STL 容器操作综述

成员类型（16.3.1节）	
value_type	元素的类型
allocator_type	存储管理器的类型
size_type	下标、元素计数等的类型
difference_type	迭代器之差的类型
iterator	行为像是*value_type**
const_iterator	行为像是*const value_type**
reverse_iterator	按反向顺序查看容器，像*value_type**
const_ reverse_iterator	按反向顺序查看容器，像*const value_type**
reference	行为像是*value_type*&
const_reference	行为像是*const value_type*&
key_type	关键码的类型（仅限于关联容器）
mapped_type	*mapped_value*的类型（仅限于关联容器）
key_compare	比较准则的类型（仅限于关联容器）

迭代器（16.3.2节）	
begin( )	指向第一个元素
end( )	指向过末端一个位置
rbegin( )	指向按反向顺序的第一个元素
rend( )	指向按反向顺序过末端一个位置

元素访问（16.3.3节）	
front( )	第一个元素
back( )	最后元素
[ ]	下标，不检查（表没有本操作）
at( )	下标，带检查访问（仅对向量和双端队列有）

堆栈和队列操作（16.3.5节、17.2.2.2节）	
push_back( )	在最后加入
pop_back( )	删除最后元素
push_front( )	加入新的首元素（仅对双端队列和表）
pop_front( )	删除首元素（仅对双端队列和表）

表操作（16.3.6节）	
insert(p, x)	在*p*前插入*x*
insert(p, n, x)	在*p*前插入*n*个*x*
insert(p, first, last)	在*p*前插入 [*first* : *last*[的元素

表操作（16.3.6节）（续）	
erase(p)	删除在*p*的元素
erase(first, last)	删除 [*first* : *last*[
clear( )	删除所有元素

其他操作（16.3.8节、16.3.9节、16.3.10节）	
size( )	元素个数
empty( )	容器为空吗?
max_size( )	可能的最大容器的规模
capacity( )	为*vector*分配的空间（仅对向量）
reserve( )	为后面扩充预留空间（仅对向量）
resize( )	改变容器的大小（仅对向量、表和双端对列）
swap( )	交换两个容器的所有元素
get_allocator( )	取得容器的分配器的一个副本
==	两个容器的元素完全相同吗?
!=	两个容器的元素不同吗?
<	一个容器按字典序在另一个之前吗?

构造函数等（16.3.4节）	
container( )	空容器
container(*n*)	*n*个默认值元素的容器（关联容器没有）
container(*n*, *x*)	*n*个*x*的拷贝（关联容器没有）
container(*first*, *last*)	用 [*first* : *last*[初始化元素
container(*x*)	复制构造函数，用容器*x*初始化元素
~*container*( )	销毁容器及其所有元素

赋值（16.3.4节）	
operator =(*x*)	复制赋值，元素来自容器*x*
assign(*n*, *x*)	赋值*n*个*x*的拷贝（关联容器没有）
assign(*first*, *last*)	用 [*first* : *last*[赋值

关联操作（17.4.1节）	
operator[ ](*k*)	访问具有关键码*k*的元素（对惟一关键码的容器）
find(*k*)	查找具有关键码*k*的元素
lower_bound(*k*)	查找具有关键码*k*的第一个元素
upper_bound(*k*)	查找关键码大于*k*的第一个元素
equal_range(*k*)	查找具有关键码*k*的元素的*lower_bound*和*upper_bound*
key_comp( )	关键码比较对象的副本
value_comp( )	*mapped_value*比较对象的副本

STL 对元素的要求

1. 容器里的元素总是被插入对象的副本，可以利用 shared_ptr 来避免拷贝操作
2. 在需要 cmp 的情况下，要求实现 operator <

- cmp 准则：

[1] *cmp*(*x*, *x*) 是*false*。

[2] 如果*cmp*(*x*, *y*) 且*cmp*(*y*, *z*)，那么*cmp*(*x*, *z*)。

[3] 定义*equiv*(*x*, *y*) 为 !(*cmp*(*x*, *y*) || *cmp*(*y*, *x*))。如果*equiv* (*x*, *y*) 且*equiv*(*y*, *z*)，那么*equiv*(*x*, *z*)。

- cmp 的两种形式：

```
template<class Ran> void sort(Ran first, Ran last);          // 用 < 做比较
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp);    // 用cmp
```

3.

