条款 1：仔细选择你的容器

- vector 是一种可以默认使用的序列类型，当很频繁地对序列中部进行插入和删除时应该用 list，当大部分插入和删除发生在序列的头或尾时可以选择 deque 这种数据结构。
- 连续内存容器：vector、string 和 deque，非标准的 rope。插入和删除时，影响效率和异常安全。
- 基于节点的容器：list、所有关联容器、非标准的 slist。插入或删除时，元素值不需要移动。

条款 2：小心对“容器无关代码”的幻想

- 并非所有容器的接口都一样，比如序列容器和关联容器。
- 容器相关的迭代器不一致，比如随机，双向，单向的区别。

条款 3：使容器里对象的拷贝操作轻量而正确

- 利用指针来避免昂贵的复制
- 利用 shared_ptr 来管理指针

条款 4：用 empty 来代替检查 size()是否为 0

- 对于所有的标准容器，empty 是一个常数时间的操作，但对于一些 list 实现，size 花费线性时间。

条款 5：尽量使用区间成员函数代替它们的单元素兄弟

- 当处理标准序列容器时，应用单元素成员函数比完成同样目的的区间成员函数需要更多地内存分配，更频繁地拷贝对象，而且/或者造成多余操作。

```
v1.assign(v2.begin() + v2.size() / 2, v2.end());

v1.insert(v1.end(), v2.begin() + v2.size() / 2, v2.end());

copy(data, data + numValues, inserter(v, v.begin()));

vector<int>::iterator insertLoc(v.begin());
for (int i = 0; i < numValues; ++i) {
        insertLoc = v.insert(insertLoc, data[i]);
        ++insertLoc;
}
```

1. copy 函数同等于显示循环
2. 每次调用 insert 会使 insertLoc 无效，上面处理可以避免迭代器无效。
3. 相比单元素插入策略，区间 insert 少执行了 n*(numValues-1)次移动。

- 区间构造（所有容器）：container::container(InputIterator begin, InputIterator end);

- 区间插入：

  （标准序列容器）void container::insert(iterator position, InputIterator begin, InputIterator end);

  （标准关联容器）void container::insert(InputIterator begin, InputIterator end);

- 区间删除：

  （标准序列容器）iterator container::erase(iterator begin, iterator end);

  （标准关联容器）void container::erase(iterator begin, iterator end);

- 区间赋值：void container::assign(InputIterator begin, InputIterator end)

条款 6：警惕 C++最令人恼怒的解析

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

- 这声明了一个函数 data，它的返回类型是 list<int>。第一个参数 dataFile，类型是 istream_iterator<int>，括号是多余的而且被忽略。第二个参数没有名字，类型是指 istream_iterator<int>。
- 解决办法 1（不可移植）：

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

- 解决办法 2（命名迭代器对象的使用和 STL 编程风格相反，但是一个值得付出的代价）：

```
ifstream dataFile("ints.dat");
istream_iterator<int> dataBegin(dataFile);
istream_iterator<int> dataEnd;
list<int> data(dataBegin, dataEnd);
```

条款 7：当使用 new 指针的容器时，记得在销毁容器前 delete 那些指针

- 可以用 shared_ptr 避免

条款 8：永不建立 auto_ptr 的容器

条款 9：在删除选项中仔细选择

- 连续内存容器：c.erase(remove(c.begin(), c.end(), 1963), c.end());
- list：成员函数 remove 更高效 c.remove(1963);

- 关联容器：c.erase(1963);
- 需要日志的删除

1. vector/string/deque

```
for (SeqContainer<int>::iterator i = c.begin();
        i != c.end();){
        if (badValue(*i)){
                logFile << "Erasing " << *i << '\n';
                i = c.erase(i);                    // 通过把 erase 的返回值
        }                                           // 赋给 i 来保持 i 有效
        else
                ++i;
}
```

2. 关联容器

```
for (AssocContainer<int>::iterator i = c.begin();        // 循环条件和前面一样
        i !=c.end();){
        if (badValue(*i)){
                logFile << "Erasing " << *i <<'\n';      // 写日志文件
                c.erase(i++);                      // 删除元素
        }
        else ++i;
}
```

