C++编程(14)

唐晓晟
北京邮电大学电信工程学院

第16章 库组织和容器

- 标准库的设计
- 容器设计
- 向量
- 忠告

16.1 标准库的设计

- C++标准库
- 提供了对一些语言特征的支持，例如，存储管理
- 提供了有关实现所确定的语言方面的一些信息，例如最大的float值
- 提供了那些无法在每个系统上由语言本身做出最优实现的函数，例如sqrt()
- 提供了一些非基本的功能，使程序员可以为可移植性而依靠它们，例如表，映射
- 提供了一个为扩展它所提供功能的基本框架
- 为其它库提供一个公用的基础

16.1.1 设计约束

- 标准库所扮演的角色为它的设计提出了许多约束
- 对于每个学生、每个专业程序员，包括其他库的构建者，都应该是及其宝贵而且又是能负担得起的
- 被每个程序员，在与库相关的领域中的每项工作中直接或者间接地使用
- 足够高效，能够在其他库的实现中成为手工编写的函数、类或模板的真正替代品

设计约束

- 在数学的意义上是最基础性的
- 对普通使用是方便而高效的、并具有合理的安全性在它们所做的工作方面是完全的
- 能很好地与内部类型和操作混合使用
- 按照默认方式是类型安全的
- 支持各种被广泛接受的程序设计风格
- 可以扩展，使之能以类似于内部类型和标准库类型的处理方式，去处理用户定义类型

16.1.2 标准库组织

- 标准库的功能都定义在std名字空间里，用一组头文件的方式呈现，这些头文件表明了这个库的各个主要部分，因此，列出这些头文件也就给出了标准库的一个概貌
- 所有名字以c开头的标准头文件等价于C标准库中的一个头文件，每个头文件<X.h>在全局名字空间和std名字空间里定义了C标准库的一个部分，与之对应的存在一个头文件<cX>，它只在名字空间std里定义同样的一组名字

标准库组织

Containers		
<vector>	*one-dimensional array of T*	§16.3
<list>	*doubly-linked list of T*	§17.2.2
<deque>	*double-ended queue of T*	§17.2.3
<queue>	*queue of T*	§17.3.2
<stack>	*stack of T*	§17.3.1
<map>	*associative array of T*	§17.4.1
<set>	*set of T*	§17.4.3
<bitset>	*array of booleans*	§17.5.3

- 关联容器multimap和multiset可以分别在<map>和<set>中找到，priority_queue在<queue>里声明

标准库组织

General Utilities		
<utility>	*operators and pairs*	§17.1.4, §17.4.1.2
<functional>	*function objects*	§18.4
<memory>	*allocators for containers*	§19.4.4
<ctime>	*C-style date and time*	§s.20.5

- 头文件<memory>还包括auto_ptr模板，主要用于指针与异常间的平滑交互

Iterators		
<iterator>	*iterators and iterator support*	Chapter 19

- 迭代器提供一些机制，使标准算法能通用于标准容器和类似类型

标准库组织

Algorithms		
<algorithm>	*general algorithms*	Chapter 18
<cstdlib>	*bsearch ( ) qsort ( )*	§18.11

- 一个典型的通用算法可以应用于任意的具有任何类型的元素的序列，C标准库函数bsearch()和qsort()可以应用于以任何类型为元素的内部数组，条件是该元素类型没有用户定义的复制构造函数和析构函数

Diagnostics		
<exception>	*exception class*	§14.10
<stdexcept>	*standard exceptions*	§14.10
<cassert>	*assert macro*	§24.3.7.2
<cerrno>	*C-style error handling*	§20.4.1

标准库组织

Strings		
<string>	*string of T*	Chapter 20
<cctype>	*character classification*	§20.4.2
<cwtype>	*wide-character classification*	§20.4.2
<cstring>	*C-style string functions*	§20.4.1
<cwchar>	*C-style wide-character string functions*	§20.4
<cstdlib>	*C-style string functions*	§20.4.1

