目录

第一章 容器

第 1 条：慎重选择容器类型

C++ 提供了几种不同的容器供你选择，你有没有发现他们的共同点在哪里？

- 标准 STL 序列容器：vector、string、deque、list。
- 标准 STL 关联容器：set、multiset、map、multimap（注：unordered_set、unordered_map 在 C++11 的时候也被引入标准库）。
- 非标准序列容器：slist：是一个单向链表
- 非标准的关联容器：hash_set、hash_multiset、hash_map、hash_multimap。

连续内存容器 ：把它的元素存放在一块或多块（动态分配）内存中，每块内存中存有多个元素，当有新元素插入或已有的元素被删除时，统一内存块的元素需要向前或向后移动。以便给新元素让出空间，或者填充被删除元素所留下的空隙，这种移动影响效率和异常安全性。

基于节点的容器：每一个（动态分配）的内存块中只存放一个元素。容器中元素的插入或删除只影响到指向节点的指针，而不影响节点本身的内容，所以当有插入或删除操作时候，元素的值不需要移动。

有了这些基础，为了选择的时候有所顾虑，少踩一些坑，《ESTL》第一章给出了简单的建议：

- **是否需要在容器的任意位置插入新元素？** 如果需要，就选择序列容器否则选择关联容器。
- **是否关心容器中元素是排序的？** 如果不关心则哈希容器是一个可行选择方案；否则你要避免哈希容器。
- **需要哪种类型的迭代器？** 如果必须是随机访问迭代器，则对容器的选择就限定为 vector、deque 和 string。如果要求使用双向迭代器，则避免使用 slist 和哈希容器。
- **当发生元素的插入和删除操作时候，避免移动容器中原来的元素是否重要？** 如果是就要避免选择序列容器。
- **容器中数据布局是否需要和 C 保持兼容？** 如果是是只能选择 vector。
- **元素的查找速度是否是关键的考虑因素？** 如果是考虑哈希容器。
- **如果容器内部使用引用计数技术是否介意？** 如果是则避免使用 string 因为许多 string 的实现都是用了引用计数。如果需要表示某种字符串的方法，可以使用 vector 方法。
- **对插入和删除操作，需要事务语义么** 在插入和操作失败的时需要回滚的能力么？如果需要就有使用基于节点的容器，如果需要对多个元素插入操作需要事务，则可以选择 list。因为在标准容器中，只有 list 提供了多个元素的事务语义。但注意：事务语义对编写异常安全代码很重要但同时付出性能上代价。
- **需要使用迭代器、指针和引用变为无效的次数最少么** 如果是就需要使用基于节点的容器，因为这类对容器的插入和删除从来不会使用迭代器和指针和引用无效；而对连续内存的容器的插入和删除一般会对指向该容器的迭代器和指针、引用变为无效。
- **如果在容器上使用 swap 使得迭代器失效了会在意吗** 如果在意那么避免使用 string，因为 string 是唯一在 STL 中 swap 操作过程中导致迭代器、指针和引用无效唯一的容器。
- **如果序列容器迭代器是随机访问，而且只要没有删除操作发生，且插入操作只发生容器末尾，则指向数据的指针和引用就不会变为无效，这样容器是否考虑** 这是一种很特殊的情况，如果是则 deque 满足你的需求，deque 是唯一的迭代器可能会变为无效和指针和引用不会变为无效的 STL 标准容器。

第 2 条：不要试图编写独立于容器类型的代码

STL 是以泛化原则为基础的：

- 数组被泛化为”以其包含的对象的类型为参数“的容器；

- 函数被泛化为”以其使用的迭代器的类型为参数“的算法；
- 指针被泛化为”以其指向的对象的类型为参数“的迭代器；
- 容器被泛化为”序列式和关联式“容器。

试图编写对序列式容器和关联式容器都适用的代码几乎是毫无意义的。面对实际情况，不同容器是不同的，它们有非常明显的优缺点，不同的适用场景适合选择不同的容器。

第 3 条：确保容器中的对象拷贝正确而高效

利用一个对象的拷贝成员函数就可以很方便地拷贝该对象，特别是对象的拷贝构造函数 和拷贝赋值操作符 。

在存在继承关系的基础下，拷贝动作会导致剥离，也就是说，如果你创建了一个存放基类对象的容器，却向其中插入派生类的对象，那么在派生类对象(通过基类的拷贝构造函数)被拷贝进容器时，它所特有的部分(即派生类中的信息)将会丢失。**剥离问题**意味着向基类对象的容器中插入派生类对象几乎总是错误的。

```
class Widget {};
class SpecialWidget: public Widget {};
int test_item_3() {
  vector<Widget> vw;
  SpecialWidget sw;
  vw.push_back(sw);//sw 作为基类对象被拷贝进 vw，它的派生类特有的部分在拷贝的时候被丢弃了
}
```

使拷贝动作高效、正确，并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。比如使用 vector<Widget*> vw; 而非上述操作。

第 4 条：调用 empty 而不是 size() 是否为 0

```
//对任意 c 容器，下面的代码本质上等价的
if (c.size() == 0) {}
if (c.empty()) {}
```

那既然如此，为何建议使用 empty 优先级于 size()，原因在于：empty 对所有的标准容器都是常数时间操作，而对一些 list 实现，size 函数耗费线性时间。（比如 list 容器的 splice 函数）。

第 5 条：区间成员函数优先于与之对应的单元素成员函数

```
class Widget5 {};

int test_item_5()
{
 vector<Widget5> v1, v2;
 v1.assign(v2.begin() + v2.size() / 2, v2.end()); // 推荐

 v1.clear();
 for (vector<Widget5>::const_iterator ci = v2.begin() + v2.size() /
2; ci != v2.end(); ++ci) // 不推荐
  v1.push_back(*ci);

 v1.clear();
 copy(v2.begin() + v2.size() / 2, v2.end(), back_inserter(v1)); // 效
率不如 assign

 v1.clear();
 v1.insert(v1.end(), v2.begin() + v2.size() / 2, v2.end()); // 对
copy 的调用可以被替换为利用区间的 insert 版本

 const int numValues = 100;
 int data[numValues];

 vector<int> v;
 v.insert(v.begin(), data, data + numValues); // 推荐，使用区间成员函
数 insert

 vector<int>::iterator insertLoc(v.begin());
 for (int i = 0; i < numValues; ++i) {
  insertLoc = v.insert(insertLoc, data[i]);
// 不推荐使用单元素成员函数
  ++insertLoc;
 }

 return 0;
}
```

