）[4]，和 SGI STL 的网站[21]。 我从未发现这个细节如此重要，但一个对严格的弱序化的要求直接指向了这个条款。那个要求就是任何一个定义了严格的弱序化的函数都必须在传入相同的值的两个拷贝时返回 false。

嗨！这就是这个条款！

STL 泛型算法 vs.手写的循环

Scott Meyers

准备进行优化？别那么急。Scott 正试图让你相信库函数比你自己写的更好。

[这篇文章源自一本即将出版的书。S. Meyers, Effective STL:50 Specific Ways to Improve Your Use of the Standard Template Library, 改自 Item 26-28（WQ 注，CUJ 上原文如此，应为 Item 43）。 2001 Addison-Wesley。 发行：permission of Pearson Education, Inc]

每个泛型算法接受至少一对选择子，以指示将被操作的元素区间。比如，min_element()寻找出此区间中的最小的值，而 accumulate()则对区间内的元素作

某种形式的整体求和运算，partition()将区间内的元素分割为满足和不满足某判决条件的两个部分。当泛型算法被执行时，它们必须检查指示给它的区间中的每个元素，并且是按你所期望的方式进行的：从区间的起始点循还到结束点。有一些泛型算法，比如 find()和 find_if()，可能在遍历完成前就返回了，但即使是这些泛型算法，内部都有着一个循环。毕竟，即使是 find()和 find_if()也必须在查看过了每个元素后，才能断定它们所寻找的元素不在此区间内。

所以，泛型算法内部是一个循环。此外，STL 泛型算法涉及面广泛，这意味着很多你本来要用循环来实现的任务，现在可以改用泛型算法实现了。比如，有一个 Widget 类，它支持 redraw()。

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

并且，你想 redraw 一个 list 中的所有 Widget 对象，你可能会使用这样一个循环：

```
list<Widget> lw;
...
for (list<Widget>::iterator i =
    lw.begin();
    i != lw.end(); ++i) {
  i->redraw();
}
```

但是你也可以用 for_each()泛型算法:

```
for_each(lw.begin(), lw.end(),
         mem_fun_ref(&Widget::redraw));
```

对许多 C++程序员而言，使用循环比调用泛型算法的想法自然多了，并且读解循环比弄明白 mem_fun_ref 和取 Widget::redraw 的地址要舒服多了。但是，这篇文章将说明调用泛型算法更可取。事实上，这篇文章将证明调用泛型算法通常比手写的循环更优越。为什么？

有三个理由：

- 效率：泛型算法通常比循环高效。
- 正确性：写循环时比调用泛型算法更容易产生错误。
- 可维护性：与相应的显式循环相比，泛型算法通常使代码更干净、更直观。

文章的以后部分将予以例证。

从效率方面看，泛型算法在 3 个方面打败了显式循环，两个主要因素，一个次要因素。次要因素是消除了多余的计算。回头看一下我们刚才写的循环：

```
for (list<Widget>::iterator i =
    lw.begin();
    i != lw.end();
```

```
    ++i) {
  i->redraw();
}
```

我已经加亮了循环终止测试语句，以强调每次循环，i 都要与 lw.end()作检查。也就是说，每次的循环，都要调用函数 list::end()。但我们不需要调用 end()一次以上的，因为我们不准备修改这个 list，对 end()调用一次就够了。而我们转过来看一下泛型算法，就可以看到只对 end()函数作了正确的求值次数：

```
// this call evaluates lw.end() exactly
// once
for_each(lw.begin(), lw.end(),
         mem_fun_ref(&Widget::redraw));
```

凭心而论，STL 的实现者知道 begin()和 end()(以及类似的函数，比如 size())用得很频繁，所以尽可能地实现得最高效。几乎肯定会 inline 它们，并编码得绝大部分编译器都能避免重复计算（通过将计算结果外提（这种优化手段））。然而，经验表明，这不是总能成功的，而且当不成功时，对重复计算的避免足以让泛型算法比手写的循环具有性能优势。

但这只是影响性能的次要因素。第一个主要影响因素是：库的实现者可以利用他们知道容器的具体实现的优势，用库的使用者无法采用的方式来优化代码。比如，在 deque 中的元素通常存储在（内部的）一个或多个固定大小的数组上。基于指针的遍历比基于选择子的遍历更快，但只有库的实现者可以使用基于指针的遍历，因为只有他们知道内部数组的大小以及如何从一个数组移向下一个。有一些 STL 容器和泛型算法的实现版本特别考虑了它们的 deque 的内部数据结构，而且已经知道，这样的实现比“通常”的实现快 20%。

