1 记号

- 用 const 标识 `const`.
- 用 ↶ 标识 函数返回值.
- 模板类参数首字母大写. 如: 𝕋, 𝕂ey, ℂompare. 用于 `template` 定义.
- 有时省略 `class`, `typename`.
- 模板类参数省略, 如 C 有时用来替代 C⟨𝕋⟩.
- “示例” 如 ☞, 输出如 ✇➠.

2 容器

2.1 对

#include <utility>

```
template⟨class 𝕋1, class 𝕋2⟩
struct pair {
      𝕋1 first;   𝕋2 second;
      pair() {}
      pair(const 𝕋1& a, const 𝕋2& b):
          first(a), second(b) {}   };
```

2.1.1 类型

pair::**first_type**
pair::**second_type**

2.1.2 函数 & 操作符

参见 2.2.3.

pair⟨𝕋1,𝕋2⟩
make_pair(const 𝕋1&, const 𝕋2&);

2.2 容器 — 公用

这里 X 表示下列类之一
{**vector**, **deque**, **list**,
 set, **multiset**, **map**, **multimap**}

2.2.1 类型

X::**value_type**
X::**reference**
X::**const_reference**
X::**iterator**
X::**const_iterator**
X::**reverse_iterator**
X::**const_reverse_iterator**
X::**difference_type**
X::**size_type**

迭代器引用 value_type (见 6).

2.2.2 成员函数 & 操作符

```
X::X();
X::X(const X&);
X::~X();
X& X::operator=(const X&);

X::iterator                X::begin();
X::const_iterator          X::begin()      const ;
X::iterator                X::end();
X::const_iterator          X::end()        const ;
X::reverse_iterator        X::rbegin();
X::const_reverse_iterator  X::rbegin()     const ;
X::reverse_iterator        X::rend();
X::const_reverse_iterator  X::rend()       const ;

X::size_type   X::size() const ;
X::size_type   X::max_size() const ;
bool           X::empty() const ;
void           X::swap(X& x);

void X::clear();
```

2.2.3 比较操作符

令, X v, w. X 可以是 **pair** (2.1).

```
v  ==  w      v  !=  w
v  <   w      v  >   w
v  <=  w      v  >=  w
```

按字典序且 ↶`bool`.

2.3 顺序容器

S 是以下类之一 {**vector**, **deque**, **list**}

2.3.1 构造器

```
S::S(S::size_type            n,
     const S::value_type&    t);
S::S(S::const_iterator  first,
     S::const_iterator  last);   ☞7.2, 7.3
```

2.3.2 方法

```
S::iterator  // 插入复制
S::insert(S::iterator              before,
          const S::value_type&     val);

S::iterator  // 插入复制
S::insert(S::iterator              before,
          S::size_type             nVal,
          const S::value_type&     val);

S::iterator  // 插入复制
S::insert(S::iterator          before,
          S::const_iterator    first,
          S::const_iterator    last);

S:iterator S::erase(S::iterator  position);

S:iterator S::erase(S::const_iterator  first,
↶ 最后移除的        S::const_iterator  last);
```

```
void S::push_back(const S::value_type& x);

void S::pop_back();

S::reference S::front();

S::const_reference S::front() const ;

S::reference S::back();

S::const_reference S::back() const ;
```

2.4 向量

#include <vector>

```
template⟨class 𝕋,
         class 𝔸lloc=allocator⟩
class vector;
```

参见 2.2 和 2.3.

```
size_type vector::capacity() const ;

void vector::reserve(size_type n);

vector::reference
vector::operator[](size_type i);

vector::const_reference
vector::operator[](size_type i) const ;
```

☞ 7.1.

2.5 双向队列

#include <deque>

```
template⟨class 𝕋,
         class 𝔸lloc=allocator⟩
class deque;
```

包含 **vector** 所有方法 (见 2.4).

```
void deque::push_front(const 𝕋& x);

void deque::pop_front();
```

2.6 链表

#include <list>

```
template⟨class 𝕋,
         class 𝔸lloc=allocator⟩
class list;
```