区间成员函数是指这样的一类成员函数，它们像 STL 算法一样，使用两个迭代器参数来确定该成员操作所执行的区间。如果不使用区间成员函数就得写一个显示的循环。

优点在于：

- C++ 标准要求区间 insert 函数把现有容器中元素直接移动到它们最终的位置上，即只需要付出每个元素移动一次的代价。

- 明智地使用区间插入而不是单元素重复插入会提高程序的性能问题，比如对于 vector 来说如果内存已满再插入新元素会触发两倍扩容，区间插入不必多次重新分配内存。
- 区间成员函数减少代码量，形成更易懂的代码，增强软件长期可维护性。

第 6 条：当心 C++ 编译器最烦人的分析机制

下面我们分析的问题现在已经得到解决了，C++现在允许使用大括号“{}”初始化来避免下面的问题。但是这一问题还是值得我们研究的。

假设我们有一个存放整型（int）的文件，你想把这些整数拷贝到一个 list 中，那么你可能会使用下面的做法

```
ifstream dataFile("ints.dat");

//使用 list 的区间构造函数来初始化 list（C++编译器可能会识别错误）
list<int>                    data(istream_iterator<int>(dataFile),
istream_iterator<int>());
```

C++编译器可能会上面 list 的构造函数理解为一种函数：包含两个参数和一个 list<int>返回值

因为编译器可能会把上面那种形式的构造函数理解为一种函数，因此上面的 data 不会做任何事情，因此 data 为空
我们可以为 istream_interator 迭代器取名（而不是使用匿名）

```
ifstream dataFile("ints.dat");

istream_iterator<int> dataBegin(dataFile);
istream_iterator<int> dataEnd;

list<int> data(dataBegin, dataEnd);
```

实际编程中都建议使用匿名的 istream_interator 对象。现在 C++标准已经提供了使用大括号“{}”来初始化，因此上面的所有问题都可以得到解决。

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

```
class Widget7 {};

struct DeleteObject {
 template<typename T>
 void operator()(const T* ptr) const {delete ptr;}
};

int test_item_7()
{
 const int num = 5;

 vector<Widget7*> vwp1, vwp2;
 for (int i = 0; i < num; ++i) {
  vwp1.push_back(new Widget7);
// 如果在后面自己不 delete，使用 vwp 在这里发生了 Widget7 的泄露
  vwp2.push_back(new Widget7);
 }

 for (vector<Widget7*>::iterator i = vwp1.begin(); i != vwp1.end();
++i) {
  delete *i; // 能行，但不是异常安全的
 }

 for_each(vwp2.begin(), vwp2.end(), DeleteObject());
// 正确，类型安全，但仍不是异常安全的

 typedef shared_ptr<Widget7> SPW;
// SPW"指向 Widget7 的 shared_ptr"
 vector<SPW> vwp3;
 for (int i = 0; i < num; ++i) {
  vwp3.push_back(SPW(new Widget7));
 // 从 Widget7 创建 SPW,然后对它进行一次 push_back 使用 vwp3,这里不会有
Widget7 泄露，即使有异常被抛出
 }

 return 0;
}
```

STL 容器很智能，但没有智能到知道是否该删除自己所包含的指针的程度。当你使用指针的容器，而其中的指针应该被删除时，为了避免资源泄漏，你必须或者用引用计数形式的智能指针对象(比如 shared_ptr)代替指针，或者当容器被析构时手工删除其中的每个指针。

第 8 条：切勿创建包含 auto_ptr 的容器对象

auto_ptr 的容器是被禁止的。当你拷贝一个 auto_ptr 时，它所指向的对象的所有权被移交到拷入的 auto_ptr 上，而它自身被置为 NULL。如果你的目标是包含智能指针的容器，这并不意味着你要倒霉，包含智能指针的容器是没有问题的。但 auto_ptr 非智能指针。

第 9 条：慎重选择删除元素的方法

- 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；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 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，或者写一个循环来遍历容器中的元素，记住当把迭代器传给 erase 时，要对它进行后缀递增。
- 要在循环内做某些(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，记住每次调用 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，记住当把迭代器传给 erase 时，要对迭代器做后缀递增。

remove_if（遍历元素，将满足条件的元素移动到容器的末尾）
remove_copy_if: 将所有不匹配元素拷贝到一个指定容器。

第 10 条：了解分配子(allocator)的约定和限制

编写自定义的分配子，需要注意：

- 你的分配子是一个模板，模板参数 T 代表你为它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 指针，reference 为 T&。
- 千万别让你的分配子拥有随对象而不同的状态。通常，分配子不应该有非静态的数据成员。*
- 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。

第 11 条：理解并自定义分配子的合理用法

```
void* mallocShared(size_t bytesNeed) {
    return malloc(bytesNeed);
}
void freeShared(void* ptr) {
    free(ptr);
}

template<typename T>
class SharedMemoryAllocator {
public:
    typedef T* pointer;//pointer 是个类型定义，它实际上总是 T*
    typedef size_t size_type; //通常情况下，size_type 是 size_t 的一
个类型定义
    typedef T value_type;

    pointer allocate(size_type numObj, const void* localHint = 0) {
        return static_cast<pointer>(mallocShared(numObj *
sizeof(T)));
    }
    void deallocate(pointer ptrToMemory, size_type numObj) {
  freeShared(ptrToMemory);
 }
    template<typename U>
 struct rebind {
  typedef allocator<U> other;
 };
};

int test_item_11() {
 typedef vector<double, SharedMemoryAllocator<double>>
SharedDoubleVec;

 // v 所分配的用来容纳其元素的内存将来自共享内存
 // 而 v 自己----包括它所有的数据成员----几乎肯定不会位于共享内存中，v
只是普通的基于栈(stack)的对象，所以，像所有基于栈的对象一样，它将会被
运行时系统放在任意可能的位置上。这个位置几乎肯定不是共享内存
 SharedDoubleVec v; // 创建一个 vector,其元素位于共享内存中


 // 为了把 v 的内容和 v 自身都放到共享内存中，需要这样做
 void* pVectorMemory = mallocShared(sizeof(SharedDoubleVec)); // 为
SharedDoubleVec 对象分配足够的内存
 SharedDoubleVec* pv = new (pVectorMemory)SharedDoubleVec; // 使用
"placement new"在内存中创建一个 SharedDoubleVec 对象
```

```
pv->~SharedDoubleVec(); // 析构共享内存中的对象
 freeShared(pVectorMemory); // 释放最初分配的那一块共享内存

 return 0;
}
```