第二个主要因素是，除了最微不足道的算法，所有的 STL 泛型算法使用的数学算法都比一般的 C++程序员能拿得出来的算法更复杂，——有时会复杂得多得多。不可能超越 sort()及其同族泛型算法的(比如，stable_sort()，nth_element()等)；适用于已序区间的搜索算法(比如，binary_search()，lower_bound()等)相当完美；就算是很平凡的任务，比如从 vector、deque 或数组中销毁元素，使用 erase-remove 惯用法都比绝大多数程序员写的循环更高效。

如果效率的因素说服不了你，也许你更愿意接受基于正确性的考虑。写循环时，比较麻烦的事在于确保所使用的选择子(a)有效，并且(b)指向你所期望的地方。举例来说，假设有一个数组，你想获得其中的每一个元素，在上面加 41，然后将结果从前端插入一个 deque。用循环，你可能这样写：

```
// C API: this function takes a pointer
// to an array of at most arraySize
// doubles and writes data to it. It
// returns the number of doubles written.
size_t fillArray(double *pArray, size_t arraySize);
// create local array of max possible size
double data[maxNumDoubles];
// create deque, put data into it
```

```
deque<double> d;
...
// get array data from API
size_t numDoubles =
  fillArray(data, maxNumDoubles);
// for each i in data, insert data[i]+41
// at the front of d; this code has a bug!
for (size_t i = 0; i < numDoubles; ++i) {
  d.insert(d.begin(), data[i] + 41);
}
```

这可以执行，只要你能满意于插入的元素是反序的。因为每次的插入点是 d.begin()，最后一个被插入的元素将位于 deque 的前端！

如果这不是你想要的（还是承认吧，它肯定不是你想要的），你可能想这样修改：

```
// remember d's begin iterator
deque<double>::iterator insertLocation = d.begin();
// insert data[i]+41 at insertLocation, then
// increment insertLocation; this code is also buggy!
for (size_t i = 0; i < numDoubles; ++i) {
  d.insert(insertLocation++, data[i] + 41);
}
```

看起来象双赢，它不只是累加了指示插入位置的选择子，还避免了每次对 begin()的调用（这消除了影响效率的次要因素）。唉，这种方法陷入了另外一个的问题中：它导致了“未定义”的结果。每次调用 deque::insert()，都将导致所有指向 deque 内部的选择子无效，包括上面的 insertLocation。在第一次调用 insert()后，insertLocation 就变得无效了，后面的循环可以产生任何行为（are allowed to head straight to looneyland）。

注意到这个问题后，你可能会这样做：

```
deque<double>::iterator insertLocation =
  d.begin();
// update insertLocation each time
// insert is called to keep the iterator valid,
// then increment it
for (size_t i = 0; i < numDoubles; ++i) {
  insertLocation =
    d.insert(insertLocation, data[i] + 41);
  ++insertLocation;
}
```

这样的代码确实完成了你相要的功能，但回想一下费了多大劲才达到这一步！和调用泛型算法 transform()对比一下：

```
// copy all elements from data to the
```

```
// front of d, adding 41 to each
transform(data, data + numDoubles,
          inserter(d, d.begin()),
          bind2nd(plus<int>(), 41));
```

这个“bind2nd(plus<int>(), 41)”可能会花上一些时间才能看明白(尤其是如果不常用 STL 的 bind 族的话)，但是与选择子相关的唯有烦扰就是指出源区间的起始点和结束点(而这从不会成为问题)，并确保在目的区间的起始点上使用 inserter。实际经验表明，为源区间和目的区间指出正确的初始选择子通常都很容易，至少比确保循环体没有于无意中将需要持续使用的选择子变得无效要容易得多。

因为在使用选择子前，必须时刻关注它们是否被不正确地操纵或变得无效，难以正确实现循环的情况太多了，这个例子只是比较有代表性。假设使用无效的选择子会导致“未定义”的行为，又假设“未定义”的行为在开发和测试期间 has a nasty habit of failing to show itself ，为什么要冒不必要的危险？将选择子扔给泛型算法，让它们去考虑操纵选择子时的各种诡异行为吧。

我已经解释了泛型算法为什么可以比手写的循环更高效，也描述了为什么循环将艰难地穿行于与选择子相关的荆棘丛中，而泛型算法正避免了这一点。运气好的话，你现在已是一个泛型算法的信徒了。然而运气是不足信的，在我休息前，我想更确保些。因此，让我们继续行进到代码清晰性的议题。最后，最好软件是那些最清晰的软件、最好懂的软件、能最被乐意于增强、维护和适用于新的环境的软件。虽然习惯于循环，但泛型算法在这个长期的竞争中具有优势。