参见 2.2 和 2.3.

```
void list::pop_front();

void list::push_front(const 𝕋& x);

void  // 移动 pos 前的所有 x (&x ≠ this)
list::splice(iterator pos, list⟨𝕋⟩& x);   ☞7.2

void  // 移动 pos 前 x 的 xElemPos
list::splice (iterator    pos,
              list⟨𝕋⟩&    x,
              iterator    xElemPos);   ☞7.2

void  // 移动 pos 前 x 的 [xFirst,xLast)
list::splice (iterator    pos,
              list⟨𝕋⟩&    x,
              iterator    xFirst,
              iterator    xLast);   ☞7.2

void list::remove(const 𝕋& value);

void list::remove_if(ℙredicate pred);
 // 调用后: ∀ this 迭代器 p, *p ≠ *(p + 1)
void list::unique();  // 移除重复

void  // 同上, 但 ¬binPred(*p, *(p + 1))
list::unique(𝔹inaryPredicate binPred);
 // 假定 this 和 x 有序
void list::merge(list⟨𝕋⟩& x);
 // 合并且假设以 cmp 排序
void list::merge(list⟨𝕋⟩& x, ℂompare cmp);

void list::reverse();

void list::sort();

void list::sort(ℂompare cmp);
```

2.7 有序集合

这里 A 是以下类之一
  {**set**, **multiset**, **map**, **multimap**}.

2.7.1 类型

对 A=[multi]set 都有

```
A::key_type       A::value_type
A::key_compare    A::value_compare
```

2.7.2 构造器

```
A::A(ℂompare c=ℂompare())

A::A(A::const_iterator   first,
     A::const_iterator   last,
     ℂompare             c=ℂompare());
```

2.7.3 成员函数

```
A::key_compare     A::key_comp() const ;
A::value_compare   A::value_comp() const ;

A::iterator
A::insert(A::iterator              hint,
          const A::value_type&     val);

void A::insert(A::iterator  first,
               A::iterator  last);

A::size_type  // # 移除的
A::erase(const A::key_type& k);

void A::erase(A::iterator p);
void A::erase(A::iterator  first,
              A::iterator  last);

A::size_type
A::count(const A::key_type& k) const ;

A::iterator A::find(const A::key_type& k) const ;
```

A::iterator
A::**lower_bound**(const A::key_type& *k*) const ;

A::iterator
A::**upper_bound**(const A::key_type& *k*) const ;

pair⟨A::iterator, A::iterator⟩ // 见 *4.3.1*
A::**equal_range**(const A::key_type& *k*) const ;

2.8 集合

#include <set>

template⟨class 𝕂ey,
class ℂompare=less⟨𝕂ey⟩,
class 𝔸lloc=allocator⟩
class **set**;

参见 2.2 和 2.7.

set::**set**(const ℂompare& *cmp*=ℂompare());

pair⟨set::iterator, bool⟩ // *bool* = 是否为新
set::**insert**(const set::value_type& *x*);

2.9 多重集合

#include <set>

template⟨class 𝕂ey,
class ℂompare=less⟨𝕂ey⟩,
class 𝔸lloc=allocator⟩
class **multiset**;

参见 2.2 和 2.7.

multiset::**multiset**(
const ℂompare& *cmp*=ℂompare());

multiset::**multiset**(
𝕀nputIterator *first*,
𝕀nputIterator *last*,
const ℂompare& *cmp*=ℂompare());

multiset::iterator // 插入复制
multiset::**insert**(const multiset::value_type& *x*);

2.10 映射

#include <map>

template⟨class 𝕂ey, class 𝕋,
class ℂompare=less⟨𝕂ey⟩,
class 𝔸lloc=allocator⟩
class **map**;

参见 2.2 和 2.7.

2.10.1 类型

map::**value_type** // pair⟨const 𝕂ey,𝕋⟩

2.10.2 成员函数

map::**map**(
const ℂompare& *cmp*=ℂompare());

pair⟨map::iterator, bool⟩ // *bool* = 是否为新
map::**insert**(const map::value_type& *x*);

𝕋& map:**operator**[](const map::key_type&);

map::const_iterator
map::**lower_bound**(
const map::key_type& *k*) const ;

map::const_iterator
map::**upper_bound**(
const map::key_type& *k*) const ;

pair⟨map::const_iterator, map::const_iterator⟩
map::**equal_range**(

const map::key_type& *k*) const ;

例

```
typedef map<string, int> MSI;
MSI  nam2num;
nam2num.insert(MSI::value_type("one", 1));
nam2num.insert(MSI::value_type("two", 2));
nam2num.insert(MSI::value_type("three", 3));
int n3 = nam2num["one"] + nam2num["two"];
cout << n3 << " called ";
for (MSI::const_iterator i = nam2num.begin();
  i != nam2num.end();   ++i)
  if ((*i).second == n3) {
    cout << (*i).first << endl;
  }
```

```
3 called three
```

2.11 多重映射

#include <map>

template⟨class 𝕂ey, class 𝕋,
class ℂompare=less⟨𝕂ey⟩,
class 𝔸lloc=allocator⟩
class **multimap**;

参见 2.2 和 2.7.