- 头文件<cstring>里声明了strlen()、strcpy()等一族函数。<cstdlib>里声明了atof()和atoi()，它们用于将C风格的字符串转换到数值

标准库组织

Input/Output		
<iosfwd>	*forward declarations of I/O facilities*	§21.1
<iostream>	*standard iostream objects and operations*	§21.2.1
<ios>	*iostream bases*	§21.2.1
<streambuf>	*stream buffers*	§21.6
<istream>	*input stream template*	§21.3.1
<ostream>	*output stream template*	§21.2.1
<iomanip>	*manipulators*	§21.4.6.2
<sstream>	*streams to/from strings*	§21.5.3
<cstdlib>	*character classification functions*	§20.4.2
<fstream>	*streams to/from files*	§21.5.1
<cstdio>	***printf*** *( ) family of I/O*	§21.8
<cwchar>	***printf*** *( ) -style I/O of wide characters*	§21.8

标准库组织

Localization		
<locale>	*represent cultural differences*	§21.7
<clocale>	*represent cultural differences C-style*	§21.7

- 一个locale使一些特征本地化，如日期的输出格式，用于表示现金的字符，字符串的排列准则等，这些特征体现了不同自然语言和文化之间的差异

标准库组织

Language Support		
<limits>	*numeric limits*	§22.2
<climits>	*C-style numeric scalar-limit macros*	§22.2.1
<cfloat>	*C-style numeric floating-point limit macros*	§22.2.1
<new>	*dynamic memory management*	§16.1.3
<typeinfo>	*run-time type identification support*	§15.4.1
<exception>	*exception-handling support*	§14.10
<cstddef>	*C library language support*	§6.2.1
<cstdarg>	*variable-length function argument lists*	§7.6
<csetjmp>	*C-style stack unwinding*	§s.18.7
<cstdlib>	*program termination*	§9.4.1.1
<ctime>	*system clock*	§s.18.7
<csignal>	*C-style signal handling*	§s.18.7

- <cstddef>头文件定义了由sizeof()的返回类型size_t，指针之差的结果类型ptrdiff_t和声名狼藉的NULL宏

标准库组织

Numerics		
<complex>	*complex numbers and operations*	§22.5
<valarray>	*numeric vectors and operations*	§22.4
<numeric>	*generalized numeric operations*	§22.6
<cmath>	*standard mathematical functions*	§22.3
<cstdlib>	*C-style random numbers*	§22.7

- 由于历史的原因，abs()、fabs()和div()位于<cstdlib>里，而不是与其他数学函数一起在<cmath>里

标准库组织

- 用户或者库的实现者都不允许在标准头文件里添加或者减少任何东西。试图通过在包含头文件之前定义一些宏的方式去修改头文件的内容，或者通过在其环境中写声明去改变头文件里的声明的意义，都是不可接受的
- 为了使用标准库的功能，必须包含有关的头文件

16.2 容器设计

- 容器就是能保存其他对象的对象，例如：表、向量和关联数组
- C++标准库容器的设计依据的是两条准则：在各个容器的设计中提供尽可能大的自由度，同时，又使各种容器能够向用户呈现出一个公共的界面，这将使容器实现能达到最佳的效率，也使用户能写出不依赖于所使用的特定容器类型的代码

16.2.1 专门化的容器和迭代器

- 要提供向量和表，最明显方式就是对它们中的每个都按照预想的方式给予定义：

```
Template<class T>class Vector{
public:
    explicit Vector(size_t n);
    T& operator[](size_t);
};
Template<class T>class List{
public:
    class Link{/*...*/};
    List();
    void put(T*);
    T* get();
};
```