关键在于具名词汇的力量。在 STL 中约有 70 个泛型算法的名字，总共超过 100 个不同的函数模板（每个重载都算一个）。每个泛型算法都完成一些精心定义的任务，而且有理由认为专业的 C++程序员知道（或应该去看一下）每个泛型算法都完成了什么。因此，当程序员调用 transform()时，他们认为对区间内的每个元素都施加了某个函数，而结果将被写到另外一个地方。当程序员调用 replace_if()时，他（她）知道区间内满足判定条件的对象都将被修改。当调用 partition()时，他（她）明白所有满足判定条件的对象将被聚集在一起。STL 泛型算法的名字传达了大量的语义信息，这使得它们比随意的循环清晰多了。

明摆着，泛型算法的名字暗示了其功能。“for”、“while”和“do”却做不到这一点。事实上，这一点对标准 C 语言或 C++语言运行库的所有部件都成立。毫无疑问地，你能自己实现 strlen()， memset()或 bsearch()，但你不会这么做。为什么不会？因为(1)已经有人帮你实现了它们，因此没必要你自己再做一遍；(2)名字是标准的，因此，每个人都知道它们做什么用的；和(3)你猜测程序库的实现者知道一些你不知道的关于效率方面的技巧，因此你不愿意错过熟练的程序库实现者可能能提供的优化。正如你不会去写 strlen()等函数的自己的版本，同样没道理用循环来实现出已存在的 STL 泛型算法的等价版本。

我很希望故事就此结束，因为我认为这个收尾很有说服力的。唉，好事多磨（this is a tale that refuses to go gentle into that good night）。 泛型算法的名字比光溜溜的循环有意义多了，这是事实，但使用循环更能让人明白加诸

于选择子上的操作。举例来说，假设想要找出 vector 中第一个比 x 大又比 y 小的元素。这是使用循环的实现：

```
vector<int> v;
int x, y;
...
// iterate from v.begin() until an
// appropriate value is found or
// v.end() is reached
vector<int>::iterator i = v.begin();
for( ; i != v.end(); ++i) {
  if (*i > x && *i < y) break;
}
// i now points to the value
// or is the same as v.end()
```

将同样的逻辑传给 find_if()是可能的，但是需要使用一个非标的 functor，比如 SGI 的 compose2[注 1]：

```
// find the first value val where the
// "and" of val > x and val < y is true
vector<int> iterator i =
  find_if(v.begin(), v.end(),
       compose2(logical_and<bool>(),
           bind2nd(greater<int>(), x),
           bind2nd(less<int>(), y)));
```

即使没使用非标的元件，许多程序员也会反对说它远不及循环清晰，我也不得不同意这个观点。

find_if()的调用可以不显得那么复杂，只要将测试的逻辑封装入一个独立的 functor(也就是申明了 operator()成员函数的类)：

```
template<typename T>
class BetweenValues:
  public std::unary_function<T, bool> {
public:
  // have the ctor save the
  // values to be between
  BetweenValues(const T& lowValue,
                const T& highValue)
  : lowVal(lowValue), highVal(highValue)
  {}
  // return whether val is
  // between the saved values
  bool operator()(const T& val) const
  {
```

```
    return val > lowVal && val < highVal;
  }
private:
  T lowVal;
  T highVal;
};
...
vector<int> iterator i =
  find_if(v.begin(), v.end(),
          BetweenValues<int>(x, y));
```

但这种方法有它自己的缺陷。首先，创建 BetweenValues 模板比写循环体要多出很多工作。就光数一下行数。 循环体：1 行；BetweenValues 模板：24 行。太不成比例了。其次，find_if()正在找寻是什么的细节被从调用上完全割裂出去了，要想真的明白对 find_if() 的这个调用，还必须查看 BetweenValues 的定义，但 BetweenValues 一定被定义在调用 find_if()的函数之外。如果试图将 BetweenValues 申明在这个函数内部，就像这样，

```
// beginning of function
{
  ...
  template <typename T>
  class BetweenValues:
    public std::unary_function<T, bool> { ... };
  vector<int>::iterator i =
    find_if(v.begin(), v.end(),
            BetweenValues<int>(x, y));
  ...
}
// end of function
```

你会发现编译不通过，因为模板不能申明在函数内部。如果试图用类代替模板而避开这个问题，

```
// beginning of function
{
  ...
  class BetweenValues:
    public std::unary_function<int, bool> { ... };
  vector<int> iterator i =
    find_if(v.begin(), v.end(),
            BetweenValues(x, y));
  ...
}
// end of function
```

