[返回主页](http://www.cnblogs.com/answeryi/)

[Play every game as if it's your last.](http://www.cnblogs.com/answeryi/)

Keep a diary, someday, it will keep you ...

* [首页](http://www.cnblogs.com/answeryi/)
* [博问](http://q.cnblogs.com/)
* [闪存](http://home.cnblogs.com/ing/)
* [新随笔](http://i.cnblogs.com/EditPosts.aspx?opt=1)
* [订阅](http://www.cnblogs.com/answeryi/rss)
* [管理](http://i.cnblogs.com/)

[C++ 容器及选用总结](http://www.cnblogs.com/answeryi/archive/2011/12/16/2289811.html)

目录

====================================================

第一章 容器

第二章 Vector和string

第三章 关联容器

第四章 迭代器

第五章 算法

第六章 函数

第七章 在程序中使用STL

====================================================

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。(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。

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。

**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（包括数据类型和形参名）用括号括起来是非法的。

**第7条：如果容器中包含通过new操作创建的指针，切记在容器对象析构前将指针delete。**

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，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”除去多余的容量。**

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。**

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未排序排序set或mapmultiset或multimap特定的值存在吗findbinary\_searchcountfind特定值存在吗？如果有，第一个在哪findequal\_rangefindfind或lower\_bound第一个不超过特定值的对象在哪里find\_iflower\_boundlower\_boundlower\_bound第一个超过某个特定值的对象在哪里find\_ifupper\_boundupper\_boundupper\_bound具有特定值的对象有多少个countequal\_range (然后distance)countcount具有特定值的对象都在哪里find(反复调用)equal\_rangeequal\_rangeequal\_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> >）。

第50条：熟悉STL相关的Web站点。

SGI STL

站点：http://www.sig.com/tech/stl/

STLport

站点：http://stlport.org

BOost

站点：http://boost.org

标签: [C++](http://www.cnblogs.com/answeryi/tag/C%2B%2B/), [容器](http://www.cnblogs.com/answeryi/tag/%E5%AE%B9%E5%99%A8/)

绿色通道：[**好文要顶关注我收藏该文**](javascript:void(0);)[**与我联系**](http://space.cnblogs.com/msg/send/Keosu)**[http://static.cnblogs.com/images/icon_weibo_24.png](javascript:void(0);)**

[http://pic.cnitblog.com/face/u52275.jpg?id=17212244](http://home.cnblogs.com/u/answeryi/)

[Keosu](http://home.cnblogs.com/u/answeryi/)  
[关注 - 0](http://home.cnblogs.com/u/answeryi/followees)  
[粉丝 - 9](http://home.cnblogs.com/u/answeryi/followers)

[+加关注](javascript:void(0);)

6

0

(请您对文章做出评价)

[«](http://www.cnblogs.com/answeryi/archive/2011/12/12/2285014.html) 上一篇：[gawk](http://www.cnblogs.com/answeryi/archive/2011/12/12/2285014.html)  
[»](http://www.cnblogs.com/answeryi/archive/2012/04/16/2452877.html) 下一篇：[设计模式分类及描述](http://www.cnblogs.com/answeryi/archive/2012/04/16/2452877.html)

posted @ 2011-12-16 10:17 [Keosu](http://www.cnblogs.com/answeryi/) 阅读(15519) 评论(1) [编辑](http://i.cnblogs.com/EditPosts.aspx?postid=2289811) [收藏](http://www.cnblogs.com/answeryi/archive/2011/12/16/2289811.html)

评论列表

[#1楼](http://www.cnblogs.com/answeryi/archive/2011/12/16/2289811.html#2591556)25915562013/1/3 22:23:042013-01-03 22:23 [长江歌声](http://www.cnblogs.com/changjianggesheng/)

好文章！不是很懂对有些东西！

[支持(0)反对(0)](javascript:void(0);)

[刷新评论](javascript:void(0);)[刷新页面](http://www.cnblogs.com/answeryi/archive/2011/12/16/2289811.html)[返回顶部](http://www.cnblogs.com/answeryi/archive/2011/12/16/2289811.html#top)

**注册用户登录后才能发表评论，请** [**登录**](javascript:void(0);) **或** [**注册**](javascript:void(0);)**，**[**访问**](http://www.cnblogs.com)**网站首页。**

[【免费课程】案例：CSS圆角进化论](http://www.imooc.com/view/118?from=cnblogs)