每个类提供一组操作，对于使用而言，这些操作基本上是最理想的，而且，我们可以为各个类选择最合适的表示方式，而不去考虑其他容器的情况

有关操作的实现基本上是最优的

专门化的容器和迭代器

- 对于大部分容器，一种公共的使用方式就是迭代式地穿过整个容器，一个接一个地查看元素，这通常可以通过定义适合于容器类型的迭代器类做到
- 最理想的情况是：一个迭代类能够应用于各种容器类

```
template<class T>class Itor { // 公共界面
public:
    virtual T* first() = 0;  // 返回第一个元素的指针
    virtual T* next() = 0; // 返回下一个元素的指针
};
```

专门化的容器和迭代器

- 现在可以为Vector和List提供迭代器

```
template<class T>class Vector_itor:public Itor<T>{
    Vector<T>& v;
    size_t index;
public:
    Vector_itor(Vector<T>& vv):v(vv), index(0){}
    T* first(){ return (v.size())? &v[index=0]:0; }
    T* next(){ return (++index<v.size())? &v[index]:0;}
};
template<class T>class List_itor:public Itor<T>{
    List<T>& lst;
    List<T>::Link p;
public: List_itor(List<T>&);
    T* first(); T* next();
};
```

专门化的容器和迭代器

- 写容器上迭代的代码

```
int count(Itor<char>& ii, char term)
{
    int c = 0;
    for(char* p = ii.first(); p; p=ii.next()) if(*p==term) c++;
    return c;
}
```

- 注意，这种方式引起了一次虚函数的调用，很多情况下，这种额外开销是无关大局的，但是，这种方式对于标准库不合适
- 本例中：Itor甚至可以在Vector和List设计和实现很久以后才提供

16.2.2 有基类的容器

- Simula采用如下风格定义了它的标准容器

```
struct Link{
    Link* pre; Link* suc;
};
class List{
    Link* head;
    Link* curr;
public:
    Link* get();
    void put(Link*);
};
// 一个List就是一个Link的
表，可以保存由Link派生的任
何类型的对象
```

```
class Ship:public Link{/*...*/};

void f(List* lst){
    while(Link* po=lst->get()){
        if(Ship* ps =
            dynamic_cast<Ship*>(po)){
                //...
        }
        else {
            //...
        }
    }
}
```

有基类的容器

- 常见情况是定义一个公共容器类型

```
class Container:public Object{
 // 公共容器类型
public:    virtual Object* get();
    virtual void put(Object*);
    virtual Object*& operator[](size_t);
}; // 操作全部是虚的
class List:public Container{
public:
    Object* get();
    void put(Object*);
};
class Vector:public Container{
public:    Object*& operator[](size_t);
};
```

对于此类实现，Container中一般提供所希望支持的各种容器类基本操作集合的并集，针对一组概念的这样一个界面被称作一个肥大界面(fat interface)

这类设计将不必要的复杂性推给了用户，强加了显著的运行时开销，并限制了可以放入容器的对象的种类，对于标准库也不理想

16.2.3 STL容器

- 标准库容器和迭代器(STL框架)可以认为是一种能够同时取得前面描述了的两种传统模型的优点的途径，STL是一心一意追求真正高效的和通用的算法而结出的硕果
- 考虑到效率因素，STL中排除了采用难以内联化的虚函数去实现短小并使用频繁的访问函数
- 为了避免肥大界面，公共操作集合中不包括那些无法有效在所有集合中有效实现的操作
- 为每种容器提供自己的迭代器，并让这些迭代器都支持同一组标准的迭代器操作

STL容器

- 标准容器不是从一个公共基类派生出来的，而是每个容器实现了完整的标准容器界面
- 也不存在公共的迭代器基类，这样，在使用标准容器或者迭代器时就不涉及任何显式或隐式的运行时类型检查
- 最重要也是最困难的问题是如何为所有容器提供公共的服务，这个问题是通过模板参数传递的“分配器”(allocator)处理的，没有通过公共基类
- STL广泛而深入的依靠模板机制，为避免大量的代码重复，通常需要通过专门化的方式，为保存指针的容器提供共享的实现

16.3 向量(vector)