2.11.1 类型

multimap::**value_type** // pair⟨const 𝕂ey,𝕋⟩

2.11.2 成员函数

multimap::**multimap**(
const ℂompare& *cmp*=ℂompare());

multimap::**multimap**(
𝕀nputIterator *first*,
𝕀nputIterator *last*,
const ℂompare& *cmp*=ℂompare());

multimap::const_iterator
multimap::**lower_bound**(
const multimap::key_type& *k*) const ;

multimap::const_iterator
multimap::**upper_bound**(
const multimap::key_type& *k*) const ;

pair⟨multimap::const_iterator,
multimap::const_iterator⟩
multimap::**equal_range**(
const multimap::key_type& *k*) const ;

3 容器适配器

3.1 栈

#include <stack>

template⟨class 𝕋,
class ℂontainer=deque⟨𝕋⟩ ⟩
class **stack**;

默认构造器. ℂontainer 要有 `back()`, `push_back()`, `pop_back()`. 所以 **vector**, **list** 和 **deque** 可用.

bool stack::**empty**() const ;

ℂontainer::size_type stack::**size**() const ;

void
stack::**push**(const ℂontainer::value_type& x);

void stack::**pop**();

const ℂontainer::value_type&
stack::**top**() const ;

ℂontainer::value_type& stack::**top**();

比较操作符

bool **operator==**(const stack& *s0*,
const stack& *s1*);

bool **operator<**(const stack& *s0*,
const stack& *s1*);

3.2 队列

#include <queue>

template⟨class 𝕋,
class ℂontainer=deque⟨𝕋⟩ ⟩
class **queue**;

默认构造器. ℂontainer 要有 `empty()`, `size()`, `back()`, `front()`, `push_back()` 和 `pop_front()`. 所以 **list** 和 **deque** 可用.

bool queue::**empty**() const ;

ℂontainer::size_type queue::**size**() const ;

void
queue::**push**(const ℂontainer::value_type& x);

void queue::**pop**();

const ℂontainer::value_type&
queue::**front**() const ;

ℂontainer::value_type& queue::**front**();

const ℂontainer::value_type&
queue::**back**() const ;

ℂontainer::value_type& queue::**back**();

比较操作符

bool **operator==**(const queue& *q0*,
const queue& *q1*);

bool **operator<**(const queue& *q0*,
const queue& *q1*);

3.3 优先队列

#include <queue>

template⟨class 𝕋,
class ℂontainer=vector⟨𝕋⟩,
class ℂompare=less⟨𝕋⟩ ⟩
class **priority_queue**;

ℂontainer 必须提供随机访问迭代器且有 `empty()`, `size()`, `front()`, `push_back()` 和 `pop_back()`. 所以 **vector** 和 **deque** 可用.

多以 堆实现.

3.3.1 构造器

explicit **priority_queue::priority_queue**(
const ℂompare& comp=ℂompare());

priority_queue::priority_queue(
𝕀nputIterator *first*,
𝕀nputIterator *last*,
const ℂompare& *comp*=ℂompare());

3.3.2 成员函数

bool priority_queue::**empty**() const ;

ℂontainer::size_type
priority_queue::**size**() const ;

const ℂontainer::value_type&
priority_queue::**top**() const ;

ℂontainer::value_type& priority_queue::**top**();

void priority_queue::**push**(
const ℂontainer::value_type& x);

void priority_queue::**pop**();

无比较操作符.

4 算法

#include <algorithm>

STL 算法使用迭代器类型参数. 名适应类 (见 6.1).

声明 `template ⟨class 𝔽oo, ...⟩` 省略, 用首字母大写表示 template.

注意: 以下两个序列: $S_1 = [first_1, last_1)$ 和 $S_2 = [first_2, ?)$ 或 $S_2 = [?, last_2)$ — 表示调用函数不会超过 S_2.