3. list

以上两种都适用

条款 10：注意分配器的协定和约束

- 有状态的分配器可以很好地编译。它们只是不按你期待的方式运行。确保一个给定类型的所有分配器都等价是你的责任(无状态)。如果你违反这个限制，不要期待编译器发出警告。

条款 11：理解自定义分配器的正确用法

```
class Heap1 {
public:
        ...
        static void* alloc(size_t numBytes, const void
*memoryBlockToBeNear);
        static void dealloc(void *ptr);
        ...
};
class Heap2 { ... };              // 有相同的 alloc/dealloc 接口
template<typenameT, typename Heap>
```

```
class SpecificHeapAllocator {
public:
       pointer allocate(size_type numObjects, const void
*localityHint = 0){
              return
static_cast<pointer>(Heap::alloc(numObjects * sizeof(T),
                                                  localityHint));
       }
       void deallocate(pointer ptrToMemory, size_type
numObjects){
              Heap::dealloc(ptrToMemory);
       }
       ...
};
//用例
vector<int, SpecificHeapAllocator<int, Heap1 > > v;
set<int, SpecificHeapAllocator<int, Heap1 > > s;
list<Widget, SpecificHeapAllocator<Widget, Heap2> > L;
map<int, string, less<int>, SpecificHeapAllocator<pair<const
int, string>, Heap2> > m;
```

条款 12：对 STL 容器线程安全性的期待现实一些

- 多个读取者是安全的。
- 对不同容器的多个写入者是安全的。

条款 13：尽量使用 vector 和 string 来代替动态分配的数组

- 一个程序员的优化就是其他人的抱怨，而且如果你在多线程环境中使用了引用计数的字符串，你可能发现避免分配和拷贝所节省下的时间都花费在后台并发控制上了。
- 如果你的 base_string 使用了引用计数，多线程环境下可以考虑使用 vector<char>来代替 string，vector 实现不允许使用引用计数，所以隐藏的多线程性能问题不会出现了。

条款 14：使用 reserve 来避免不必要的重新分配

- 第一个可用的情况是当你确切或者大约知道有多少元素将最后出现在容器中。

```
vector<int> v;
for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

上面代码在循环过程中将会导致 2 到 10 次重新分配，下面代码是 0 次

```
vector<int> v;
v.reserve(1000);
for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

- 第二种情况是保留你可能需要的最大的空间，然后，一旦你添加完全部数据，修整掉任何多余的容量。

```
v.reserve(1000);
for (int i = 1; i <= n; ++i) v.push_back(i);
```

```
vector <string>(v).swap(v);
```

条款 15：小心 string 实现的多样性

- 字符串值可能是或可能不是引用计数的。默认情况下，很多实现的确是用了引用计数，但它们通常提供了关闭的方法，一般是通过预处理器宏。引用计数只对频繁拷贝的字符串有帮助，而有些程序不经常拷贝字符串，所以没有那个开销。
- string 对象的大小可能从 1 到至少 7 倍 char*指针的大小。
- 新字符串值的建立可能需要 0、1 或 2 次动态分配。
- string 对象可能是或可能不共享字符串的大小和容量信息。
- string 可能是或可能不支持每对象配置器。
- 不同实现对于最小化字符缓冲区的配置器有不同策略。

条款 16：如何将 vector 和 string 的数据传给遗留的 API

- vector to char *:

```
vector<int> v;
void doSomething(const int* pInts, size_t numInts);
if (!v.empty())
    doSomething(&v[0], v.size()); // doSomething(&*v.begin(),
v.size());
}
```

- string to char *: c_str()

条款 17：使用“交换技巧”来修整过剩容量

- 收缩到合适

```
vector<string>(v).swap(v);
string(s).swap(s)
```

- 清除值和容量

```
vector<string>().swap(v);          // 清除 v 而且最小化它的容量
string().swap(s);                  // 清除 s 而且最小化它的容量
```

条款 18：避免使用 vector<bool>

- 因为 vector<bool>是一个伪容器，并不保存真正的 bool，而是打包 bool 以节省空间。在一个典型的实现中，每个保存在“vector”中的“bool”占用一个单独的比特，而一个 8 比特的字节将容纳 8 个“bool”。在内部，vector<bool>使用了与位域（bitfield）等价的思想来表示它假装容纳的 bool。

```
vector<bool> v;
bool *pb = &v[0]; //意味这是一个单 bit 的引用，编译不会通过
```