你会发现仍然运气不佳，因为定义在函数内部的类是个局部类，而局部类不能绑定在模板的类型参数上(比如 find_if()所需要的 functor 类型)。很失望吧，functor 类和 functor 类不能被定义在函数内部，不管它实现起来有多方便。

在泛型函数与手写循环的长久较量中，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。如果你要做的是泛型算法已经提供了的，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，但调用泛型算法时却要使用 bind 族和 adapter 或者独立的 functor 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，天平再次倾向于泛型算法。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泛型算法(通常是 for_each())，以使得最终代码直截了当。

如果你同意调用泛型算法通常优于手写循环这个主题，并且，如果你也同意作用于某个区间的成员函数优于循环调用作用于单元素的成员函数[注 2]，一个有趣的结论出现了：使用 STL 容器的 C++精致程序中的循环比不使用 STL 的等价程序少多了。这是好事。只要能用高层次的术语（如 insert()、find()和 for_each()）取代了低层次的词汇（如 for、while 和 do），我们就提升了软件的抽象层次，并因此使得它更容易实现、文档化、增强，和维护。

条款 44: 尽量用成员函数代替同名的算法

有些容器拥有和 STL 算法同名的成员函数。关联容器提供了 count、find、lower_bound、upper_bound 和 equal_range，而 list 提供了 remove、remove_if、unique、sort、merge 和 reverse。大多数情况下，你应该用成员函数代替算法。这样做有两个理由。首先，成员函数更快。其次，比起算法来，它们与容器结合得更好（尤其是关联容器）。那是因为同名的算法和成员函数通常并不是是一样的。

我们以对关联容器的实验开始。假如有一个 set<int>，它存储了一百万个元素，而你想找到元素 727 的第一个出现位置（如果存在的话）。这儿有两个最自然的方法来执行搜索：

```
set<int> s;      // 建立 set，放 1,000,000 个数据
...      // 进去
set<int>::iterator i = s.find(727);   // 使用 find 成员函数
if (i != s.end()) ...
set<int>::iterator i = find(s.begin(), s.end(), 727); // 使用 find 算法
if (i != s.end()) ...
```

find 成员函数运行花费对数时间，所以不管 727 是否存在于此 set 中，set::find 只需执行不超过 40 次比较来查找它，而一般只需要大约 20 次。相反，find 算法运行花费线性时间，所以如果 727 不在此 set 中，它需要执行 1,000,000 次比较。即使 727 在此 set 中，也平均需要执行 500,000 次比较来找到它。效率得分如下：

find 成员：大约 40（最坏的情况）至大约 20（平均情况）
find 算法：1,000,000（最坏的情况）至 500,000（平均情况）

和高尔夫比赛一样，分值低的赢。正如你所见，这场比赛结果没什么可说的。

我必须对 find 成员函数所需的比较次数表示小小的谨慎，因为它有些依赖于关联容器的实现。绝大部分的实现是使用的红黑树——平衡树的一种——失衡度可能达到 2。在这样的实现中，对一百万个元素的 set 进行搜索所需最多的比较次数是 38 次。但对绝大部分的搜索情况而言，只需要不超过 22 次。一个基于完全平衡树的实现绝不需要超过 21 次比较，但在实践中，完全平衡树的效率总的来说不如红黑树。这就是为什么大多数的 STL 实现都使用红黑树。

效率不是 find 成员函数和 find 算法间的唯一差别。正如条款 19 所解释的，STL 算法判断两个对象是否相同的方法是检查的是它们是否相等(equality)，而关联容器是用等价(equivalence)来测试它们的“同一性”。 因此，find 算法搜索 727 用的是相等，而 find 成员函数用的是等价。相等和等价间的区别可能造成成功搜索和不成功搜索的区别。比如说，条款 19 演示了用 find 算法在关联容器搜索失败而用 find 成员函数却搜索成功的情况！因此，如果使用关联容器的话，你应该尽量使用成员函数形式的 find、count、lower_bound 等等，而不是同名的泛型算法，因为这些成员函数版本提供了和其它成员函数一致的行为。由于相等和等价间的差别，泛型算法不能提供这样的一致行为。

这一差别对 map 和 multimap 尤其明显，因为它们容纳的是对象对(pair object)，而它们的成员函数只在意对象对的 key 部分。因此，count 成员函数只统计 key 值匹配的对象对的数目（所谓“匹配”，自然是检测等价情况）；对象对的 value 部份被忽略。成员函数 find、lower_bound、upper_bound 和 equal_range 也是如此。但是如果你使用 count 算法，它的寻找将基于(a)相等和(b)对象对的全部组成部分；泛型算法 find、lower_bound 等同样如此。要想让泛型算法只关注于对象对的 key 部分，必须要跳出条款 23 描述的限制（那儿介绍了用相等测试代替等价测试的方法）。