在实践中也经常用小于关系（默认为 <）定义一个等价关系而不是相等（默认为 ==）。

有了 < 和 == 之后，我们很容易构造出其他常用比较。标准库在名字空间*std*::*rel_ops*里定义了它们，并通过<*utility*>给出：

```
template<class T> bool rel_ops::operator!=(const T& x, const T& y) { return !(x==y); }
template<class T> bool rel_ops::operator>(const T& x, const T& y) { return y<x; }
template<class T> bool rel_ops::operator<=(const T& x, const T& y) { return !(y<x); }
template<class T> bool rel_ops::operator>=(const T& x, const T& y) { return !(x<y); }
```

```
void f()
{
    using namespace std;
    // 按默认不产生 !=、> 等
}

void g()
{
    using namespace std;
    using namespace std::rel_ops;
    // 默认地产生 !=、> 等
}
```

没有把 != 等运算符直接定义在名字空间*std*里，因为并不总需要它们，而且有时它们的定义还会与用户代码相互干扰。例如，假定我写了一个通用数学库，我可能就希望用自己的关系运算，而不是标准库的版本。

LIST 的额外操作

- list 是一个适合需要频繁插入、删除的序列

```
template <class T, class A = allocator<T> > class list {
public:
    // ...
    // 表的特殊操作

    void splice(iterator pos, list& x);              //        将x的所有元素移到
                                                     // 本表的pos之前，且不做复制
    void splice(iterator pos, list& x, iterator p);  //              将x中的 *p移到
                                                     // 本表的pos之前，且不做复制
    void splice(iterator pos, list& x, iterator first, iterator last);

    void merge(list&);          // 归并排序的表
    template <class Cmp> void merge(list&, Cmp);

    void sort();
    template <class Cmp> void sort(Cmp);

    // ...
};
```

1. Splice 操作

```
fruit:
    apple   pear

citrus:
    orange   grapefruit   lemon
```

我们可以像

```
list<string>::iterator p = find_if(fruit.begin(),fruit.end(),initial('p'));
fruit.splice(p,citrus,citrus.begin());
```

这样将*orange*从*citrus*粘接入*fruit*。这样做的效果是从*citrus*(*citrus.begin*( ))删去了第一个元素，并将它放到*fruit*里的第一个名字以*p*开始的元素之前，结果是

```
fruit:
    apple   orange   pear

citrus:
    grapefruit   lemon
```

2. merge 操作

```
f1:
    apple   quince   pear
f2:
    lemon   grapefruit   orange   lime
```

可以按

```
f1.sort();
f2.sort();
f1.merge(f2);
```

这样的方式排序和归并，结果是

```
f1:
    apple   grapefruit   lemon   lime   orange   pear   quince
f2:
    <empty>
```

3. 其他操作

```
template <class T, class A = allocator<T> > class list {
public:
    // ...

    void remove(const T& val);
    template <class Pred> void remove_if(Pred p);

    void unique();                                  // 根据 == 删除重复元素
    template <class BinPred> void unique(BinPred b);  // 根据b删除重复元素

    void reverse();                                 // 元素反转
};
```

如果希望删除某些确定的重复情况，我们可以提供一个谓词，刻画所需要删除的那种重复。例如，假定我们已经定义了一个二元谓词（18.4.2节）*initial2(x)* 来比较*string*，如果串的起始字母是*x*就成立，对不以*x*开头的*string*返回*false*。给定

```
pear  pear  apple  apple
```

通过

```
fruit.unique(initial2('p'));
```

这一调用，我们就能从*fruit*里删除以 '*p*' 开头的连续的重复串了，这将给出

```
pear  apple  apple
```

双端队列 dequeue

- 两端操作，下标操作都是高效的，中间插入删除和 vector 一样低效

stack，queue，priority_queue 都为适配器

- stack，原生容器默认为 deque

```
template <class T, class C = deque<T> > class std::stack {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit stack(const C& a = C()) : c(a) { }   可以从原生容器拷贝构造

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    value_type& top() { return c.back(); }
    const value_type& top() const { return c.back(); }

    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};
```