条款 19：了解相等和等价的区别

- STL 算法（find）对“相同”的定义是相等，基于 operator==。
- 关联容器（set.insert）对“相同”的定义是等价，通常基于 operator<(比如!X < Y && !y < x)。

条款 20：为指针的关联容器指定比较类型

```
struct StringPtrLess: public binary_function<const string*,
const string*,bool> {
        bool operator()(const string *ps1, const string *ps2)
const {
                return *ps1 < *ps2;
        }
};
typedef set<string*, StringPtrLess> StringPtrSet;
StringPtrSet ssp;
//方式 1
void print(const string *ps){
        cout << *ps << endl;
}
for_each(ssp.begin(), ssp.end(), print);
//方式 2
struct Dereference {
        template <typename T>
        const T& operator()(const T *ptr) const        {
                return *ptr;
        }
};
transform(ssp.begin(), ssp.end(),
ostream_iterator<string>(cout, "\n"), Dereference());
```

条款 21：永远让关联容器和 sort 算法的比较函数对相等的值返回 false

- 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”（传给 sort 等算法的比较函数也有同样的限制）
- 因为 == 是用 !(x<y) && !(y<x) 来实现的

条款 22：避免原地修改 set 和 multiset 的键

- 这对于 map 和 multimap 特别简单，因为试图改变这些容器里的一个键值的程序将不能编译：

```
map<int, string> m;
...
m.begin()->first = 10;                    // 错误！map 键不能改变
multimap<int, string> mm;
```

```
...
mm.begin()->first = 20;                    // 错误！Multimap 键也不能改变
```

- 如果你改变 set 或 multiset 里的元素， 你必须确保不改变一个键部分（影响容器有序性的元素部分）。如果你做了，你会破坏容器，再使用那个容器将产生未定义的结果， 而且那是你的错误。
- 安全的方法：复制副本，修改副本，删除老元素，增加修改的副本

条款 23：考虑用有序 vector 代替关联容器

- 标准关联容器是基于红黑树的，红黑树是一个对插入、删除和查找的混合操作优化的数据结构。
- 在很多应用中，使用数据结构并没有那么混乱。通常分为 3 个阶段：1、建立 2、查找 3、重组，没有频繁的插入和删除。使用二分法查找作用一个有序 vector 可能比关联容器的查找更快，而且消耗更少的内存，当页面错误值得重视的时候，是一个优化。

条款 24：当关乎效率时应该在 map::operator[]和 map-insert 之间仔细选择

- 如果你要更新已存在的 map 元素，operator[]更好，但如果你要增加一个新元素，insert 则有优势。

条款 27：用 distance 和 advance 把 const_iterator 转化成 iterator

```
typedef deque<int> IntDeque;               // 和以前一样
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;
IntDeque  d;
ConstIter ci;
...                                        // 让 ci 指向 d
Iter i(d.begin());                                 // 初始化 i 为 d.begin()
advance(i, distance<ConstIter>(i, ci));            // 把 i 移到指向 ci 位置
```

条款 28：了解如何通过 reverse_iterator 的 base 得到 iterator

```
vector<int>::reverse_iterator ri =     // 使 ri 指向 3
       find(v.rbegin(), v.rend(), 3);
vector<int>::iterator i(ri.base());    // 使 i 和 ri 的 base 一样
```

条款 29：需要一个一个字符输入时考虑使用 istreambuf_iterator

```
ifstream inputFile("Data.txt");
inputFile.unset(ios::skipws);  // 关闭忽略空格标志
string fileData((istream_iterator<char>(inputFile)),
istream_iterator<char>());
```

条款 30：确保目标区间足够大

- back_inserter、front_inserter 或 inserter 插入器可以自动分配内存
- 用 reserve 来改善性能，也需要使用插入器，因为 reserve 只分配内存，没有创建对象

```
vector<int> values;
vector<int> results;
results.reserve(results.size() + values.size());
transform(values.begin(), values.end(),        // 把 transmogrify 的结
                back_inserter(results),          // 写入 results 的结尾，
                transmogrify);                   // 处理时避免了重新分配
```