遵守同一类型的分配子必须是等价的这一限制要求。

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：

- 多个线程读取是安全的。
- 多个线程对不同的容器做写入操作是安全的

考虑当一个库视图实现完全的容器线程安全性时可能采取的方式：

- 对容器成员函数的每次调用，都锁住容器直到调用结束
- 在容器所返回的每个迭代器的生存期结束前，都锁住容器
- 对作用于容器的每个算法，都锁住该该容器，直到容器结束

当涉及到 STL 容器和线程安全性时，你可以指望一个 STL 库允许多个线程同时读一个容器，以及多个线程对不同的容器做写入操作。你不能指望 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。

使用一个类 Lock 来管理资源的生存期。

```
{
  Lock<vector<int>> lock(v);
  vector<int>::iterator first5(find(v.begin(),v.end(),5));
  if(*first!=v.end()){
      *first=5;
    }
}//关闭块
//释放互斥量
```

这种基于 Lock 的方法在有异常的情况下是稳健的，如果抛出了异常，局部对象就会销毁，所以 Lock 也会释放它的互斥量。

第二章 vector 和 string

第 13 条：vector 和 string 优先于动态分配的数组

如果使用动态的分配数组，那么可能需要做更多的工作，为了减轻负担，使用 vector 和 string 。

第 14 条：使用 reserve 来避免不必要的重新分配

```
int test_item_14() {
 vector<int> v;
 v.reserve(1000); // 如果不使用 reserve,下面的循环在进行过程中将导致 2 到 10 次重新分配;加上 reserve，则在循环过程中,将不会再发生重新分配
 for (int i = 1; i <= 1000; ++i) v.push_back(i);
 return 0;
}
```

对于 vector 和 string，增长过程是这样来实现的：每当需要更多空间时，就调用与 realloc 类似的操作。这一类似于 realloc 的操作分为四部分：

- 分配一块大小为当前容量的某个倍数的新内存。在大多数实现中，vector 和 string 的容量每次以 2 的倍数增长，即，每当容器需要扩张时，它们的容量即加倍。
- 把容器的所有元素从旧的内存拷贝到新的内存中。
- 析构掉就内存中的对象。
- 释放旧内存。

reserve 函数能使你把重新分配的次数减少到最低程度，从而避免了重新分配和指针/迭代器/引用失效带来的开销。避免重新分配的关键在于，尽早地使用 reserve ，把容器的容量设为足够大的值，最好是在容器刚被构造出来之后就使用 reserve。

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元素，则此时可以使用 reserve。第二种方式是，先预留足够大的空间(根据你的需要而定)，然后，当把所有数据都加入以后，再去除多余的容量。

第 15 条：注意 string 实现的多样性

```
int test_item_15() {
```

```
 fprintf(stdout, "string size: %d, char* size: %d\n", sizeof(string),
sizeof(char*));

 return 0;
}
```

- string 的值可能会被引用计数，也可能不会。很多实现在默认情况下会使用引用计数，但它们通常提供了关闭默认选择的方法，往往是通过预处理宏来做到这一点。
- string 对象大小的范围可以是一个 char* 指针大小的 1 倍到 7 倍。
- 创建一个新的字符串值可能需要零次、一次或两次动态分配内存。
- string 对象可能共享，也可能不共享其大小和容量信息。
- string 可能支持，也可能不支持针对单个对象的分配子。
- 不同的实现对字符内存的最小分配单位有不同的策略。

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

C++ 标准要求 vector 中的元素存储在连续的内存中，就像数组一样。string 中的数据不一定存储在连续的内存中，而且 string 的内部表示不一定是以空字符结尾的。

第 17 条：使用”swap 技巧”除去多余的容量

考虑下面的需求，对于 vec 开始的时候有 1000 个元素，后来只有 10 个元素，那么 vec 的 capacity 至少还是 1000，后面的 990 个内存单元，没有使用，但是还被 vec 霸占着。如何释放这些内存呢？

vector 进行 copy 构造的时候，根据 rhs 的 size 进行分配内存。因此，我们可以建立一个临时对象，然后交换一下就可以了。如下：

```
    vector<int>(vec).swap(vec);
```

vector<int>(vec) 是个临时对象，可认为 capacity 为 10，而 vec 的 capacity 为 1000，二者交换后，vec 的 capacity 为 10，临时对象析构。C++11 中增加了 shrink_to_fit 成员函数。它减少容器的容量以适应其大小并销毁超出容量的所有元素。

考虑一个特殊情况，我想清空一个容器，并释放所有内存，该怎么办？

首先，clear 方法是不行的，因为它只是把元素清空，内存还被霸占着。由上面的分析，很容易想到，拿一个空容器与当前容器交换一下，就行了。也就是：vector<int>().swap(vec);

第 18 条： 避免使用 vector < bool> 类型

```
int test_item_18() {
 vector<bool> v;
 // error: cannot convert 'vector<bool>::reference* {aka
_Bit_reference}' to 'bool*' in initialization
 //bool* pb = &v[0]; // 不能被编译，原因：vector<bool>是一个假的容
器，它并不真的储存 bool，相反，为了节省空间，它储存的是 bool 的紧凑表
示
 return 0;
}
```

作为一个 STL 容器，vector 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。

在一个典型的实现中，储存在 ”vector” 中的每个 ”bool” 仅占一个二进制位，一个 8 位的字节可容纳 8 个 ”bool”。在内部，vector 使用了与位域 (bit field)一样的思想，来表示它所存储的那些 bool，实际上它只是假装存储了这些 bool。

位域与 bool 相似，它只能表示两个可能的值，但是在 bool 和看似 bool 的位域之间有一个很重要的区别：你可以创建一个指向 bool 的指针，而指向单个位的指针则是不允许的。指向单个位的引用也是被禁止的。

当你需要 vector 时，标准库提供了两种选择，可以满足绝大多数情况下的需求。

- 第一种是 deque。deque 几乎提供了 vector 所提供的一切(没有 reserve 和 capacity)，但 deque 是一个 STL 容器，而且它确实存储 bool。当然 deque 中元素的内存不是连续的，所以你不能把 deque 中的数据传递给一个期望 bool 数组的 C API。
- 第二种可以替代 vector 的选择是 bitset。bitset 不是 STL 容器，但它是标准 C++ 库的一部分。与 STL 容器不同的是，它的大小(即元素的个数)在编译时就确定了，所以它不支持插入和删除元素。