另一方面，如果你真的关心效率，你可以采用条款 23 中的技巧，联合条款 34 中讲的对数时间搜索算法。相对于性能的提升，这只是一个很小的代价。再者，如果你真的在乎效率，你应该考虑非标准的 hash 容器（在条款 25 进行了描述），只是，你将再次面对相等和等价的区别。

对于标准的关联容器，选择成员函数而不是同名的算法有几个好处。首先，你得到的是对数时间而不是线性时间的性能。其次，你判断两个元素“相同”使用的是等价，这是关联容器的默认定义。第三，当操纵 map 和 multimap 时，你可以自动地只处理 key 值而不是 (key，value)对。这三点给了优先使用成员函数完美的铁甲。

让我们转到 list 的与算法同名的成员函数身上。这里的故事几乎全部是关于效率的。每个被 list 作了特化的算法（remove、remove_if、unique、sort、merge 和 reverse）都要拷贝对象，而 list 的特别版本什么都没有拷贝；它们只是简单地操纵连接 list 的节点的指针。算法和成员函数的算法复杂度是相同的，但如果操纵指针比拷贝对象的代价小的话，list 的版本应该提供更好的性能。

牢牢记住这一点很重要：list 成员函数的行为和它们的算法兄弟的行为经常不相同。正如条款 32 所解释的，如果你真的想从容器中清除对象的话，调用 remove、remove_if 和 unique 算法后，必须紧接着调用 erase 函数；但 list 的 remove、remove_if 和 unique 成员函数真的去掉了元素，后面不需要接着调用 erase。

在 sort 算法和 list 的 sort 成员函数间的一个重要区别是前者不能用于 list。作为单纯的双向迭代器，list 的迭代器不能传给 sort 算法。merge 算法和 list 的 merge 成员函数之间也同样存在巨大差异。这个算法被限制为不能修改源范围，但 list::merge 总是修改它的宿主 list。

所以，你明白了吧。 当面临着 STL 算法和同名的容器成员函数间进行选择时，你应该尽量使用成员函数。几乎可以肯定它更高效，而且它看起来也和容器的惯常行为集成得更好。

条款 45: STL 搜索算法的区别[1]

你要寻找什么，而且你有一个容器或者你有一个由迭代器划分出来的区间——你要找的东西就在里面。你要怎么完成搜索呢？你箭袋中的箭有这些：count、count_if、find、find_if、binary_search、lower_bound、upper_bound 和 equal_range。面对着它们，你要怎么做出选择？

简单。你能很快很容易地做到。能更快，更容易，更好。

暂时，我假设你有一对指定了搜索区间的迭代器。然后，我会考虑到你有的是一个容器而不是一个区间的情况。

要选择搜索策略，必须依赖于你的迭代器是否定义了一个已序区间。如果是，你就可以通过 binary_search、lower_bound、upper_bound 和 equal_range 来加速（通常是对数时间）搜索。如果迭代器并没有划分一个已序区间，你就只能用线性时间的算法 count、count_if、find 和 find_if。在下文中，我会忽略掉 count 和 find 是否有_if 的不同，就像我会忽略掉 binary_search、lower_bound、upper_bound 和 equal_range 是否带有谓词(predicate)的不同。你是依赖默认的搜索谓词还是指定一个自己的，对选择搜索算法的考虑是一样的。

如果你有一个无序区间，你的选择是 count 或着 find。它们分别可以回答略微不同的问题，所以值得仔细去区分它们。count 回答的问题是："是否存在这个值，如果有，那么存在几份拷贝？"而 find 回答的问题是："是否存在，如果有，那么它在哪儿？"

假设你想知道的东西是，是否有一个特定的 Widget 值w 在 list 中。如果用 count，这段代码看起来像这样：

```
list<Widget> lw;    // Widget 的列表
Widget w;     // 特殊的 Widget 值
...
if (count(lw.begin(), lw.end(), w)) {
 ...     // w 在 lw 中
} else {
 ...     // 不在
}
```

这里示范了一种常用手法：把 count 用来作为是否存在的检查。count 返回零或者一个正数，所以我们把非零转化为 true 而把零转化为 false。如果这样能使我们要做的更加显而易见，

```
if (count(lw.begin(), lw.end(), w) != 0) ...
```