条款 31：了解你的排序选择

- 算法 sort、stable_sort、partial_sort 和 nth_element 需要随机访问迭代器，所以它们可能只能用于 vector、string、deque 和数组。对标准关联容器排序元素没有意义，因为这样的容器使用它们的比较函数来在任何时候保持有序。List 通过提供 sort 成员函数。
- partial_sort 排序了前面的元素，而 nth_element 没有排序

```
partial_sort(widgets.begin(), widgets.begin() + 20,
widgets.end(), qualityCompare);
nth_element(widgets.begin(), widgets.begin() + 20,
widgets.end(), qualityCompare);
```

- partition 和 stable_partition 只需要双向迭代器。因此你可以在任何标准序列迭代器上使用 partition 和 stable_partition。
- 需要资源排序(少->多)：1. partition 2. stable_partition 3. nth_element 4. partial_sort 5. sort 6. stable_sort

条款 32：如果你真的想删除东西的话就在类似 remove 的算法后接上 erase

- remove 需要前向迭代器和可以通过这些迭代器赋值的能力。所以，它不能应用于由输入迭代器划分的区间，也不能是 map 或 multimap，也不能是 set 和 multiset 的一些实现。

- remove 并不真正删除元素，而是把不需要删除的元素移到前面，返回一个迭代器，指向最后一个不删除元素的下一个（即删除元素的开头）。
- list 的 remove 是真正删除元素的。Void remove(const _Tp& __value);
- unique 行为也像 remove。它用来从一个区间删除东西（邻近的重复值）而不用访问持有区间元素的容器。结果，如果你真的要从容器中删除元素，你也必须成对调用 unique 和 erase，例外：unique 在 list 中也类似于 remove。

条款 33：提防在指针的容器上使用类似 remove 的算法

- 如果容器是指针类型，erase(remove())之前先删除指针所指向的对象，否则内存泄漏。

  解决方法 1：

```
void delAndNullifyUncertified(Widget*& pWidget) {
       if (!pWidget->isCertified()) {
              delete pWidget;
              pWidget = 0;
       }
}
for_each(v.begin(), v.end(), delAndNullifyUncertified); //在移出容器前先删除资源
v.erase(remove(v.begin(), v.end(), static_cast<Widget*>(0)), v.end());
```

  解决方法 2：使用智能指针（shared_ptr）

条款 34：注意哪个算法需要有序区间

- 需要有序区间的算法

  binary_search
  lower_bound
  upper_bound
  equal_range
  set_union
  set_intersection
  set_difference
  set_symmetric_difference
  merge
  inplace_merge
  includes
- 一般用于有序区间，虽然它们不要求

  unique（删除连续重复的值，如果需要删除全部重复的值，先排序）
  unique_copy
- 需要保证用于算法的比较函数和用于排序的比较函数一致

条款35：通过mismatch或lexicographical_compare实现简单的忽略大小写字符串比较

- mismatch

```
int ciStringCompareImpl(const string& s1,
                    const string& s2);
int ciStringCompare(const string& s1, const string& s2){
       if (s1.size() <= s2.size()) return
ciStringCompareImpl(s1, s2);
       else return -ciStringCompareImpl(s2, s1);
}
int ciStringCompareImpl(const string& si, const strings s2){
       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 ciCharCompare(*p.first, *p.second);
}
```

- lexicographical_compare

```
bool ciCharLess(char c1, char c2){

       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);
}
```

条款36：了解copy_if的正确实现

- copy的诸多形式中，没有copy_if

```
copy                          copy_backward
replace_copy                  reverse_copy
replace_copy_if               unique_copy
remove_copy                   rotate_copy
remove_copy_if                partial_sort_copy
unintialized_copy
```

- copy_if 的有 Bug 的实现

```
template<typename InputIterator,
                    typename OutputIterator,
                    typename Predicate>
OutputIterator copy_if(InputIterator begin,
                           InputIterator end,
                           OutputIterator destBegin, Predicate
p){
       return remove_copy_if(begin, end, destBegin, not1(p));
//不能应用于函数指针
}
```

- 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;
}
```

条款 37：用 accumulate 或 for_each 来统计区间

- accumulate 存在两种形式。带有一对迭代器和初始值的形式可以返回初始值加由迭代器划分出的区间中值的和