- queue，原生容器默认为 deque

```
template <class T, class C = deque<T> > class std::queue {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit queue(const C& a = C()) : c(a) { }

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    value_type& front() { return c.front(); }
    const value_type& front() const { return c.front(); }

    value_type& back() { return c.back(); }
    const value_type& back() const { return c.back(); }

    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};
```

- priority_queue，原生容器默认为 vector

```
template <class T, class C = vector<T>, class Cmp = less<typename C::value_type> >
class std::priority_queue {
protected:
    C c;
    Cmp cmp;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit priority_queue(const Cmp& a1 = Cmp(), const C& a2 = C())
        : c(a2), cmp(a1) { make_heap(c.begin(),c.end(),cmp); }   // 见18.8节

    template <class In>
    priority_queue(In first, In last, const Cmp& = Cmp(), const C& = C());

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    const value_type& top() const { return c.front(); }

    void push(const value_type&);
    void pop();
};
```

关联容器

- map（map 要求关键类型提供 < 操作，以保持元素有序性，如果不需要有序特性，可以使用 hash_map）

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > >
class std::map {
public:
    // 类型:
    typedef Key key_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;
    typedef Cmp key_compare;

    typedef A allocator_type;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef implementation_defined1 iterator;
    typedef implementation_defined2 const_iterator;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    // ...
};
```

- map 的迭代器函数

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > > class map {
public:
    // ...
    // 迭代器:

    iterator begin();
    const_iterator begin() const;

    iterator end();
    const_iterator end() const;

    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;

    reverse_iterator rend();
    const_reverse_iterator rend() const;

    // ...
};
```

- pair（最后的复制构造函数提供了隐式转换）

```
template <class T1, class T2> struct std::pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair() :first(T1()), second(T2()) { }
    pair(const T1& x, const T2& y) :first(x), second(y) { }
    template<class U, class V>
        pair(const pair<U, V>& p) :first(p.first), second(p.second) { }
};
```

- make_pair

pair的用途也不限于map的实现，它本身也是一个标准库类。pair的定义可以在 <utility> 里找到。这里还提供了一个用以方便地创建pair的函数：

```
template <class T1, class T2> pair<T1, T2> std::make_pair (T1& t1, T2& t2)
{
    return pair<T1, T2>(t1, t2);
}
```

- map 下标

下标运算符将关键码作为下标去执行查找，并返回对应的值。如果不存在这个关键码，它就将一个具有该关键码和*mapped_type*类型默认值的元素插入这个*map*。例如，

```
void f()
{
    map<string, int> m;   // map开始时为空
    int x = m["Henry"]; // 为Henry建立新项，初始化为0，返回0
    m["Harry"] = 7;     // 为Harry建立新项，初始化为0并赋值7
    int y = m["Henry"]; // 返回Henry对应项的值
    m["Harry"] = 9;     // 将Harry对应项的值修改为9
}
```

- map 操作

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > >
class map {
public:
    // ...
    // 表操作:

    pair<iterator, bool> insert(const value_type& val);   // 插入(关键码,值)对
    iterator insert(iterator pos, const value_type& val); // pos只是个提示
    template <class In> void insert(In first, In last);   // 从序列中插入

    void erase(iterator pos);                  // 删除被指元素
    size_type erase(const key_type& k);        // 删除关键码为k的元素
    void erase(iterator first, iterator last); // 删除一个区间
    void clear();                              // 删除所有元素