而且有些程序员这样写，但是使用隐式转换则更常见，就像最初的例子。

和最初的代码比较，使用 find 略微更难懂些，因为你必须检查 find 的返回值和 list 的end 迭代器是否相等：

```
if (find(lw.begin(), lw.end(), w) != lw.end()) {
 ...     // 找到了
} else {
 ...     // 没找到
```

```
}
```

如果是为了检查是否存在，count 这个惯用方法编码起来比较简单。但是，当搜索成功时，它的效率比较低，因为当找到匹配的值后 find 就停止了，而 count 必须继续搜索，直到区间的结尾以寻找其他匹配的值。对大多数程序员来说，find 在效率上的优势足以证明略微增加复杂度是合适的。

通常，只知道区间内是否有某个值是不够的。取而代之的是，你想获得区间中的第一个等于该值的对象。比如，你可能想打印出这个对象，你可能想在它前面插入什么，或者你可能想要删除它。当你需要知道的不止是某个值是否存在，而且要知道哪个对象（或哪些对象）拥有该值，你就得用 find：

```
list<Widget>::iterator i = find(lw.begin(), lw.end(), w);
if (i != lw.end()) {
 ...    // 找到了，i 指向第一个
} else {
 ...    // 没有找到
}
```

对于已序区间，你有其他的选择，而且你应该明确的使用它们。count 和 find 是线性时间的，但已序区间的搜索算法（binary_search、lower_bound、upper_bound 和 equal_range）是对数时间的。

从无序区间迁移到已序区间导致了另一个迁移：从使用相等来判断是否两个值相同到使用等价[2]。那是因为 count 和 find 算法都用相等来搜索，而 binary_search、lower_bound、upper_bound 和 equal_range 则用等价。

要测试在已序区间中是否存在一个值，使用 binary_search。不像标准 C 库中的（因此也是标准 C++库中的）bsearch，binary_search 只返回一个 bool：这个值是否找到了。binary_search 回答这个问题："它在吗？"它的回答只能是是或者否。如果你需要比这样更多的信息，你需要一个不同的算法。

这里有一个 binary_search 应用于已序 vector 的例子:

```
vector<Widget> vw;   // 建立 vector，放入
...    // 数据，
sort(vw.begin(), vw.end()); // 把数据排序
Widget w;    // 要找的值
...
```

```
if (binary_search(vw.begin(), vw.end(), w)) {
 ...    // w 在 vw 中
} else {
 ...    // 不在
}
```

如果你有一个已序区间而且你的问题是："它在吗，如果是，那么在哪儿？"你需要 equal_range，但你可能想要用 lower_bound。我会很快讨论equal_range，但首先，让我们看看怎么用lower_bound 来在区间中定位某个值。

当你用 lower_bound 来寻找一个值的时候，它返回一个迭代器，这个迭代器指向这个值的第一个拷贝（如果找到的话）或者到可以插入这个值的位置（如果没找到）。因此 lower_bound 回答这个问题："它在吗？如果是，第一个拷贝在哪里？如果不是，它将在哪里？"和 find 一样，你必须测试 lower_bound 的结果，来看看它是否指向你要寻找的值。但又不像 find，你不能只是检测 lower_bound 的返回值是否等于 end 迭代器。取而代之的是，你必须检测 lower_bound 所标示出的对象是不是你需要的值。

很多程序员这么用 lower_bound：

```
vector<Widget>::iterator i = lower_bound(vw.begin(), vw.end(), w);
if (i != vw.end() && *i == w) { // 保证 i 指向一个对象；
    // 也就保证了这个对象有正确的值。
    // 这是个 bug！
 ...    // 找到这个值，i 指向
    // 第一个等于该值的对象
} else {
 ...    // 没找到
}
```

大部分情况下这是行得通的，但不是真的完全正确。再看一遍检测需要的值是否找到的代码：

```
if (i != vw.end() && *i == w) ...
```

这是一个相等的测试，但 lower_bound 搜索用的是等价。大部分情况下，等价测试和相等测试产生的结果相同，但就像注释 2 论证的，相等和等价的结果不同的情况并不难见到。在这种情况下，上面的代码就是错的。

要完全完成，你就必须检测 lower_bound 返回的迭代器指向的对象的值是否和你要寻找的值等价。你可以手动完成，但可以更狡猾地完成，因为你必须确认使用了和 lower_bound 使用的相同的比较函数。一般而言，那可以是一个自由函数（或函数对象）。如果你传递一个比较函数给 lower_bound，你必须确认和你的手写的等价检测代码使用了相同的比较函数。这意味着如果你改变了你传递给 lower_bound 的比较函数，你也得对你的等价检测部分作出修改。保持比较函数同步不是火箭发射，但却是另一个要记住的东西，而且我想你已经有很多需要你记的东西了。