4.1 查询算法

```
𝔽unction // f 不改变 [first, last)
for_each(𝕀nputIterator first,
         𝕀nputIterator last,
         𝔽unction      f);  ☞7.4
```

```
𝕀nputIterator // 首个 i 满足 i==last 或 *i==val
find(𝕀nputIterator first,
     𝕀nputIterator last,
     const 𝕋       val);  ☞7.2
```

```
𝕀nputIterator // 首个 i 满足 i==last 或 pred(i)
find_if(𝕀nputIterator first,
        𝕀nputIterator last,
        ℙredicate     pred);  ☞7.7
```

```
𝔽orwardIterator // 首个重复
adjacent_find(𝔽orwardIterator first,
              𝔽orwardIterator last);
```

```
𝔽orwardIterator // 首个 binPred-定义重复
adjacent_find(𝔽orwardIterator first,
              𝔽orwardIterator last,
              𝔹inaryPredicate binPred);
```

```
void // n = # 等于 val
count(𝔽orwardIterator first,
      𝔽orwardIterator last,
      const 𝕋         val,
      𝕊ize&           n);
```

```
void // n = # 满足 pred
count_if(𝔽orwardIterator first,
         𝔽orwardIterator last,
         ℙredicate       pred,
         𝕊ize&           n);
```

```
// ↶ 首个 != 值对
pair⟨𝕀nputIterator1, 𝕀nputIterator2⟩
mismatch(𝕀nputIterator1 first1,
         𝕀nputIterator1 last1,
         𝕀nputIterator2 first2);
```

```
// ↶ 首个 binPred-定义不匹配值对
pair⟨𝕀nputIterator1, 𝕀nputIterator2⟩
mismatch(𝕀nputIterator1  first1,
         𝕀nputIterator1  last1,
         𝕀nputIterator2  first2,
         𝔹inaryPredicate binPred);
```

```
bool
equal(𝕀nputIterator1 first1,
      𝕀nputIterator1 last1,
      𝕀nputIterator2 first2);
```

```
bool
equal(𝕀nputIterator1  first1,
      𝕀nputIterator1  last1,
      𝕀nputIterator2  first2,
      𝔹inaryPredicate binPred);
```

// $[first_2, last_2) \sqsubseteq [first_1, last_1)$

```
𝔽orwardIterator1
search(𝔽orwardIterator1 first1,
       𝔽orwardIterator1 last1,
       𝔽orwardIterator2 first2,
       𝔽orwardIterator2 last2);
```

// $[first_2, last_2) \sqsubseteq_{binPred} [first_1, last_1)$

```
𝔽orwardIterator1
search(𝔽orwardIterator1 first1,
       𝔽orwardIterator1 last1,
       𝔽orwardIterator2 first2,
       𝔽orwardIterator2 last2,
       𝔹inaryPredicate  binPred);
```

4.2 修改算法

$𝕆utputIterator$ // ↶ $first_2 + (last_1 - first_1)$

```
copy(𝕀nputIterator  first1,
     𝕀nputIterator  last1,
     𝕆utputIterator first2);
```

// ↶ $last_2 - (last_1 - first_1)$

```
𝔹idirectionalIterator2
copy_backward(
    𝔹idirectionalIterator1 first1,
    𝔹idirectionalIterator1 last1,
    𝔹idirectionalIterator2 last2);
```

```
void swap(𝕋& x, 𝕋& y);
```

𝔽orwardIterator2 // ↶ $first_2 + \#[first_1, last_1)$

```
swap_ranges(𝔽orwardIterator1 first1,
            𝔽orwardIterator1 last1,
            𝔽orwardIterator2 first2);
```

𝕆utputIterator // ↶ $result + (last_1 - first_1)$

```
transform(𝕀nputIterator  first,
          𝕀nputIterator  last,
          𝕆utputIterator result,
          𝕌naryOperation op);  ☞7.6
```

𝕆utputIterator // $\forall s_i^k \in S_k \; r_i = \mathrm{bop}(s_i^1, s_i^2)$

```
transform(𝕀nputIterator1   first1,
          𝕀nputIterator1   last1,
          𝕀nputIterator2   first2,
          𝕆utputIterator   result,
          𝔹inaryOperation  bop);
```

```
void replace(𝔽orwardIterator first,
             𝔽orwardIterator last,
             const 𝕋&        oldVal,
             const 𝕋&        newVal);
```

```
void
replace_if(𝔽orwardIterator first,
           𝔽orwardIterator last,
           ℙredicate&      pred,
           const 𝕋&        newVal);
```