    // ...
};
```

- map::insert 的返回值为 pair<map::iterator, bool>类型

```
void f(map<string, int>& m)
{
    pair<string, int> p99("Paul", 99);

    pair<map<string, int>::iterator, bool> p = m.insert(p99);
    if (p.second) {
        // "Paul"被插入
    }
    else {
        // 已有"Paul"
    }
    map<string, int>::iterator i = p.first;        // 指向m["Paul"]
    // ...
}
```

m[k]等效于 (*(m.insert(makepair(k, T())).first)).second

T()需要默认的构造函数

- map 的其他函数

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > >
class map {
public:
    // ...
    // 容量:

    size_type size() const;              // 元素个数
    size_type max_size() const;          // map的最大可能规模
    bool empty() const { return size()==0; }

    void swap(map&);
};
```

此外，map还提供了 ==、!=、<、>、<=、>= 和swap()，它们都作为非成员函数：

```
template <class Key, class T, class Cmp, class A>
bool operator==(const map<Key,T,Cmp,A>&, const map<Key,T,Cmp,A>&);

// !=、<、>、<=、>= 类似

template <class Key, class T, class Cmp, class A>
void swap(map<Key,T,Cmp,A>&, map<Key,T,Cmp,A>&);
```

multi_map

```
void print_numbers(const multimap<string, int>& phone_book)
{
    typedef multimap<string, int>::const_iterator I;
    pair<I, I> b = phone_book.equal_range("Stroustrup");
    for (I i = b.first; i != b.second; ++i) cout << i->second << '\n';
}
```

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > >
class std::multimap {
public:
    // 与map类似，除了：

    iterator insert(const value_type&);   // 返回iterator，不是pair

    // 无下标操作符[]
};
```

集合 set

```
template <class Key, class Cmp = less<Key>, class A = allocator<Key> >
class std::set {
public:
    // 与map类似，除了：

    typedef Key value_type;          // 关键码就是值
    typedef Cmp value_compare;
    // 无下标操作符[]
};
```

多重集合 multi_set

*multiset*是一种允许重复关键码的*set*：

```
template <class Key, class Cmp = less<Key>, class A = allocator<Key> >
class std::multiset {
public:
    // 与set类似，除了：
    iterator insert(const value_type&);   // 返回iterator，不是pair
};
```

equal_range( )、*lower_bound*( ) 和*upper_bound*( ) 操作（17.4.1.6节）是访问关键码重复出现的基本手段。

bitset（对于无法放入一个 long int 的标志位而设计）

```
template<size_t N> class bitset {
public:
    // ...
    // 构造函数：
    bitset();                          // N个二进制位0
    bitset(unsigned long val);         // 二进制位来自val

    template<class Ch, class Tr, class A>                  // Tr 是一个字符特征（20.2节）
    explicit bitset(const basic_string<Ch,Tr,A>& str,      // 二进制位来自串str
        typename basic_string<Ch,Tr,A>::size_type pos = 0,
        typename basic_string<Ch,Tr,A>::size_type n = basic_string<Ch,Tr,A>::npos);

    // ...
};
```

通过内部指针不可能直接对单个的位寻址（5.1节），因此，*bitset*提供了一种位引用类型。这实际上也是一种一般性的技术，用于对由于某些原因而使内部指针不能适用的对象寻址：

```
template<size_t N> class std::bitset {
public:
    class reference {              // 对单个位的引用：
        friend class bitset;
        reference();
    public:                                          // b[i] 引用第i+1个位：
        ~reference();
        reference& operator=(bool x);                // 用于b[i] = x;
        reference& operator=(const reference&);      // 用于b[i] = b[j];
        bool operator~() const;                      // 返回~b[i];
        operator bool() const;                       // 用于x = b[i];
        reference& flip();                           // b[i].flip();
    };

    // ...
};
```

*bitset*设计的关键想法之一就是为能够放进一个机器字的*bitset*提供优化的实现，其界面也反应了这一考虑。

- bitset 操作