```
list<double> ld;
double sum = accumulate(ld.begin(), Id.end(), 0.0); //注意：初始值 0.0 类型和声明类型一致，如果用 0，最后结果为整形
```

- 另一种形式，带有一个初始和值与一个任意的统计函数

```
//字符串长度和
string::size_type stringLengthSum(string::size_type sumSoFar,
const string& s){
       return sumSoFar + s.size();
}
set<string> ss;
string::size_type lengthSum = accumulate(ss.begin(), ss.end(),
0, stringLengthSum);
//乘积和
vector<float> vf;
float product = accumulate(vf.begin(), vf.end(), 1.0f,
multiplies<float>());
//利用 for_each，返回值为函数对象，可以利用返回的函数对象取运算结果
struct Point {...};
class PointAverage:
```

```
                public unary_function<Point, void> {
public:
        PointAverage(): xSum(0), ySum(0), numPoints(0) {}
        void operator()(const Point& p){
                ++numPoints;
                xSum += p.x;
                ySum += p.y;
        }
        Point result() const{
                return Point(xSum/numPoints, ySum/numPoints);
        }
private:
        size_t numPoints;
        double xSum;
        double ySum;
};
list<Point> lp;
Point avg = for_each(lp.begin(), lp.end(),
PointAverage()).result;
```

条款 38：把仿函数类设计为用于值传递

- 函数对象以值传递和返回，你的任务就是确保当那么传递（也就是拷贝）时你的函数对象行为良好。这暗示了两个东西。

  1．第一，你的函数对象应该很小。否则它们的拷贝会很昂贵。

  2．第二，你的函数对象必须单态（也就是，非多态）——它们不能用虚函数。

- 支持多态的解决办法（Bridge 模式）：

```
template<typename T>
class BPFCImpl : public unary_function<T, void> {
private:
        Widget w;//拷贝昂贵的对象
        ...
        virtual ~BPFCImpl();
        virtual void operator()(const T& val) const;//虚，支持多
态
        friend class BPFC<T>;
};
template<typename T>
class BPFC : public unary_function<T, void> {
private:
        shared_ptr<BPFCImpl<T> > pImpl;//这里应该用 shared_ptr，否
则需要在拷贝构造函数里进行处理
public:
        explicit BPFC(BPFCImpl<T> *imp) : pImpl(imp){}
        void operator()(const T& val) const {//非虚，const：修改
拷贝后的函数对象无意义
                pImpl->operator() (val);
        }
        ...
```

```
};
//用例
vector <string> c;
for_each(c.begin(), c.end(), BPFC<string>(new
BPFCImpl<string>()));
```

条款 39：用纯函数做判断式

- 判断式是返回 bool 的东西（或者其他可以隐式转化为 bool 的东西）
- 纯函数是返回值只依赖于参数的函数。
- 判断式类中的 operator()函数应该是纯函数（因为函数对象以值传递和返回，修改拷贝后的对象无意义，利用引用计数 shared_ptr 实现仿函数类除外）

条款 40：使仿函数类可适配

- 所有东西都是 public 的仿函数类的作者经常把它们声明为 struct 而不是 class，也许只因为可以避免在基类和 operator()函数前面输入“public”。
- 如果我们没有把仿函数类继承自 unary_function 或 binary_function，将不能编译，因为 not1 和 bind2nd 都只和可适配的函数对象合作。
- STL 函数对象模仿了 C++函数，而一个 C++函数只有一套参数类型和一个返回类型。结果，STL 暗中假设每个仿函数类只有一个 operator()函数，而且这个函数的参数和返回类型和传给 unary_function 或 binary_function 的一致。

条款 41：了解使用 ptr_fun、mem_fun 和 mem_fun_ref 的原因

- ptr_fun：在对函数指针做 not1 等运算时，需要加 ptr_fun
- mem_fun：通过指针方式调用成员函数时，需要加 mem_fun
- mem_fun_ref：通过引用或值调用成员函数时，需要加 mem_fun_ref

条款 42：确定 less<T>表示 operator<

- 如果需要 less<T>根据不同的数据进行处理，请定义不同的仿函数类