- 作为标准库的一个完整例子，除非另有说明，关于vector所说的情况都适用于每种标准库容器
- 17章将描述list, set, map等的专有特征
- 对vector的介绍分几个步骤进行：成员类型、迭代器、元素访问、构造函数、堆栈操作、表操作、大小和容量、协助函数、以及vector<bool>

16.3.1 类型

```
template <class T, class A = allocator<T> > class std: :vector {
public:    // types:
    typedef T value_type; // 元素类型
    typedef A allocator_type; // 存储管理器类型
    typedef typename A::size_type size_type; // 一般做容器下标size_t
    typedef typename A::difference_type difference_type; // ptrdiff_t
    typedef implementation_dependent1 iterator; // T* (迭代器)
    typedef implementation_dependent2const_ iterator; // const T*
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> // 反向迭代器
                                        const_reverse_iterator;
    typedef typename A::pointer pointer; // 元素指针
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference; // 元素引用
    typedef typename A::const_reference const_reference;
};
```

类型

- 使用容器的代码
- 必须在作为模板参数的成员类型名字之前加typename，不加的话，编译器无法判断

```
template<class C> typename C::value_type sum(const C& c)
{
    typename C::value_type s = 0;
    typename C::const_iterator p = c.begin() ; // 从头开始
    while (p!=c.end()) { // 继续到结束处
        s += *p; // 取得元素的数值
        ++p; // 使p指向下一个元素
    }
    return s;
}
```

16.3.2 迭代器

```
template <class T, class A = allocator<T> > class vector {
public:
    iterator begin() ; // 指向首元素
    const_iterator begin() const;
    iterator end() ; // 指向末端元素的下一个位置
    const_iterator end() const;
    reverse_iterator rbegin() ; // 指向反向序列的首元素
    const_reverse_iterator rbegin() const;
    reverse_iterator rend() ; // 同上(end())
    element of reverse sequence
    const_reverse_iterator rend() const;
    // ...
};
```

迭代器

```
template<class C> typename C::iterator
    find_last(const C& c, typename C::value_type v)
{ // 对vector进行反向搜索，找到第一个出现v的位置
    typename C::reverse_iterator  ri = find(c.rbegin(), c.rend(), v);
    if(ri == c.rend()) return c.end(); // 表示未找到
    typename C::iterator i = ri.base();
    return --i;
}
```

- 对于reverse_iterator，ri.base()返回一个iterator，指向ri所指之处后面的一个位置，若没有reverse_iterator，必须使用iterator写一个显式的循环来完成同样的任务
- 注意：C::reverse_iterator和C::iterator不是同一个类型

16.3.3 元素访问

- 与其他容器相比，vector的一个重要特点就是可以方便高效地按照任何顺序访问其中的元素

```
template <class T, class A = allocator<T> > class vector {
public:
    reference operator[](size_type n) ; // 不加检查的访问
    const_reference operator[](size_type n) const;
    reference at(size_type n) ; // 受检查的访问
    const_reference at(size_type n) const;
    reference front() ; // 首元素
    const_reference front() const;
    reference back() ; // 尾元素
    const_reference back() const;
    // ...
};
```

元素访问示例

- 下标索引通过operator[]()和at()完成，at()进行下标检查，越界时抛出out_of_range异常
- 访问操作返回类型为const_reference或者reference，对于vector<X>，reference就是X&

```
void f(vector<int>& v, int i1, int i2)
try {
    for(int i = 0; i < v.size() ; i++) {
        // 已经做范围检查，这里用不检查的v[i]
    }
    v.at(i1) = v.at(i2) ; // 检查下标
}
catch(out_of_range) {
    // 错误: out_of_range
}
```

元素访问示例

- 在各种标准序列中，只有vector和deque支持下标
- 试图创建一个越界引用的效果是无定义的

```
void f(vector<double>& v)
{
    double d = v[v.size()] ; // 无定义：下标错误
    list<char> lst;
    char c = lst.front() ; // 无定义：表为空
}
```