第三章 关联容器

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

相等的概念是基于 operator== 的。等价关系是以”在已排序的区间中对象值

的相对顺序”为基础的。标准关联容器是基于等价而不是相等。

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)来决定保持怎样的顺序。等价的定义正是通过该比较函数而确定的，因此，标准关联容器的使用者要为所使用的每个容器指定一个比较函数(用来决定如何排序)。

如果该关联容器使用相等来决定两个对象是否有相同的值，那么每个关联容器除了用于排序的比较函数外，还需要另一个比较函数来决定两个值是否相等。(默认情况下，该比较函数应该是 equal_to，但 equal_to 从来没有被用作 STL 的默认比较函数。当 STL 中需要相等判断时，一般的惯例是直接调用 operator ==)。

第 20 条：为包含指针的关联容器指定比较类型

每当你要创建包含指针的关联容器时，一定要记住，容器将会按照指针的值进行排序。绝大多数情况下，这不会是你所希望的，所以你几乎肯定要创建自己的函数子类作为该容器的比较类型(comparison type)。

如果你有一个包含智能指针或迭代器的容器，那么你也要考虑为它指定一个比较类型。对指针的解决方案同样也适用于那些类似指针的对象。

第 21 条：总是让比较函数在等值情况下返回 false

在 STL 中，对于 sort 函数中的排序算法，需要遵循严格弱序(strict weak ordering)的原则。调试跟踪定位发现 sort 的函数调用链最终会调用 __unguarded_partition

```
template<typename _RandomAccessIterator, typename _Tp, typename _Compare>
    _RandomAccessIterator
    __unguarded_partition(_RandomAccessIterator __first,
                          _RandomAccessIterator __last,
                          _Tp __pivot, _Compare __comp) {
      while (true) {
      while (__comp(*__first, __pivot))
        ++__first;
      --__last;
      while (__comp(__pivot, *__last))
        --__last;
      if (!(__first < __last))
        return __first;
      std::iter_swap(__first, __last);
      ++__first;
    }
   }
```

其中，__first 为迭代器，__pivot 为中间值，__comp 为传入的比较函数。 如果传入的 vector 中，按照之前的写法 >= 元素完全相等的情况下那么 __comp 比较函数一直是 true，那么后面 ++__first，最终就会使得迭代器失效，从而导致 coredump。 至此，分析完毕，请记住，STL sort 自定义比较函数，总是对相同值的比较返回 false。

第 22 条：切勿直接修改 set 或 multiset 中的键

```
int test_item_22() {
 std::map<int, std::string> m{ { 0, "xxx" } };
 //m.begin()->first = 10; // build error, map的键不能修改

 std::multimap<int, std::string> mm{ { 1, "yyy" } };
 //mm.begin()->first = 10; // build error, multimap的键同样不能修改

 std::set<int> s{ 1, 2, 3 };
 //*(s.begin()) = 10; // build error, set的键不能修改
 const_cast<int&>(*s.begin()) = 10; // 强制类型转换

 std::vector<int> v{ 1, 2, 3 };
 *v.begin() = 10;

 return 0;
}
```

set 和 multiset 按照一定的顺序来存放自己的元素，而这些容器的正确行为也是建立在其元素保持有序的基础之上的。如果你把关联容器中的一个元素的值改变了(比如把 10 改为 1000)，那么，新的值可能不在正确的位置上，这将

会打破容器的有序性。

对于 map 和 multimap 尤其简单，因为如果有程序试图改变这些容器中的键，它将不能通过编译。这是因为，对于一个 map<K, V>或 multimap<K, V> 类型的对象，其中的元素类型是 pair<const K, V>。因为键的类型是 const K，所以它不能被修改。(如果利用 const_cast，你或许可以修改它。)

对于 set 或 multiset 类型的对象，容器中元素的类型是 T，而不是 const T。注：不通过强制类型转换并不能改变 set 或 multiset 中的元素。

第 23 条：考虑用排序的 vector 替代关联容器

这个建议的前提是：

- 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。
- 查找阶段：查询该数据结构找到特点的信息，在这个阶段，几乎所有的操作都是查找，很少或几乎没有删除。
- 重组阶段：改变数据结构的内容。

这种方式使用其数据结构的应用程序来说，排序的 vector 可能比管理容器提供了更好的性能。

第 24 条：当效率至关重要的时候，请在 map::operator[] 和 map::insert 之间作慎重选择

当做“添加”操作时，insert 效率比 operator[] 更高

map::operator[] 工作原理：operator[] 返回一个引用，它指向与 k 相关联的值对象。然后 v 被赋给了该引用所指向的对象。如果键 k 已经有了相关联的值，则该值被更新。如果 k 还没有在映射表中，那就没有 operator[] 可以指向的值对象，这种情况下，它使用值类型的默认构造函数创建一个新的对象，然后 operator[] 就能返回一个指向该新对象的引用。

看一个样例：

```
map<int, Widget> m;

m[1] = 5.13
```

表达式 m[1] 是 m.operator 的缩写形式，所以这是对 map::operator[] 的调用。该函数必须返回一个指向 Widget 的引用，因为 m 所映射的值对象类型是

Widget。这时候 m 中什么也没有，所以键 1 没有多余的值对象。因此，operator[]默认构造了一个 Widget，作为 1 相关联的值，然后返回一个指向 Widget 的引用，最后，这个 Widget 赋值为 5.13

而如果直接用　m.insert(Widget::value_type(1,5.13))

这样直接用我们所需的值构造了一个 Widget 比先默认构造一个 Widget 在赋值效率更高。与之相比，通常节省了三个函数调用：一个用于创建默认构造的临时 Widget 对象，一个用以析构该临时对象，一个是调用 Widget 的赋值操作符。

当做“更新”操作时，operator[] 效率比 insert 更高

原因在于 insert 调用需要一个 Widget::value_type 类型的参数（pair<int,Widget>），所以当我们调用 insert 时候，必须构造和析构一个该类型的对象，这样付出一个 pair 构造函数和一个 pair 析构函数的代价。而这又会导致 Widget 的构造和析构，因为 pair<int,Widget> 本身包含了一个 Widget 对象，而 operator[] 不使用 pair 对象，所以它不会构造和析构任何 pair 和 Widget

第 25 条：熟悉非标准的散列容器

C++11 中新增了四种关联容器，使用哈希函数组织的，即 unordered_map、unordered_multimap、unordered_set、unordered_multiset。