**最新IT新闻**:  
· [Novena笔记本项目的教训](http://news.cnblogs.com/n/502324/)  
· [Linux诞生23周年，Linux 3.17-rc2发布](http://news.cnblogs.com/n/502323/)  
· [Mono 3.6发布改进的调试器](http://news.cnblogs.com/n/502322/)  
· [今年的秋老虎还咬人吗？](http://news.cnblogs.com/n/502321/)  
· [2.57米 世界第一高人去世：身患巨人症](http://news.cnblogs.com/n/502320/)  
» [更多新闻...](http://news.cnblogs.com/)

**最新知识库文章**:

· [中国软件开发工程师之痛](http://kb.cnblogs.com/page/183773/)  
· [6 个重构方法可帮你提升 80% 的代码质量](http://kb.cnblogs.com/page/199831/)  
· [有关云架构建设和选型的思考](http://kb.cnblogs.com/page/502002/)  
· [程序员生存定律-打造属于自己的稀缺性](http://kb.cnblogs.com/page/500844/)  
· [防止代码变质的思考与方法](http://kb.cnblogs.com/page/501814/)

» [更多知识库文章...](http://kb.cnblogs.com/)

搜索





我的标签

* [C++](http://www.cnblogs.com/answeryi/tag/C%2B%2B/)(8)
* [算法](http://www.cnblogs.com/answeryi/tag/%E7%AE%97%E6%B3%95/)(3)
* [数据结构](http://www.cnblogs.com/answeryi/tag/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/)(2)
* [设计模式](http://www.cnblogs.com/answeryi/tag/%E8%AE%BE%E8%AE%A1%E6%A8%A1%E5%BC%8F/)(2)
* [code](http://www.cnblogs.com/answeryi/tag/code/)(2)
* [Matlab](http://www.cnblogs.com/answeryi/tag/Matlab/)(2)
* [PyQt4](http://www.cnblogs.com/answeryi/tag/PyQt4/)(2)
* [Python GUI](http://www.cnblogs.com/answeryi/tag/Python%20GUI/)(2)
* [Python图形界面](http://www.cnblogs.com/answeryi/tag/Python%E5%9B%BE%E5%BD%A2%E7%95%8C%E9%9D%A2/)(2)
* [scc](http://www.cnblogs.com/answeryi/tag/scc/)(1)
* [更多](http://www.cnblogs.com/answeryi/tag/)

随笔分类(44)

* [Algorithm(3)](http://www.cnblogs.com/answeryi/category/241639.html)
* [C++(6)](http://www.cnblogs.com/answeryi/category/395368.html)
* [Code Snippet(7)](http://www.cnblogs.com/answeryi/category/325885.html)
* [ComplexNetwork(2)](http://www.cnblogs.com/answeryi/category/214738.html)
* [Debug(1)](http://www.cnblogs.com/answeryi/category/199794.html)
* [Experience(1)](http://www.cnblogs.com/answeryi/category/214127.html)
* [Java(8)](http://www.cnblogs.com/answeryi/category/199796.html)
* [MFC(1)](http://www.cnblogs.com/answeryi/category/214252.html)
* [Others(12)](http://www.cnblogs.com/answeryi/category/199797.html)
* [Win32(3)](http://www.cnblogs.com/answeryi/category/199795.html)

阅读排行榜

* [1. C++ 容器及选用总结(15519)](http://www.cnblogs.com/answeryi/archive/2011/12/16/2289811.html)
* [2. WinDbg 源码模式调试(6179)](http://www.cnblogs.com/answeryi/archive/2009/07/14/1523662.html)
* [3. Htmlparser Filter 简要归纳(2273)](http://www.cnblogs.com/answeryi/archive/2009/11/03/1595509.html)
* [4. HtmlParser 简单测试实例(2085)](http://www.cnblogs.com/answeryi/archive/2009/11/02/1594598.html)
* [5. 【转】小世界网络的概念(1801)](http://www.cnblogs.com/answeryi/archive/2009/10/19/1586221.html)
* [6. 找出数组中唯一的重复元素(1524)](http://www.cnblogs.com/answeryi/archive/2012/09/24/2700529.html)
* [7. HtmlParser 简介(1432)](http://www.cnblogs.com/answeryi/archive/2009/11/02/1594543.html)
* [8. PyQT 入门（2）：实现自己的界面类(1408)](http://www.cnblogs.com/answeryi/archive/2012/09/27/2706184.html)
* [9. 搜索引擎基本原理(1292)](http://www.cnblogs.com/answeryi/archive/2012/09/30/2709291.html)
* [10. WinDbg(1225)](http://www.cnblogs.com/answeryi/archive/2009/07/08/1519519.html)

Copyright ©2014 Keosu