16.3.4 构造函数

```
template <class T, class A = allocator<T> > class vector {
public:
    explicit vector(const A& =A()) ;
    explicit vector(size_type n, const T& val = T() , const A& =A());
        // n个val的副本
    template <class In> // In 必须是输入迭代器 (§19.2.1)
    vector(In first, In last, const A& =A()) ; // 从first到last进行拷贝
    vector(const vector& x) ;
    ~vector() ;
    vector& operator=(const vector& x) ;
    template <class In> // In 必须是输入迭代器(§19.2.1)
    void assign(In first, In last) ; //从first到last进行拷贝
    void assign(size_type n, const T& val) ; // n个val的副本
    // ...
};
// vector提供了完整的一组构造、析构和复制操作
```

构造函数示例

- vector提供了对任意元素的快速访问，但是修改它的值的代价是相当昂贵的，因此，我们在创建vector时通常给出一个初始大小

```
vector<Record> vr(10000) ;
void f(int s1, int s2) {
    vector<int> vi(s1) ;
    vector<double>* p = new vector<double>(s2) ;
}
// 这样分配的vector元素都将用元素类型的默认构造函数做初始化，
如果没有默认构造函数，就不能创建以这个类型为元素的向量
class Num{
public:   Num(long) ; // 无默认构造函数
};
vector<Num> v1(1000) ; // 错误: 无默认构造函数
vector<Num> v2(1000,Num(0)) ; // ok
```

构造函数示例

- vector的大小也可以通过初始元素的集合隐式给定，这样做需要给vector提供一个值的序列，此时，vector的构造函数会调整vector的大小

```
void f(const list<X>& lst)
{
    vector<X> v1(lst.begin() ,lst.end()) ; // 从表复制元素
    char p[] = "despair";
    vector<char> v2(p,&p[sizeof(p)-1]); // 从c风格字符串复制字符
}
```

构造函数示例

- 复制构造函数和复制赋值运算符拷贝vector的所有元素，其代价可能会很高，所以vector通常采用引用传递

```
void f1(vector<int>&) ; // 常见风格
void f2(const vector<int>&) ; // 常见风格
void f3(vector<int>) ; // 罕见风格
void h()
{
    vector<int> v(10000) ;
    // ...
    f1(v) ; // 传递引用
    f2(v) ; // 传递引用
    f3(v) ; // 会导致10000个元素的赋值操作
}
```

构造函数示例

- assign函数，可以被用来一次性地为vector中的多个元素赋值，这样做时，没有引进大量的构造函数或者转换函数
- 注意：赋值将完全改变一个向量里的全部元素，所有的老元素都被删除，新元素插入

```
class Book {   /*...*/  };
void f(vector<Num>& vn, vector<char>& vc,
            vector<Book>& vb, list<Book>& lb)
{
    vn.assign(10,Num(0)) ; // 用10个Num(0)给vn赋值
    char s[] = "literal";
    vc.assign(s,&s[sizeof(s)1]); // 用“literal”给vc赋值
    vb.assign(lb.begin() ,lb.end()) ; // 用表的元素赋值
}
```

16.3.5 堆栈操作

- 最常见的情况是，我们把vector看作一种可以直接通过下标访问元素的紧凑数据结构
- vector的push_back将隐式地导致vector的size()增长，所以不会上溢，但是可能下溢

```
template <class T, class A = allocator<T> > class vector {
public:
    void push_back(const T& x) ; // 在最后加入
    void pop_back() ; // 删除最后元素
    // ...
};
```

堆栈操作

- 每次调用push_back时vector就增长一个元素，这个元素被加到最后，所以s[s.size()-1]就是s.back()
- vector可以做为realloc()的更通用、更优美而且丝毫不降低效率的替代机制

```
void f(vector<char>& s){
    s.push_back('a') ;
    s.push_back('b') ;
    s.push_back('c') ;
    s.pop_back() ; // 注意，并不返回数值，只是弹出
    if (s[s.size()-1]!='b') error("impossible!") ;
    s.pop_back() ;
    if (s.back() != 'a') error("should never happen!") ;
}
```