这儿有一个简单的方法：使用 equal_range。equal_range 返回一对迭代器，第一个等于 lower_bound 返回的迭代器，第二个等于 upper_bound 返回的（也就是，等价于要搜索值区间的末迭代器的下一个）。因此，equal_range，返回了一对划分出了和你要搜索的值等价的区间的迭代器。一个名字很好的算法，不是吗？（当然，也许叫 equivalent_range 会更好，但叫 equal_range 也非常好。）

对于 equal_range 的返回值，有两个重要的地方。第一，如果这两个迭代器相同，就意味着对象的区间是空的；这个只没有找到。这个结果是用 equal_range 来回答"它在吗？"这个问题的答案。你可以这么用：

```
vector<Widget> vw;
...
sort(vw.begin(), vw.end());
typedef vector<Widget>::iterator VWIter; // 方便的 typedef
typedef pair<VWIter, VWIter> VWIterPair;
VWIterPair p = equal_range(vw.begin(), vw.end(), w);
if (p.first != p.second) {    // 如果 equal_range 不返回
      // 空的区间...
 ...      // 说明找到了，p.first 指向
      // 第一个而 p.second
      // 指向最后一个的下一个
} else {
 ...      // 没找到，p.first 和
      // p.second 都指向搜索值
```

```
}       // 的插入位置
```

这段代码只用等价，所以总是正确的。

第二个要注意的是 equal_range 返回的东西是两个迭代器，对它们作 distance 就等于区间中对象的数目，也就是，等价于要寻找的值的对象。结果，equal_range 不光完成了搜索已序区间的任务，而且完成了计数。比如说，要在 vw 中找到等价于 w 的 Widget，然后打印出来有多少这样的 Widget 存在，你可以这么做：

```
VWIterPair p = equal_range(vw.begin(), vw.end(), w);
cout << "There are " << distance(p.first, p.second)
<< " elements in vw equivalent to w.";
```

到目前为止，我们所讨论的都是假设我们要在一个区间内搜索一个值，但是有时候我们更感兴趣于在区间中寻找一个位置。比如，假设我们有一个 Timestamp 类和一个 Timestamp 的 vector，它按照老的 timestamp 放在前面的方法排序：

```
class Timestamp { ... };
bool operator<(const Timestamp& lhs, // 返回在时间上 lhs
 const Timestamp& rhs);    // 是否在 rhs 前面
vector<Timestamp> vt;    // 建立 vector，填充数据，
...      // 排序，使老的时间
sort(vt.begin(), vt.end());  // 在新的前面
```

现在假设我们有一个特殊的 timestamp——ageLimit，而且我们从 vt 中删除所有比 ageLimit 老的 timestamp。在这种情况下，我们不需要在 vt 中搜索和 ageLimit 等价的 Timestamp，因为可能不存在任何等价于这个精确值的元素。

取而代之的是，我们需要在 vt 中找到一个位置：第一个不比 ageLimit 更老的元素。这是再简单不过的了，因为 lower_bound 会给我们答案的：

```
Timestamp ageLimit;
...
vt.erase(vt.begin(), lower_bound(vt.begin(), // 从 vt 中排除所有
 vt.end(),      // 排在 ageLimit 的值
```

```
 ageLimit));      // 前面的对象
```

如果我们的需求稍微改变了一点，我们要排除所有至少和 ageLimit 一样老的 timestamp，也就是我们需要找到第一个比 ageLimit 年轻的 timestamp 的位置。这是一个为 upper_bound 特制的任务：

```
vt.erase(vt.begin(), upper_bound(vt.begin(), // 从 vt 中排除所有
 vt.end(),      // 排在 ageLimit的值前面
 ageLimit));      // 或者等价的对象
```

如果你要把东西插入一个已序区间，而且对象的插入位置是在有序的等价关系下它应该在的地方时，upper_bound 也很有用。比如，你可能有一个已序的 Person 对象的 list，对象按照 name 排序：

```
class Person {
public:
 ...
 const string& name() const;
 ...
};
struct PersonNameLess:
public binary_function<Person, Person, bool> {
 bool operator()(const Person& lhs, const Person& rhs) const
 {
  return lhs.name() < rhs.name();
 }
};
list<Person> lp;
...
lp.sort(PersonNameLess());  // 使用 PersonNameLess 排序 lp
```

要保持 list 仍然是我们希望的顺序（按照 name，插入后等价的名字仍然按顺序排列），我们可以用 upper_bound 来指定插入位置：

```
Person newPerson;
...
lp.insert(upper_bound(lp.begin(),  // 在 lp 中排在 newPerson
 lp.end(),     // 之前或者等价
 newPerson,     // 的最后一个
 PersonNameLess()),    // 对象后面
 newPerson);     // 插入 newPerson
```