- set：集合。底层为红黑树，元素有序，不重复；multiset：底层为红黑树，元素有序，可重复
- map：底层为红黑树，键有序，不重复；multimap：底层为红黑树，键有序，可重复
- unordered_set：底层为哈希表，无序，不重复；unordered_multiset：底层为哈希表，无序，可重复
- unordered_map：底层为哈希表，无序，不重复；unordered_multiap：底层为哈希表，无序，可重复

第四章 迭代器

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

STL 中的所有标准容器都提供了 4 种迭代器类型。

对容器类 container 而言，iterator 类型的功效相当于 T*，而 const_iterator 则相当于 const T*。对一个 iterator 或者 const_iterator 进行递增则可以移动到容器中的下一个元素，通过这种方式可以从容器的头部一直遍历到尾部。reverse_iterator 与 const_reverse_iterator 同样分别对应于 T 和 const T，所不同的是，对这两个迭代器进行递增的效果是由容器的尾部反向遍历到容器头部。

注意：vector::insert，对于 C++98 中，第一个参数均为 iterator；而对于 C++11 中，第一个参数均为 const_iterator。vector::erase 的情况也是这样。

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

distance 用以取得两个迭代器(它们指向同一个容器)之间的距离；advance 则用于将一个迭代器移动指定的距离。

```
int test_item_27() {
 typedef std::deque<int> IntDeque;
 typedef IntDeque::iterator Iter;
 typedef IntDeque::const_iterator ConstIter;

 IntDeque d(5, 10);
 ConstIter ci;
 ci = d.cbegin() + 1; // 使ci指向d
 Iter i(d.begin());
 std::advance(i, std::distance<ConstIter>(i, ci));

 return 0;
}
```

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

```
int test_item_28() {
 std::vector<int> v;
 v.reserve(5);

 for (int i = 1; i <= 5; ++i) v.push_back(i);

 std::vector<int>::reverse_iterator ri = std::find(v.rbegin(), v.rend(), 3); // 使ri指向3
 std::vector<int>::iterator i(ri.base());
 fprintf(stdout, "%d\n", (*i)); // 4
 v.insert(i, 99);
 for (auto it = v.cbegin(); it != v.cend(); ++it) fprintf(stdout, "value: %d\n", *it); // 1 2 3 99 4 5

 v.clear(); v.reserve(5);
 for (int i = 1; i <= 5; ++i) v.push_back(i);
 ri = std::find(v.rbegin(), v.rend(), 3);
 v.erase((++ri).base());
 for (auto it = v.cbegin(); it != v.cend(); ++it) fprintf(stdout, "value: %d\n", *it); // 1 2 4 5

 return 0;
}
```

如果要在一个 reverse_iterator ri 指定的位置上插入新元素，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。

第 29 条：对于逐个字符的输入请考虑使用

istreambuf_iterator

```
int test_item_29() {
 // 把一个文本文件的内容拷贝到一个string对象中
 std::ifstream inputFile("interestingData.txt");
 inputFile.unsetf(std::ios::skipws); // 禁止忽略inputFile中的空格
 std::string fileData((std::istream_iterator<char>(inputFile)), std::istream_iterator<char>()); // 速度慢

 std::string fileData2((std::istreambuf_iterator<char>(inputFile)), std::istreambuf_iterator<char>()); // 速度快

 return 0;
}
```

istream_iterator 对象使用 operator>>从输入流中读取单个字符，而 istreambuf_iterator 则直接从流的缓冲区中读取下一个字符。istreambuf_iterator 不会跳过任何字符，它只是简单地取回流缓冲区中的下一个字符，而不管它们是什么字符，因此用不着清除输入流的 skipws 标志。

第五章 算法

第 30 条：确保目标区间足够大

需求 1：希望像 transform 这样的算法把结果以新元素的形式插入到容器末尾开始。

```
int transmogrify(int x) { return (x + 1); }

int test_item_30() {
 std::vector<int> values{ 1, 2, 3 };
 std::vector<int> results;
 results.reserve(results.size() + values.size()); // 可避免内存的重新分配
 //std::transform(values.cbegin(), values.cend(), results.end(), transmogrify); // 错误，segmentation fault
 std::transform(values.cbegin(), values.cend(), std::back_inserter(results), transmogrify); // 正确 {2,4,6}
 // 在内部，std::back_inserter返回的迭代器将使得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

 std::list<int> results2;
 std::transform(values.cbegin(), values.cend(), std::front_inserter(results2), transmogrify);
 // std::front_inserter在内部利用了push_front，所以front_inserter仅适用于那些提供了push_front成员函数的容器

 return 0;
}
```

需求 2：假设希望 transform 这样的算法覆盖容器中已有的元素，那么就需要确保 result 已有的元素至少和 values 的元素一样多。否则，就必须使用 resize 来保证这一点

```
std::vector<int> values;
std::vector<int> results;
//...
if (results.size() < values.size()) {
 results.resize(values.size());//确保size一样大
}
std::transform(values.begin(), values.end(),results.begin(), fun);//覆盖 results 中前 values.size() 的元素
或者，也可以先清空 results 然后按照普通的方式使用一个插入行迭代器

results.clear();
results.reserve(values.size());
std::transform(values.begin(), values.end(), std::back_inserter(results), fun);//覆盖 results 中前 values.size() 的
```

无论何时，**如果所使用的算法需要指定一个目标区间，那么必须确保目标区间足够大，或者确保它会随着算法的运行而增大。要在算法执行过程中增大目标区间，请使用插入型迭代器**，比如 ostream_iterator 或者由 back_inserter、front_inserter 和 inserter 返回的迭代器。

第 31 条：了解各种与排序有关的选择

```
bool qualityCompare(const string& lhs, const string& rhs) {
 return (lhs < rhs);
}

bool hasAcceptableQuality(const string& w) {
 return true; // 判断 w 的质量值是否为 2 或者更好
}

int test_item_31() {
 vector<string> vec(50, "xxx");
 partial_sort(vec.begin(), vec.begin() + 20, vec.end(),
qualityCompare);
// 将质量最好的 20 个元素顺序放在 vec 的前 20 个位置上

 nth_element(vec.begin(), vec.begin() + 19, vec.end(),
qualityCompare);
// 将最好的 20 个元素放在 vec 的前部，但并不关心它们的具体排列顺序

 // partia_sort 和 nth_element 在效果上唯一不同之处在于：partial_sort
对位置 1--20 中的元素进行了排序，而 nth_element 没有对它们进行排序。然
而，这两个算法都将质量最好的 20 个 vec 放到了矢量的前部

 vector<string>::iterator begin(vec.begin());
 vector<string>::iterator end(vec.end());
 vector<string>::iterator goalPosition;
// 用于定位感兴趣的元素

// 找到具有中间质量级别的 string
 goalPosition = begin + vec.size() / 2;
// 如果全排序的话，待查找的 string 应该位于中间
 nth_element(begin, goalPosition, end, qualityCompare);
// 找到 vec 的中间质量值
// 现在 goalPosition 所指的元素具有中间质量

// 找到区间中具有 75%质量的元素
 vector<string>::size_type goalOffset = 0.25 * vec.size(); // 找出如
果全排序的话，待查找的 string 离起始处有多远
 nth_element(begin, begin + goalOffset, end, qualityCompare);
// 找到 75%处的质量值
```

```
 // 将满足 hasAcceptableQuality 的所有元素移到前部，然后返回一个迭代器，指向第一个不满足条件的 string
vector<string>::iterator goodEnd = partition(vec.begin(), vec.end(), hasAcceptableQuality);

 return 0;
}
```