16.3.6 表操作

- push_back()、pop_back()和back()操作使vector可以有效地作为堆栈使用
- 然而，在vector的中间增加元素，从vector里删除元素有时也很有用

```
template <class T, class A = allocator<T> > class vector {
public:
    iterator insert(iterator pos, const T& x) ; // 在pos前添加x
    void insert(iterator pos, size_type n, const T& x) ;  // 添加n个x
    template <class In> void insert(iterator pos, In first, In last) ;
        // In必须是输入迭代器 // 插入一批来自序列的元素
    iterator erase(iterator pos) ; // 删除pos处的元素
    iterator erase(iterator first, iterator last) ; // 删除一段元素
    void clear() ; // 删除所有元素
}; // insert()返回的迭代器指向新插入的元素，erase()返回的迭代器指向被删除的最后元素之后的那个元素，这两种操作都可能导致迭代器指向另外的元素
```

表操作

```
vector<string> fruit;
fruit.push_back("peach");
fruit.push_back("apple");
fruit.push_back("kiwifruit");
fruit.push_back("pear");
fruit.push_back("starfruit");
fruit.push_back("grape");
sort(fruit.begin() ,fruit.end()); // 排序
vector<string>::iterator p1 =
    find_if(fruit.begin() ,fruit.end() ,initial('p');
    // 找到第一个以p开头的水果
vector<string>::iterator p2 =
    find_if(p1,fruit.end() ,initial_not('p');
    // 找到最后一个以p开头的水果
fruit.erase(p1,p2) ;
// 删除p1到p2之间的所有元素，但是不包括p2
```

注意，erase()要求两个类型一样的参数，而iterator和reverse_iterator是不同的类型，不可在这里混用。

删除元素将改变vector的大小，位于被删除元素之后的元素将被复制到空位中。

可以使用insert()插入元素，但是注意这将导致批量元素被移动，若插入元素太多，，请考虑使用list容器

16.3.7 元素定位

- 一般来说，insert或者erase函数的目标是某个明确的位置(begin()或end())，或者通过检索操作得到的结果(find())，或者在迭代中确定的某个下标
- 实际应用中，也可以直接使用数值下标(比如数字7)作为索引，方法是c.erase(c.begin()+7)

元素定位示例

```
template<class C> void f(C& c){
    c.erase(c.begin()+7); // ok
    // 但并非所有容器的迭代器都支持+运算，list不支持c.begin()+7
    c.erase(&c[7]); // 一般不行
    // 此处，c[7]引用相应的元素，其地址是合法的迭代器
    // 但是，对于其他容器，比如map，将返回一个mapped_type&
    // 而不是一个到元素的引用(其类型应该是value_type&)
    c.erase(c+7); // 错误：容器+7没有意义
    c.erase(c.back()); // 错误：c.back()是引用，不是迭代器
    c.erase(c.end()-2); // 可以(倒数第二个元素)
    c.erase(c.rbegin()+2);
        // 错误：vector::reverse_iterator和vector::iterator的类型不同
    c.erase((c.rbegin()+2).base()) ; // 难懂，但可行($19.2.5)
}
```

16.3.8 大小和容量

- 通常，vector在需要时将自动增长，然而，我们也可以直接提出vector应当如何使用存储的问题
- 任意时刻，vector中的元素个数可以通过size()获得，并可通过resize()改变(和C中的realloc()很类似)

```
template <class T, class A = allocator<T> > class vector {
public:
    size_type size() const; // 元素个数
    bool empty() const { return size()==0; } // 是否为空
    size_type max_size() const; // 最大可能大小
    void resize(size_type sz, T val = T());
        // 增加的元素用val初始化
    size_type capacity() const; // 当前已经分配的存储的大小
    void reserve(size_type n); // 做出n个元素空位，不初始化
        // 如果n>max_size()，则抛出length_error
};
```