```
template<size_t N> class std::bitset {
public:
    // ...
    // bitset操作：

    reference operator[](size_t pos);          // b[i]

    bitset& operator&=(const bitset& s);       // 与
    bitset& operator|=(const bitset& s);       // 或
    bitset& operator^=(const bitset& s);       // 异或

    bitset& operator<<=(size_t n);             // 逻辑左移（填充0）
    bitset& operator>>=(size_t n);             // 逻辑右移（填充0）

    bitset& set();                             // 将所有位都设置为1
    bitset& set(size_t pos, int val = 1);      // b[pos]=val

    bitset& reset();                           // 将所有位都设置为0
    bitset& reset(size_t pos);                 // b[pos]=0

    bitset& flip();                            // 改变每个位的值
    bitset& flip(size_t pos);                  // 改变b[pos]的值

    bitset operator~() const { return bitset<N>(*this).flip(); }    // 做出补集
    bitset operator<<(size_t n) const { return bitset<N>(*this)<<=n; } // 做出左移的集合
    bitset operator>>(size_t n) const { return bitset<N>(*this)>>=n; } // 做出右移的集合

    // ...
};
```

- 如果下标越界，抛出 out_of_range 异常
- 其他操作

*bitset*还支持许多常用操作，如*size*( )、==、I/O等：

```
template<size_t N> class bitset {
public:
    // ...

    unsigned long to_ulong() const;

    template <class Ch, class Tr, class A> basic_string<Ch,Tr,A> to_string() const;

    size_t count() const;                      // 值为1的二进制位个数
    size_t size() const { return N; }          // 位数

    bool operator==(const bitset& s) const;
    bool operator!=(const bitset& s) const;

    bool test(size_t pos) const;               // 如果b[pos]为1则true
    bool any() const;                          // 如果任何位为1则true
    bool none() const;                         // 如果没有位为1则true
};
```

```
void binary(int i)
{
    bitset<8*sizeof(int)> b = i;        // 假定8位字节（见22.2节）
    cout << b.template to_string< char,char_traits<char>,allocator<char> >() << '\n';
}
```

```
template<size_t N> bitset<N> std::operator&(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N> std::operator|(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N> std::operator^(const bitset<N>&, const bitset<N>&);

template <class charT, class Tr, size_t N>
basic_istream<charT,Tr>& std::operator>>(basic_istream<charT,Tr>&, bitset<N>&);
template <class charT, class Tr, size_t N>
basic_ostream<charT,Tr>& std::operator<<(basic_ostream<charT,Tr>&, const bitset<N>&);
```

- 包装内部数组

完全可能为常规数组做出一种外观形式，以提供标准容器那样的记法规范，而又不改变其低级特性，这种做法有时也很有用：

```
template<class T, int max> struct c_array {
    typedef T value_type;

    typedef T* iterator;
    typedef const T* const_iterator;

    typedef T& reference;
    typedef const T& const_reference;

    T v[max];
    operator T*() { return v; }

    reference operator[](size_t i) { return v[i]; }
    const_reference operator[](size_t i) const { return v[i]; }

    iterator begin() { return v; }
    const_iterator begin() const { return v; }

    iterator end() { return v+max; }
    const_iterator end() const { return v+max; }