nth_element：用于排序一个区间，它使得位置 n 上的元素正好是全排序情况下的第 n 个元素。而且，当 nth_element 返回的时候，所有按全排序规则(即 sort 的结果)排在位置 n 之前的元素也都被排在位置 n 之前，而所有按全排序规则排在位置 n 之后的元素则都被排在位置 n 之后。

partial_sort 和 nth_element 在排列等价元素的时候，有它们自己的做法，你无法控制它们的行为。

partial_sort、nth_element 和 sort 都属于非稳定的排序算法，但是有一个名为 stable_sort 的算法可以提供稳定排序特性。

nth_element 除了可以用来找到排名在前的 n 个元素以外，它还可以用来找到一个区间的中间值，或者找到某个特定百分比上的值。

partition：可以把所有满足某个特定条件的元素放在区间的前部。

如果需要对 vector、string、deque 或者数组中的元素执行一次完全排序，那么可以使用 sort 或者 stable_sort。

如果有一个 vector、string、deque 或者数组，并且只需要对等价性最前面的 n 个元素进行排序，那么可以使用 partial_sort。

如果有一个 vector、string、deque 或者数组，并且需要找到第 n 个位置上的元素，或者，需要找到等价性前面的 n 个元素但又不必对这 n 个元素进行排序，那么，nth_element 正是你所需要的函数。

如果需要将一个标准序列容器中的元素按照是否满足某个特定的条件区分开来，那么，partition 和 stable_partition 可能正是你所需要的。

如果你的数据在一个 list，那么你仍然可以直接调用 partition 和 stable_partition 算法；你可以用 list::sort 来替代 sort 和 stable_sort 算法。但是，如果你需要获得 partial_sort 或 nth_element 算法的效果，那么，你可以有一些间接的途径来完成这项任务。

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

记住一句话：remove 不是真正意义上的删除，因为它做不到。

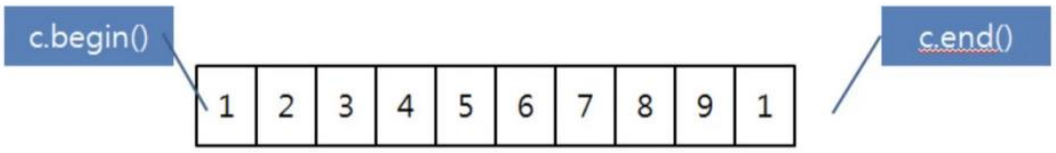
remove 的原理：移动了区间中的元素，将 “不用被删除”的元素在 v.begin()和 newEnd 之间，“需要被删除”的元素在 newEnd 和 v.end()之间。它返回的迭代器是指向最后一个“不用被删除”的元素之后的元素。这个返回值相当于该区间“新的逻辑结尾”。

```
int test_item_32() {
 std::vector<int> v;
 v.reserve(10);
 for (int i = 1; i <= 10; ++i) v.push_back(i);
 fprintf(stdout, "v.size: %d\n", v.size()); // 输出10
 v[3] = v[5] = v[9] = 99;
 std::remove(v.begin(), v.end(), 99); // 删除所有值等于99的元素
 fprintf(stdout, "v.size: %d\n", v.size()); // 仍然输出10
 for (auto i : v) fprintf(stdout, "%d\n", i);

 v.erase(std::remove(v.begin(), v.end(), 99), v.end()); // 真正删除所有值等于99的元素

 return 0;
}
```

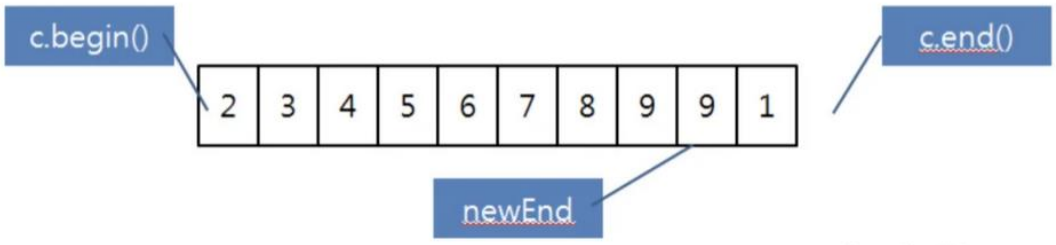
调用 remove 之前，c 的布局如下：



调用 remove 之后：

```
vector<int>::iterator newEnd(remove(c.begin(), c.end(), 1));
```

c 的布局如下



remove 并不接受容器作为参数，所以 remove 并不知道这些元素被存放在哪个容器中。并且，remove 也不可能推断出是什么容器，因为无法从迭代器推知对应的容器类型。因为从容器中删除元素的唯一方法是调用该容器的成员函数，

而 remove 并不知道它操作的元素所在的容器，所以 remove 不可能从容器中删除元素。

std::list 的 remove 成员函数是 STL 中唯一一个名为 remove 并且确实删除了容器中元素的函数。

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

当容器中存放的是指向动态分配的对象的指针的时候，应该避免使用 remove 和类似的算法(remove_if 和 unique)。 如果容器中存放的不是普通指针，而是具有引用计数功能的智能指针，那么就可以直接使用 erase-remove 的习惯用法。