𝕆utputIterator // ↶ $result_2 + \#[first, last)$

```
replace_copy(𝕀nputIterator  first,
             𝕀nputIterator  last,
             𝕆utputIterator result,
             const 𝕋&       oldVal,
             const 𝕋&       newVal);
```

```
𝕆utputIterator // 同上但用 pred
replace_copy_if(𝕀nputIterator  first,
                𝕀nputIterator  last,
                𝕆utputIterator result,
                ℙredicate&     pred,
                const 𝕋&       newVal);
```

```
void fill(𝔽orwardIterator first,
          𝔽orwardIterator last,
          const 𝕋&        value);
```

```
void fill_n(𝔽orwardIterator first,
            𝕊ize            n,
            const 𝕋&        value);
```

```
void // 区间调用 gen()
generate(𝔽orwardIterator first,
         𝔽orwardIterator last,
         𝔾enerator       gen);
```

```
void // n 次调用 gen()
generate_n(𝔽orwardIterator first,
           𝕊ize            n,
           𝔾enerator       gen);
```

所有 **remove** 和 **unique** 的变体返回指向新的末端或上次复制的迭代器.

```
𝔽orwardIterator // [↶,last) is all value
remove(𝔽orwardIterator first,
       𝔽orwardIterator last,
       const 𝕋&        value);
```

```
𝔽orwardIterator // 同上但用 pred
remove_if(𝔽orwardIterator first,
          𝔽orwardIterator last,
          ℙredicate       pred);
```

```
𝕆utputIterator // ↶ 上次复制
remove_copy(𝕀nputIterator  first,
            𝕀nputIterator  last,
            𝕆utputIterator result,
            const 𝕋&       value);
```

```
𝕆utputIterator // 同上但用 pred
remove_copy_if(𝕀nputIterator  first,
               𝕀nputIterator  last,
               𝕆utputIterator result,
               ℙredicate      pred);
```

所以 **unique** 模板函数的变体移除连续 (binPred-) 重复. 排序后很有用 (见 4.3).

```
𝔽orwardIterator // [↶,last) 得重复
unique(𝔽orwardIterator first,
       𝔽orwardIterator last);
```

```
𝔽orwardIterator // 同上但用 binPred
unique(𝔽orwardIterator first,
       𝔽orwardIterator last,
       𝔹inaryPredicate binPred);
```

```
𝕆utputIterator // ↶ 上次复制
unique_copy(𝕀nputIterator  first,
            𝕀nputIterator  last,
            𝕆utputIterator result);
```

```
𝕆utputIterator // 同上但用 binPred
unique_copy(𝕀nputIterator   first,
            𝕀nputIterator   last,
            𝕆utputIterator  result,
            𝔹inaryPredicate binPred);
```

```
void
reverse(𝔹idirectionalIterator first,
        𝔹idirectionalIterator last);
```

```
𝕆utputIterator // ↶ 上次复制
reverse_copy(𝔹idirectionalIterator first,
             𝔹idirectionalIterator last,
             𝕆utputIterator        result);
```

```
void // 将 first 移动到 middle
rotate(𝔽orwardIterator first,
       𝔽orwardIterator middle,
       𝔽orwardIterator last);
```

```
𝕆utputIterator // first 到 middle 位置
rotate_copy(𝔽orwardIterator first,
            𝔽orwardIterator middle,
            𝔽orwardIterator last,
            𝕆utputIterator  result);
```

```
void
random_shuffle(
    ℝandomAccessIterator first,
    ℝandomAccessIterator last);
```

```
void // rand() 返回 [0,1) 间的 double
random_shuffle(
    ℝandomAccessIterator first,
    ℝandomAccessIterator last,
    ℝandomGenerator      rand);
```

```
𝔹idirectionalIterator // 以 true 开始
partition(𝔹idirectionalIterator first,
          𝔹idirectionalIterator last,
          ℙredicate             pred);
```

```
𝔹idirectionalIterator // 以 true 开始
stable_partition(
    𝔹idirectionalIterator first,
    𝔹idirectionalIterator last,
    ℙredicate             pred);
```

4.3 排序和应用

```
void sort(ℝandomAccessIterator first,
          ℝandomAccessIterator last);
```

```
void sort(ℝandomAccessIterator first,
          ℝandomAccessIterator last,
  ☞7.3    ℂompare              comp);
```

```
void
stable_sort(ℝandomAccessIterator first,
            ℝandomAccessIterator last);
```

```
void
stable_sort(ℝandomAccessIterator first,
            ℝandomAccessIterator last,
            ℂompare              comp);
```

```
void          // [first,middle) 有序,
partial_sort( // [middle,last) 大于等于
    ℝandomAccessIterator first,
    ℝandomAccessIterator middle,
    ℝandomAccessIterator last);
```

```
void  // 同上但用 comp(e_i, e_j)
partial_sort(
    ℝandomAccessIterator first,
    ℝandomAccessIterator middle,
    ℝandomAccessIterator last,
    ℂompare              comp);
```

```
ℝandomAccessIterator     // 上次排序
partial_sort_copy(
    𝕀nputIterator        first,
    𝕀nputIterator        last,
    ℝandomAccessIterator resultFirst,
    ℝandomAccessIterator resultLast);
```

```
ℝandomAccessIterator
partial_sort_copy(
    𝕀nputIterator        first,
    𝕀nputIterator        last,
    ℝandomAccessIterator resultFirst,
    ℝandomAccessIterator resultLast,
    ℂompare              comp);
```