大小和容量

- 对reserve(n)的调用保证，在v的大小增加到使v.size()超过n之前不需要再做任何存储分配

```
struct Link {
    Link* next;
    Link(Link* n =0) : next(n) {}
    // ...
};
vector<Link> v;
void chain(size_t n) // 填入n个Link，每个Link指向前一个
{
    v.reserve(n) ;
    v.push_back(Link(0)) ;
    for (int i = 1; i<n; i++) v.push_back(Link(&v[i1]));
    // ...
}
```

大小和容量

- 为后面工作预留空间有两个优点
  - 即使是没有多加思考的实现也可以先分配足够的空间，以避免随着工作进展慢慢地请求足够的内存，这种情况下可能会使得效率有所提高
  - 只有通过调用reserve()，保证向量构建过程中不再出现分配，才能保证v的元素间能建立正确的链接
- 此外，还可以确保潜在的存储耗尽问题和那些代价高昂的元素重新分配只可在预期的时刻出现，对于那些有着严格的运行时间约束的程序，这些极为重要

大小和容量

- reserve()并不改变vector的大小，就是说，并没有要求它对任何新元素做初始化
- capacity()给出当时所保留的所有存储位置的个数，c.capacity()-c.size()就是在不致于重新分配的情况下，可以插入的元素数
- 减小vector的大小不会减少其容量，这样做只能为vector随后的增长多留下了一些空间

16.3.9 其他成员函数

- 许多算法涉及到元素的交换，通常交换元素就是简单的复制所有元素
- 然而，vector常常是用一种类似于到元素组的句柄的结构实现的，这样，交换两个vector的工作可以通过交换它们的句柄的方式更有效的实现

```
template <class T, class A = allocator<T> > class vector {
    public:
    // ...
    void swap(vector&) ;
    allocator_type get_allocator() const;
};
```

16.3.10 协助函数

- 用==和<可以比较两个vector

```
template <class T, class A>
bool std: :operator==(const vector<T,A>& x, const vector<T,A>& y);
template <class T, class A>
bool std: :operator<(const vector<T,A>& x, const vector<T,A>& y);
```

- 标准库还提供了!=、<=、>和>=等协助函数

16.3.11 vector<bool>

- 标准库还提供了专门化vector(bool)，作为bool类型的紧凑vector，bool变量可以寻址，所以它至少要占一个字节
- 通常的vector操作都可以对vector(bool)使用，而且保持了它们原来的意义
- 标准库还提供了布尔值集合bitset，带有一组布尔集合运算

16.4 忠告

- [1] 利用标准库功能，以维持可移植性
- [2] 决不要另行定义标准库的功能
- [3] 决不要认为标准库比什么都好
- [4] 在定义一种新功能时，应考虑它是否能够纳入标准库所提供的框架中
- [5] 记住标准库功能都定义在名字空间***std***
- [6] 通过包含标准库头文件声明其功能，不要自己另行显示声明

忠告

- [7] 利用后续抽象的优点
- [8] 避免肥大的界面
- [9] 与自己写按照反向顺序的显式循环相比，最好是写利用反向迭代器的算法
- [10] 用***base***()从***reverse_iterator***抽取出***iterator***
- [11] 通过引用传递容器
- [12] 用迭代器类型，如***list***<***char***>::***iterator***，而不要采用索引容器元素的指针

忠告

- [13] 在不需要修改容器元素时，使用***const***迭代器
- [14] 如果希望检查访问范围，请(直接或间接)使用***at***()
- [15] 多用容器和***push_back***()或***resize***()，少用数组和***realloc***()
- [16] ***vector***改变大小之后，不要使用指向其中的迭代器
- [17] 利用***reserve***() 避免使迭代器非法
- [18] 在需要的时候，***reserve***() 可以使执行情况更容易预期