容器中的元素为 1 2 3 4 3 5，调用 remove 算法删除的元素为 3，
容器中元素会变成：1 2 4 5 3 5。
之前容器中的第一个 3 将会被直接覆盖掉。
想想如果这些都是堆内存指针，同时这些指针没有其他拷贝，那么就会出现内存泄露，因为已经没有指针指向 3 指向的堆内存。

```
class Widget33 {
 public:
 bool isRemove() const { return true; }
};

//如果 pwidget 是一个未被验证的 Widget33 则删除该指针，并置位空
void delAndNullifyUnremove(Widget33*& pWidget) {
 if (!pWidget->isRemove()) {
  delete pWidget;
  pWidget = nullptr;
 }
}
int test_item_33() {
 std::vector<Widget33*> v;
 for (int i = 0; i < 5; ++i) v.push_back(new Widget33);

 // 删除那些指向未被验证过的Widget33对象的指针，会资源泄露
 v.erase(std::remove_if(v.begin(), v.end(), std::not1(std::mem_fun(&Widget33::isCertified))), v.end());

 // 一种可以消除资源泄露的做法
 // 将所有指向未被验证的Widget33对象的指针删除并置成空
 std::for_each(v.begin(), v.end(), delAndNullifyUnremove);
 // 删除v中的空指针，必须将0转换成一个指针，这样C++才能正确推断出remove的第三个参数类型
 v.erase(std::remove(v.begin(), v.end(), static_cast<Widget33*>(0)), v.end());

 // 使用智能指针可防止资源泄露
 std::vector<std::shared_ptr<Widget33>> v2;
 for (int i = 0; i < 5; ++i) v2.push_back(std::make_shared<Widget33>());
 // 下面语句需要编译器必须能够把智能指针类型std::shared<Widget33>隐式转换为对应的内置指针类型Widget33*才能通过编译
 //v2.erase(std::remove_if(v2.begin(), v2.end(), std::not1(std::mem_fun(&Widget33::isCertified))), v2.end());

 return 0;
}
```

第 34 条：了解哪些算法要求使用排序的区间作为参数

并非所有的算法都可以应用于任何区间。举例来说，remove 算法要求单向迭代器并且要求可以通过这些迭代器向容器中的对象赋值。所以，它不能用于由输入迭代器指定的区间，也不适用于 map 或 multimap，同样不适用于某些 set 和 multiset 的实现。同样地，很多排序算法要求随机访问迭代器，所以对于 list 的元素不可能调用这些算法。有些算法要求排序的区间，即区间中的值是排过序的。有些算法既可以与排序的区间一起工作，也可以与未排序的区间一起工作，但是当它们作用在排序的区间上时，算法会更加有效。

要求排序区间的 STL 算法：binaray_search、lower_bound、upper_bound、equal_range、set_union、set_intersection、set_difference、set_symmetric_difference、merge、inplace_merge、includes。

unique、unique_copy 并不一定要求排序的区间，但通常情况下会与排序区间一起使用。

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

std::lexicographical_compare 是 strcmp 的一个泛化版本。不过，strcmp 只能与字符数组一起工作，而 lexicographical_compare 则可以与任何类型的值的区间一起工作。而且，strcmp 总是通过比较两个字符来判断它们的关系相等、小于还是大于，而 lexicographical_compare 则可以接受一个判别式，由该判别式来决定两个值是否满足一个用户自定义的准则。

strcmp 通常是被优化过的，它们在字符串的处理上一般要比通用算法 mismatch 和 lexicographical_compare 快。

第 36 条：理解 copy_if 算法的正确实现

std::distance 函数接受两个迭代器参数，first 和 last，并返回它们之间的距离。返回值的类型是由迭代器的类型和特征决定的（通常是整形）。

```
int test_item_36() {
 std::vector<int> v1{ 1, 2, 3, 4, 5 }, v2(v1.size());

 auto it = std::copy_if(v1.begin(), v1.end(), v2.begin(), [](int i) { return (i % 2 == 1); });
 v2.resize(std::distance(v2.begin(), it));

 for (const auto& v : v2)
  fprintf(stdout, "%d\n", v); // 1 3 5

 return 0;
}
```

C++11 中增加了 std::copy_if 函数。拷贝带条件判断的算法。

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

std::accumulate 有两种形式：第一种形式有两个迭代器和一个初始值，它返回该初始值加上由迭代器标识的区间中的值的总和。

std::accumulate 只要求输入迭代器，所以你可以使用 std::istream_iterator 和 std::istreambuf_iterator。

std::accumulate 的第二种形式带一个初始值和一个任意的统计函数。

std::for_each 是另一个可被用来统计区间的算法，而且它不受 accumulate 的那些限制。如同 accumulate 一样，for_each 也带两个参数：一个是区间，另一个是函数(通常是函数对象)----对区间中的每个元素都要调用这个函数，但

是，传给 for_each 的这个函数只接收一个实参(即当前的区间元素)。for_each 执行完毕后会返回它的函数。(实际上，它返回的是这个函数的一份拷贝。)重要的是，传给 for_each 的函数(以及后来返回的函数)可以有副作用。

std::for_each 和 std::accumulate 在两个方面有所不同：首先，名字 accumulate 暗示着这个算法将会计算出一个区间的统计信息。而 for_each 听起来就好像是对一个区间的每个元素做一个操作。用 for_each 来统计一个区间是合法的，但是不如 accumulate 来得清晰。其次，accumulate 直接返回我们所要的统计结果，而 **for_each 却返回一个函数对象**，我们必须从这个函数对象中提取出我们所要的统计信息。在 C++中，这意味着我们必须在函数子类中 加入一个成员函数，以便获得我们想要的统计信息。

accumulate，可以直接计算数组或容器中 C++内置数据类型。对于自定义数据类型，accumulate 提供了回调函数（第四个参数），来实现自定义数据的处理。

```
// accmulate原型
template<class _Init,
    class _Ty,
    class _Fn2> inline
    _Ty _Accmulate(_Init _First, _Init _Last, _Ty _Val, _Fn2 _Func)
{
    for (; _First != _Last; ++_First) {
        // _Func为回调函数
        _Val = _Func(_Val, *_First);
    }

    return _Val;
}
```

第六章 仿函数

第 38 条：遵循按值传递的原则来设计函数子类

无论是 C 还是 C++，都不允许将一个函数作为参数传递给另一个函数，相反，你必须传递函数指针。C 和 C++的标准库函数都遵循这一规则：**函数指针是按值传递的。**

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(或者可以隐式地转换为 bool 类型)的函数。在 STL 中，判别式有着广泛的用途。标准关联容器的比较函数就是判别式；对于像 find_if 以及各种与排序有关的算法，判别式往往也被作为参数来传递。