令 $n = position - first$, **nth_element** 划分 $[first, last)$ 到: $L = [first, position)$, e_n, $R = [position + 1, last)$ 使得 $\forall l \in L, \forall r \in R \quad l \not> e_n \leq r$.

```
void
nth_element(
    ℝandomAccessIterator first,
    ℝandomAccessIterator position,
    ℝandomAccessIterator last);
```

```
void  // 同上但用 comp(e_i, e_j)
nth_element(
    ℝandomAccessIterator first,
    ℝandomAccessIterator position,
    ℝandomAccessIterator last,
    ℂompare              comp);
```

4.3.1 二分查找

```
bool
binary_search(𝔽orwardIterator first,
              𝔽orwardIterator last,
              const 𝕋&        value);
```

```
bool
binary_search(𝔽orwardIterator first,
              𝔽orwardIterator last,
              const 𝕋&        value,
              ℂompare         comp);
```

```
𝔽orwardIterator
lower_bound(𝔽orwardIterator first,
            𝔽orwardIterator last,
            const 𝕋&        value);
```

```
ForwardIterator
lower_bound(𝔽orwardIterator first,
            𝔽orwardIterator last,
            const 𝕋&        value,
            ℂompare         comp);
```

```
𝔽orwardIterator
upper_bound(𝔽orwardIterator first,
            𝔽orwardIterator last,
            const 𝕋&        value);
```

```
ForwardIterator
upper_bound(𝔽orwardIterator first,
            𝔽orwardIterator last,
            const 𝕋&        value,
            ℂompare         comp);
```

equal_range 返回 lower_bound 和 upper_bound 返回的迭代器对.

```
pair⟨𝔽orwardIterator,𝔽orwardIterator⟩
equal_range(𝔽orwardIterator first,
            𝔽orwardIterator last,
            const 𝕋&        value);
```

```
pair⟨𝔽orwardIterator,𝔽orwardIterator⟩
equal_range(𝔽orwardIterator first,
            𝔽orwardIterator last,
            const 𝕋&        value,
            ℂompare         comp);
```

☞ 7.5

4.3.2 合并

假设 $S_1 = [first_1, last_1)$ 和 $S_2 = [first_2, last_2)$ 有序, 稳定合并入 $[result, result + N)$ 其中 $N = |S_1| + |S_2|$.

```
𝕆utputIterator
merge(𝕀nputIterator1  first1,
      𝕀nputIterator1  last1,
      𝕀nputIterator2  first2,
      𝕀nputIterator2  last2,
      𝕆utputIterator  result);
```

```
𝕆utputIterator
merge(𝕀nputIterator1  first1,
      𝕀nputIterator1  last1,
      𝕀nputIterator2  first2,
      𝕀nputIterator2  last2,
      𝕆utputIterator  result,
      ℂompare         comp);
```

```
void  // 区间 [first,middle) [middle,last)
inplace_merge( // 到 [first,last)
    𝔹idirectionalIterator first,
    𝔹idirectionalIterator middle,
    𝔹idirectionalIterator last);
```

```
void  // 同上但用 comp
inplace_merge(
    𝔹idirectionalIterator first,
    𝔹idirectionalIterator middle,
    𝔹idirectionalIterator last,
    ℂompare               comp);
```

4.3.3 函数和集合

用于操作有序集合容器 (见 2.7). 对于 **multiset** 的操作 — *union*, *intersection* 和 *difference* 取决于: *maximum*, *minimum* 和 *substraction* 的行为.