    size_t size() const { return max; }
};
```

算法

非修改性的序列操作（18.5节）<algorithm>	
for_each( )	对序列中每个元素执行某个操作
find( )	在序列中找出某个值的第一个出现
find_if( )	在序列中找出符合某谓词的第一个元素
find_first_of( )	在一序列中找出另一个序列里的值
adjust_find( )	找出相邻的一对值
count( )	在序列中统计某个值出现的次数
count_if( )	在序列中统计与某谓词匹配的次数
mismatch( )	找出使两个序列相异的第一个元素
equal( )	如果两个序列对应元素都相同则真
search( )	找出一序列作为子序列的第一个出现位置
find_end( )	找出一序列作为子序列的最后一个出现位置
search_n( )	找出一序列作为子序列的第*n*个出现位置

- 修改性序列操作：借助于迭代器的操作都不能修改容器的大小，修改操作产生的输出，是输入序列修改后的副本

修改性的序列操作（18.6节）<algorithm>	
transform( )	将操作应用于序列中的每个元素
copy( )	从序列的第一个元素起进行复制
copy_backward( )	从序列的最后元素起进行复制
swap( )	交换两个元素
iter_swap( )	交换由迭代器所指的两个元素
swap_ranges( )	交换两个序列中的元素
replace( )	用一个给定值替换一些元素
replace_if( )	替换满足谓词的一些元素
replace_copy( )	复制序列时用一个给定值替换元素
replace_copy_if( )	复制序列时替换满足谓词的元素
fill( )	用一个给定值取代所有元素
fill_n( )	用一个给定值取代前*n*个元素
generate( )	用一个操作的结果取代所有元素
generate_n( )	用一个操作的结果取代前*n*个元素
remove( )	删除具有给定值的元素
remove_if( )	删除满足一个谓词的元素
remove_copy( )	复制序列时删除给定值的元素
remove_copy_if( )	复制序列时删除满足谓词的元素
unique( )	删除相邻的重复元素
unique_copy( )	复制序列时删除相邻的重复元素
reverse( )	反转元素的次序
reverse_copy( )	复制序列时反转元素的次序
rotate( )	循环移动元素
rotate_copy( )	复制序列时循环移动元素
random_shuffle( )	采用均匀分布随机移动元素

序列排序（18.7节）<algorithm>	
sort( )	以很好的平均效率排序
stable_sort( )	排序，且维持相同元素原有的顺序
partial_sort( )	将序列的前一部分排好序
partial_sort_copy( )	复制的同时将序列的前一部分排好序
nth_element( )	将第*n*个元素放到它的正确位置
lower_bound( )	找到某个值的第一个出现
upper_bound( )	找到大于某个值的第一个元素

序列排序（18.7节）<algorithm>（续）	
equal_range( )	找出具有给定值的一个子序列
binary_search( )	在排好序的序列中确定给定元素是否存在
merge( )	归并两个排好序的序列
inplace_merge( )	归并两个接续的排好序的序列
partition( )	将满足某谓词的元素都放到前面
stable_partition( )	将满足某谓词的元素都放到前面且维持原顺序

集合算法（18.7.5节）<algorithm>	
include( )	如果一个序列是另一个的子序列则真
set_union( )	构造一个已排序的并集
set_intersection( )	构造一个已排序的交集
set_difference( )	构造一个已排序序列，其中包含所有在第一个序列中但不在第二个序列中的元素
set_symmetric_ difference( )	构造一个已排序序列，其中包括所有只在两个序列之一中的元素

堆操作（18.8节）<algorithm>	
make_heap( )	将序列调整得能够作为堆使用
push_heap( )	向堆中加入一个元素
pop_heap( )	从堆中去除元素
sort_heap( )	对堆排序

基于比较的选择元素的算法：

最大和最小（18.9节）<algorithm>	
min( )	两个值中较小的
max( )	两个值中较大的
min_element( )	序列中的最小元素
max_element( )	序列中的最大元素
lexicographic_compare( )	两个序列中按字典序第一个在前

排列（18.10节）<algorithm>	
next_permutation( )	按字典序的下一个排列
prev_permutation( )	按字典序的前一个排列

- 谓词是返回为 bool 类型的函数对象

谓词 <functional>		
equal_to	二元	arg1 == arg2
not_equal_to	二元	arg1 != arg2
greater	二元	arg1 > arg2
less	二元	arg1 < arg2
greater_equal	二元	arg1 >= arg2
less_equal	二元	arg1 <= arg2
logical_and	二元	arg1 && arg2
logical_or	二元	arg1 || arg2
logical_not	一元	!arg

算术运算 <functional>		
plus	二元	arg1 + arg2
minus	二元	arg1 – arg2
multiplies	二元	arg1 * arg2
divides	二元	arg1 / arg2
modulus	二元	arg1 % arg2
negate	一元	– arg

约束器、适配器和否定器 <functional>		
bind2nd(y)	*binder2nd*	以*y*作为第二个参数调用二元函数
bind1st(x)	*binder1st*	以*x*作为第一个参数调用二元函数
mem_fun( )	*mem_fun_t*	通过指针调用0元成员函数
	mem_fun1_t	通过指针调用一元成员函数
	const_mem_fun_t	通过指针调用0元*const*成员函数
	const_mem_fun1_t	通过指针调用一元*const*成员函数

约束器、适配器和否定器 <functional> （续）		
mem_fun_ref( )	*mem_fun_ref_t*	通过引用调用0元成员函数
	mem_fun1_ref_t	通过引用调用一元成员函数
	const_mem_fun_ref_t	通过引用调用0元*const*成员函数
	const_mem_fun1_ref_t	通过引用调用一元*const*成员函数
ptr_fun( )	*pointer_to_unary_function*	调用一元函数指针
ptr_fun( )	*pointer_to_binary_function*	调用二元函数指针
not1( )	*unary_negate*	否定一元谓词
not2( )	*binary_negate*	否定二元谓词

迭代器操作

	迭代器操作和类别				
类别： 简写：	输出 *Out*	输入 *In*	前向 *For*	双向 *Bi*	随机访问 *Ran*
读：		=*p	=*p	=*p	=*p
访问：		->	->	->	-> []
写：	*p=		*p=	*p=	*p=
迭代：	++	++	++	++ --	++ -- + - += -=
比较：		== !=	== !=	== !=	== != < > <= >=

读和写操作都通过由 * 表示的迭代器间接引用：