第 40 条：若一个类是函数子(functor)，则应使它可配接（adaptable）

```
public:
    int mx, my;
    //const int yy;
    Widget(int x = 0, int y = 0) :mx(x), my(y) {}
    void testfunc()const {
        cout << "call testfunc!" << endl;
    }
};
//自定义一个判断是否interesting的Pred
bool isInteresting(const shared_ptr<Widget>& pw) {
    if (pw->mx + pw->my > 6)return true;//此时认为widget是足够有趣的!
    return false;//此时认为widget是无趣的!
}
```

```
//注意！这里你不能直接用!（非号）直接取反！因为这样是无法通过编译的!
//必须要用not1(ptr_fun(Pred));这种形式才能do到将函数子Functor适配！！！
 auto it2 =     find_if(widgetPtrs.begin(),widgetPtrs.end(),not1(ptr_fun(isInteresting)));
 if(it2 != widgetPtrs.end()){
     cout<<"find the first un intersting one!"<<endl;
     cout<<"and it is "<<" mx = "<<(*it2)->mx<<" my = "<<(*it2)->my<<endl;
 }
 else cout<<"can not find it!"<<endl;
 return 0;
```

第 41 条：理解 ptr_fun、men_fun 和 mem_fun_ref 的来由

std::ptr_fun：将函数指针转换为函数对象。

std::mem_fun：将成员函数转换为函数对象(指针版本)。

std::mem_fun_ref：将成员函数转换为函数对象(引用版本)。

第 42 条：确保 less 与 operator<具有相同的语义

应该尽量避免修改 less 的行为，因为这样做很可能会误导其他的程序员。如果你使用了 less，无论是显式地或是隐式地，你都需要确保它与 operator<具有相同的意义。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。

第七章 使用 STL 编程

第 43 条：算法调用优先于手写的循环

理由：

- 效率：算法通常比程序员自己写的循环效率更高。
- 正确性：自己写循环比使用算法更容易出错。
- 可维护性：使用算法的代码通常比手写循环的代码更加简洁明了。

如果你要做的工作与一个算法所实现的功能很相近，那么用算法调用更好。但是如果你的循环很简单，而若使用算法来实现的话，却要求混合使用绑定器和配接器或者要求一个单独的函数子类，那么，可能使用手写的循环更好。最后，如果你在循环中要做的工作很多，而且又很复杂，则最好使用算法调用。

第 44 条：容器的成员函数优先于同名的算法

有些 STL 容器提供了一些与算法同名的成员函数。比如，关联容器提供了 count、find、lower_bound、upper_bound 和 equal_range，而 list 则提供了 remove、remove_if、unique、sort、merge 和 reverse。

大多数情况下，你**应该使用这些成员函数，而不是相应的 STL 算法**。这里有两个理由：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)结合得更加紧密，这是算法所不能比的。

原因在于，算法和成员函数虽然有同样的名称，但是它们所做的事情往往不完全相同。

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

如果区间是排序的，那么通过 binary_search、lower_bound、upper_bound 和 equal_range，你可以获得更快的查找速度(通常是对数时间的效率)。 如果迭代器并没有指定一个排序的区间，那么你的选择范围将局限于 count、count_if、find 以及 find_if，而这些算法仅能提供线性时间的效率。

第 46 条：考虑使用函数对象而不是函数作为 STL 算法的参数

在 C/C++中并不能真正地将一个函数作为参数传递给另一个函数。如果我们试

图将一个函数作为参数进行传递，则编译器会隐式地将它转换成一个指向该函数的指针，并将该指针传递过去。**函数指针参数抑制了内联机制**。

例如：一个指向函数的指针，每次在 sort 中用到时，编译器产生一个间接函数调用即通过指针调用。大部分编译器不会试图去内联通过函数指针调用的函数

第 47 条：避免产生直写型(write-only)的代码

当你编写代码的时候，它看似非常直接和简捷，因为它是由某些基本想法(比如，erase-remove 习惯用法加上在 find 中使用 reverse_interator 的概念)自然而形成的。

然而，阅读代码的人却很难将最终的语句还原成它所依据的思路，虽然很容易编写，但是难以阅读和理解。

第 48 条：总是包含(#include)正确的头文件

C++标准与 C 的标准有所不同，它没有规定标准库中的头文件之间的相互包含关系。

总结每个与 STL 有关的标准头文件中所包含的内容：

几乎所有的标准 STL 容器都被声明在与之同名的头文件中，比如 vector 被声明在中，list 被声明在中。

除了 4 个 STL 算法以外，其它所有的算法都被声明在<algorithm>中，例外的是这 4 个算法 accumulate、inner_product、adjacent_difference 和 partial_sum，它们被声明在<numeric>中。

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

标准的仿函数函数(比如 less)和仿函数配接器(比如 not1、bind2nd)被声明在<functional>中。

任何时候如果你使用了某个头文件中的一个 STL 组件，那么你就一定要提供对应的#include 指令，即使你正在使用的 STL 平台允许你省略#include 指令，你也要将它们包含到你的代码中。当你需要将代码移植到其它平台上的时候，移植的压力就会减轻。

第 49 条：学会分析与 STL 相关的编译器诊断信息

一些技巧：

- vector 和 string 的迭代器通常就是指针，所以当错误地使用了 iterator 的时候，编译器的诊断信息中可能会引用到指针类型。例如，如果源代码中引用了 vector::iterator，那么编译器的诊断信息中极有可能就会提及 double*指针。
- 如果诊断信息中提到了 back_insert_iterator、front_insert_iterator 或者 insert_iterator，则几乎总是意味着你错误地调用了 back_inserter、front_inserter 或者 inserter。如果你并没有直接调用这些函数，则一定是你所调用的某个函数直接或者间接地调用了这些函数。
- 类似地，如果诊断信息中提到了 binder1st 或者 binder2nd，那么你可能是错误地使用了 bind1st 和 bind2nd。
- 输出迭代器(如 ostream_iterator、ostreambuf_iterator 以及那些由 back_inserter、front_inserter、front_inserter 和 inserter 函数返回的迭代器)在赋值操作符内部完成其输出或者插入工作，所以，如果在使用这些迭代器的时候犯了错误，那么你所看到的错误消息中可能会提到与赋值操作符有关的内容。
- 如果你得到的错误消息来源于某一个 STL 算法的内部实现(例如，引起错误的源代码在中)，那也许是你在调用算法的时候使用了错误的类型。例如，你可能使用了不恰当的迭代器类型。
- 如果你正在使用一个很常见的 STL 组件，比如 vector、string 或者 for_each 算法，但是从错误消息来看，编译器好像对此一无所知，那么可能是你没有包含相应的头文件。

第 50 条：熟悉与 STL 相关的 Web 站点