令 $S_i = [first_i, last_i)$ 对于 $i = 1, 2$.

```
bool  // S_1 ⊇ S_2
includes(𝕀nputIterator1 first1,
         𝕀nputIterator1 last1,
         𝕀nputIterator2 first2,
         𝕀nputIterator2 last2);
```

```
bool  // 同上但用 comp
includes(𝕀nputIterator1 first1,
         𝕀nputIterator1 last1,
         𝕀nputIterator2 first2,
         𝕀nputIterator2 last2,
         ℂompare        comp);
```

```
𝕆utputIterator // S_1 ∪ S_2, ↶ 上次结尾
set_union(𝕀nputIterator1 first1,
          𝕀nputIterator1 last1,
          𝕀nputIterator2 first2,
          𝕀nputIterator2 last2,
          𝕆utputIterator result);
```

```
𝕆utputIterator // 同上但用 comp
set_union(𝕀nputIterator1 first1,
          𝕀nputIterator1 last1,
          𝕀nputIterator2 first2,
          𝕀nputIterator2 last2,
          𝕆utputIterator result,
          ℂompare        comp);
```

```
𝕆utputIterator // S_1 ∩ S_2, ↶ 上次结尾
set_intersection(𝕀nputIterator1 first1,
                 𝕀nputIterator1 last1,
                 𝕀nputIterator2 first2,
                 𝕀nputIterator2 last2,
                 𝕆utputIterator result);
```

```
𝕆utputIterator // 同上但用 comp
set_intersection(𝕀nputIterator1 first1,
                 𝕀nputIterator1 last1,
                 𝕀nputIterator2 first2,
                 𝕀nputIterator2 last2,
                 𝕆utputIterator result,
                 ℂompare        comp);
```

```
𝕆utputIterator // S_1 \ S_2, ↶ 上次结尾
set_difference(𝕀nputIterator1 first1,
               𝕀nputIterator1 last1,
               𝕀nputIterator2 first2,
               𝕀nputIterator2 last2,
               𝕆utputIterator result);
```

```
𝕆utputIterator // 同上但用 comp
set_difference(𝕀nputIterator1 first1,
               𝕀nputIterator1 last1,
               𝕀nputIterator2 first2,
               𝕀nputIterator2 last2,
               𝕆utputIterator result,
               ℂompare        comp);
```

```
𝕆utputIterator // S_1 △ S_2, ↶ 上次结尾
set_symmetric_difference(
    𝕀nputIterator1 first1,
    𝕀nputIterator1 last1,
    𝕀nputIterator2 first2,
    𝕀nputIterator2 last2,
    𝕆utputIterator result);
```

```
𝕆utputIterator // 同上但用 comp
set_symmetric_difference(
    𝕀nputIterator1   first1,
    𝕀nputIterator1   last1,
    𝕀nputIterator2   first2,
    𝕀nputIterator2   last2,
    𝕆utputIterator   result,
    ℂompare          comp);
```

4.3.4 堆

```
void // (last − 1) 压入
push_heap(ℝandomAccessIterator   first,
          ℝandomAccessIterator   last);

void // 同上但用 comp
push_heap(ℝandomAccessIterator   first,
          ℝandomAccessIterator   last,
          ℂompare                comp);

void // first 弹出
pop_heap(ℝandomAccessIterator   first,
         ℝandomAccessIterator   last);

void // 同上但用 comp
pop_heap(ℝandomAccessIterator   first,
         ℝandomAccessIterator   last,
         ℂompare                comp);

void // [first,last) 乱序
make_heap(ℝandomAccessIterator   first,
          ℝandomAccessIterator   last);

void // 同上但用 comp
make_heap(ℝandomAccessIterator   first,
          ℝandomAccessIterator   last,
          ℂompare                comp);

void // 排序 [first,last) 堆
sort_heap(ℝandomAccessIterator   first,
          ℝandomAccessIterator   last);

void // 同上但用 comp
sort_heap(ℝandomAccessIterator   first,
          ℝandomAccessIterator   last,
          ℂompare                comp);
```

4.3.5 最大和最小

```
const 𝕋& min(const 𝕋& x0, const 𝕋& x1);

const 𝕋& min(const 𝕋&   x0,
             const 𝕋&   x1,
             ℂompare    comp);

const 𝕋& max(const 𝕋& x0, const 𝕋& x1);

const 𝕋& max(const 𝕋&   x0,
             const 𝕋&   x1,
             ℂompare    comp);

𝔽orwardIterator
min_element(𝔽orwardIterator   first,
            𝔽orwardIterator   last);

𝔽orwardIterator
min_element(𝔽orwardIterator   first,
            𝔽orwardIterator   last,
            ℂompare           comp);

𝔽orwardIterator
max_element(𝔽orwardIterator   first,
            𝔽orwardIterator   last);

𝔽orwardIterator
max_element(𝔽orwardIterator   first,
            𝔽orwardIterator   last,
            ℂompare           comp);
```