```
*p = x;    // 通过p写x
x = *p;    // 通过p读到x里
```

插入器

```
template <class Cont> back_insert_iterator<Cont> back_inserter(Cont& c);
template <class Cont> front_insert_iterator<Cont> front_inserter(Cont& c);
template <class Cont, class Out> insert_iterator<Cont> inserter(Cont& c, Out p);
```

流迭代器

— *ostream_iterator*：用于向*ostream*写入（3.4节、21.2.1节）。
— *istream_iterator*：用于由*istream*读出（3.6节、21.3.1节）。
— *ostreambuf_iterator*：用于向流缓冲区写入（21.6.1节）。
— *istreambuf_iterator*：用于由流缓冲区读出（21.6.2节）。

分配器

```
template <class T> class std::allocator {
public:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    typedef T* pointer;
    typedef const T* const_pointer;

    typedef T& reference;
    typedef const T& const_reference;
```

```
    pointer address(reference r) const { return &r; }
    const_pointer address(const_reference r) const { return &r; }

    allocator() throw();
    template <class U> allocator(const allocator<U>&) throw();
    ~allocator() throw();

    pointer allocate(size_type n, allocator<void>::const_pointer hint = 0); // 给n个T的空间
    void deallocate(pointer p, size_type n);  // 释放n个T, 不销毁

    void construct(pointer p, const T& val) { new(p) T(val); }     // 用val初始化 *p
    void destroy(pointer p) { p->~T(); }           // 销毁 *p, 不释放

    size_type max_size() const throw();

    template <class U>
    struct rebind { typedef allocator<U> other; };  // 效果: typedef allocator<U> other
};

template<class T> bool operator==(const allocator<T>&, const allocator<T>&) throw();
template<class T> bool operator!=(const allocator<T>&, const allocator<T>&) throw();
```

new、delete

```
class bad_alloc : public exception { /* ... */ };

struct nothrow_t {};
extern const nothrow_t nothrow;      // 分配时不抛出异常的指示符

typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();

void* operator new(size_t) throw(bad_alloc);
void operator delete(void*) throw();

void* operator new(size_t, const nothrow_t&) throw();
void operator delete(void*, const nothrow_t&) throw();

void* operator new[](size_t) throw(bad_alloc);
void operator delete[](void*) throw();

void* operator new[](size_t, const nothrow_t&) throw();

void operator delete[](void*, const nothrow_t&) throw();

void* operator new(size_t, void* p) throw() { return p; }      // 放置 (10.4.11节)
void operator delete(void* p, void*) throw() { }    // 什么也不做

void* operator new[](size_t, void* p) throw() { return p; }
void operator delete[](void* p, void*) throw() { } // 什么也不做
```

```
void f()
{
    int* p = new int[100000]; // 可能抛出bad_alloc

    if (int* q = new(nothrow) int[100000]) { // 不会抛出异常
        // 分配成功
    }
    else {
        // 分配失败
    }
}
```