```
//根据重量比较
bool operator<(const Widget& lhs, const Widget& rhs){
       return lhs.weight() < rhs.weight();
}
//需要根据速度处理时，定义一个新的仿函数类
struct MaxSpeedCompare: public binary_function<Widget, Widget,
bool> {
       bool operator()(const Widget& lhs, const Widget& rhs)
const {
              return lhs.maxSpeed() < rhs.maxSpeed();
       }
};
```

条款 43：尽量用算法调用代替手写循环

- 效率：算法通常比程序员产生的循环更高效。

  1. 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遍历。

  2. 除了最微不足道的 STL 算法，所有的 STL 算法使用的计算机科学都比一般的 C++ 程序员能拿得出来的算法复杂——有时候会复杂得多。

- 正确性：写循环时比调用算法更容易产生错误。

  1. 写循环时，比较麻烦的事在于确保所使用的迭代器（a）有效，并且（b）指向你所期望的地方。

```
//错误
deque<double>::iterator insertLocation = d.begin();
for (size_t i = 0; i < numDoubles; ++i) {
       d.insert(insertLocation++, data[i] + 41);//插入后
insertLocation 无效
}
//正确
deque<double>::iterator insertLocation = d.begin();
for (size_t i = 0; i < numDoubles; ++i) {
       insertLocation = d.insert(insertLocation, data[i]
+ 41);//这样确保 insertLocation 有效
       ++insertLocation;
}
```

- 可维护性：算法通常使代码比相应的显式循环更干净、更直观。

条款 44：尽量用成员函数代替同名的算法

- 关联容器提供了 count、find、lower_bound、upper_bound 和 equal_range。
- 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。

  1. 首先，成员函数更快。

     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），效率比完全平衡树高。

  2. 其次，它们与容器结合得更好（尤其是关联容器）。

     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（==），而关联容器是用等价（!(x<y) && !(y<x)）来测试它们的“同一性”。

     如果 STL 算法作用于 map，用于比较的对象是 pair，而成员函数是比较 key。

条款 45：注意

count、find、binary_search、lower_bound、upper_bound 和 equal_range 的区别

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

条款 46：考虑使用函数对象代替函数作算法的参数

- 当我们试图把一个函数作为参数时，编译器默默地把函数转化为一个指向那个函数的指针。
- 大部分编译器不会内联通过函数指针调用的函数。

条款 47：避免产生只写代码

- 假设你有一个 vector<int>，你想去掉 vector 中值小于 x 而出现在至少和 y 一样大的最后一个元素之后的所有元素。一个方案的轮廓跳入脑中：

  1. 找到 vector 中一个值的最后一次出现需要用逆向迭代器调用 find 或 find_if 的某种形式。
  2. 去掉元素需要调用 erase 或 erase-remove 惯用法。

把这两个想法加在一起，你可以得到这个伪代码，其中“something”表示一个用于还没有填充的代码的占位符：

```
v.erase(remove_if(find_if(v.rbegin(), v.rend(),
something).base(),
                          v.end(),
                          something)),
                          v.end());
```

一旦你完成了这个，确定 something 就不是很难了，读者们却很难把最后的产物分解回它基于的想法。这就被称为只写代码：很容易写，但很难读和理解。

- 代码的读比写更经常，这是软件工程的真理。
- 解决办法：分解难于理解的长语句

条款 48：总是#include 适当的头文件

- 无论何时你使用了一个头文件中的任意组件，就要确定提供了相应的#include 指示，就算你的开发平台允许你不用它也能通过编译。当你发现移植到一个不同的平台时这么做可以减少压力，你的勤奋将因而得到回报。

1. 几乎所有的容器都在同名的头文件里，比如，vector 在<vector>中声明，list 在<list>中声明等。例外的是<set>和<map>。<set>声明了 set 和 multiset，<map>声明了 map 和 multimap。
2. 除了四个算法外，所有的算法都在<algorithm>中声明。例外的是 accumulate、inner_product、adjacent_difference 和 partial_sum。这些算法在<numeric>中声明。
3. 特殊的迭代器，包括 istream_iterator 和 istreambuf_iterator，在<iterator>中声明。
4. 标准仿函数（比如 less<T>）和仿函数适配器（比如 not1、bind2nd）在<functional>中声明。