这工作的很好而且很方便，但很重要的是不要被误导——错误地认为 upper_bound 的这种用法让我们魔术般地在一个 list 里在对数时间内找到了插入位置。我们并没有。因为我们是用 list，查找花费线性时间，但是它只用了对数次的比较。

一直到这里，我都只考虑我们有一对定义了搜索区间的迭代器的情况。通常我们有一个容器，而不是一个区间。在这种情况下，我们必须区别序列和关联容器。对于标准的序列容器（vector、string、deque 和 list），你应该遵循我在本条款提出的建议，使用容器的 begin 和 end 迭代器来划分出区间。

这种情况对标准关联容器（set、multiset、map 和 multimap）来说是不同的，因为它们提供了搜索的成员函数，它们往往是比用 STL 算法更好的选择[3]。幸运的是，成员函数通常和相应的算法有同样的名字，所以前面的讨论推荐你使用的算法 count、find、equal_range、lower_bound 或 upper_bound，在搜索关联容器时你都可以简单的用同名的成员函数来代替。

binary_search 调用的策略不同，因为这个算法没有提供对等的成员函数。要测试在 set 或 map 中是否存在某个值，使用 count 的惯用方法来对成员进行检测：

```
set<Widget> s;  // 建立 set，放入数据
...
Widget w;   // w 仍然是保存要搜索的值
...
if (s.count(w)) {
 ...   // 存在和 w 等价的值
} else {
 ...   // 不存在这样的值
}
```

要测试某个值在 multiset 或 multimap 中是否存在，find 往往比 count 好，因为一旦找到等于期望值的单个对象，find 就可以停下了，而 count，在最遭的情况下，必须检测容器里的每一个对象。

但是，count 给关联容器计数是可靠的。特别，它比调用 equal_range 然后应用 distance 到结果迭代器更好。首先，它更清晰：count 意味着"计数。"第二，它更简单；不用建立一对迭代器然后把它的组成（译注：就是 first 和 second）传给 distance。第三，它可能更快一点。

要给出所有我们在本条款中所考虑到的，我们的从哪儿着手？下面的表格道出了一切。

	使用的算法		使用的成员函数	
你想知道的	**在无序区间**	**在已序区间**	**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 or lower_bound(see article)
第一个不等于期望值的对象在哪里？	find_if	lower_bound	lower_bound	lower_bound
第一个等于期望值的对象在哪里？	find_if	upper_bound	upper_bound	upper_bound
有多少对象等于期望值？	count	equal_range	count	count
等于期望值的所有对象在哪里？	find（迭代）	equal_range	equal_range	equal_range

上表总结了要怎么操作已序区间，equal_range 的出现频率可能令人吃惊。当搜索时，这个频率因为等价检测的重要性而上升了。对于 lower_bound 和 upper_bound，它很容易在相等检测中退却，但对于 equal_range，只检测等价是很自然的。在第二行已序区间，equal_range 打败了 find 还因为一个理由：equal_range 花费对数时间，而 find 花费线性时间。

对于 multiset 和 multimap，当你在搜索第一个等于特定值的对象的那一行，这个表列出了 find 和 lower_bound 两个算法作为候选。 已对于这个任务 find 是通常的选择，而且你可能已经注意到在 set 和 map 那一列里，这项只有 find。但是对于 multi 容器，如果不只有一个值存在，find 并不保证能识别出容器里的等于给定值的第一个元素；它只识别这些元素中的一个。如果你真的需要找到等于给定值的第一个元素，你应该使用 lower_bound，而且你必须手动的对第二部分做等价检测。（你可以用 equal_range 来避免作手动等价检测，但是调用 equal_range 的花费比调用 lower_bound 多得多。）

在 count、find、binary_search、lower_bound、upper_bound 和 equal_range 中做出选择很简单。当你调用时，选择算法还是成员函数可以给你需要的行为和性能，而且是最少的工作。按照这个建议做（或参考那个表格），你就不会再有困惑。

[1] Scott Meyers，Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library，Addison-Wesley，2001，ISBN 0-201-74962-9。这篇文章是基于 Effective STL 的条款 45 写的。

[2] 如果在某种排序顺序下，一个没有在另一个之前，那么两个对象等价。通常，等价的值相等，但不总是这样。比如，字符串“STL”和“stl”在大小写不敏感的排序中是等价的，但是它们显然不相等。要知道等价和相等区别的详细内容，请参考条款 19。