4.3.6 排列

所有排列, 开始于递增, 结束于递减.

```
bool // ↶ 当且仅当可用
next_permutation(
    𝔹idirectionalIterator   first,
    𝔹idirectionalIterator   last);

bool // 同上但用 comp
next_permutation(
    𝔹idirectionalIterator   first,
    𝔹idirectionalIterator   last,
    ℂompare                 comp);

bool // ↶ 当且仅当可用
prev_permutation(
    𝔹idirectionalIterator   first,
    𝔹idirectionalIterator   last);

bool // 同上但用 comp
prev_permutation(
    𝔹idirectionalIterator   first,
    𝔹idirectionalIterator   last,
    ℂompare                 comp);
```

4.3.7 字典序

```
bool lexicographical_compare(
        𝕀nputIterator1   first1,
        𝕀nputIterator1   last1,
        𝕀nputIterator2   first2,
        𝕀nputIterator2   last2);

bool lexicographical_compare(
        𝕀nputIterator1   first1,
        𝕀nputIterator1   last1,
        𝕀nputIterator2   first2,
        𝕀nputIterator2   last2,
        ℂompare          comp);
```

4.4 计算

#include <numeric>

```
𝕋 // $\sum_{[first,last)}$   ☞7.6
accumulate(𝕀nputIterator   first,
           𝕀nputIterator   last,
           𝕋               initVal);

𝕋 // 同上但用 binop
accumulate(𝕀nputIterator     first,
           𝕀nputIterator     last,
           𝕋                 initVal,
           𝔹inaryOperation   binop);

𝕋 // $\sum_i e_i^1 \times e_i^2$ 其中 $e_i^k \in S_k, (k = 1, 2)$
inner_product(𝕀nputIterator1   first1,
              𝕀nputIterator1   last1,
              𝕀nputIterator2   first2,
              𝕋                initVal);

𝕋 // 类似地, 使用 $\sum^{(sum)}$ 和 $\times_{mult}$
inner_product(𝕀nputIterator1    first1,
              𝕀nputIterator1    last1,
              𝕀nputIterator2    first2,
              𝕋                 initVal,
              𝔹inaryOperation   sum,
              𝔹inaryOperation   mult);

𝕆utputIterator // $r_k = \sum_{i=first}^{first+k} e_i$
partial_sum(𝕀nputIterator    first,
            𝕀nputIterator    last,
            𝕆utputIterator   result);

𝕆utputIterator // 同上但用 binop
partial_sum(
    𝕀nputIterator     first,
    𝕀nputIterator     last,
    𝕆utputIterator    result,
    𝔹inaryOperation   binop);

𝕆utputIterator // $r_k = s_k - s_{k-1}$ 其中 $k > 0$
adjacent_difference(               // $r_0 = s_0$
    𝕀nputIterator    first,
    𝕀nputIterator    last,
    𝕆utputIterator   result);

𝕆utputIterator // 同上但用 binop
adjacent_difference(
    𝕀nputIterator     first,
    𝕀nputIterator     last,
    𝕆utputIterator    result,
    𝔹inaryOperation   binop);
```

5 函数对象

#include <functional>

```
template⟨class 𝔸rg, class ℝesult⟩
struct unary_function {
   typedef 𝔸rg argument_type;
   typedef ℝesult result_type;}
```

预定义的一元对象:

```
struct negate⟨𝕋⟩;
struct logical_not⟨𝕋⟩;
```

☞ 7.6

```
template⟨class 𝔸rg1, class 𝔸rg2,
        class ℝesult⟩
struct binary_function {
  typedef 𝔸rg1 first_argument_type;
  typedef 𝔸rg2 second_argument_type;
  typedef ℝesult result_type;}
```

以下预定义模板对象接受两个操作数. 结果如其名.

```
struct plus⟨𝕋⟩;
struct minus⟨𝕋⟩;
struct multiplies⟨𝕋⟩;
struct divides⟨𝕋⟩;
struct modulus⟨𝕋⟩;
struct equal_to⟨𝕋⟩;
struct not_equal_to⟨𝕋⟩;
struct greater⟨𝕋⟩;
struct less⟨𝕋⟩;
struct greater_equal⟨𝕋⟩;
struct less_equal⟨𝕋⟩;
struct logical_and⟨𝕋⟩;
struct logical_or⟨𝕋⟩;
```

5.1 函数适配器

5.1.1 取反

```
template⟨class ℙredicate⟩
class unary_negate : public
  unary_function⟨ℙredicate::argument_type,
                 bool⟩;
```

```
unary_negate::unary_negate(
    ℙredicate pred);

bool // negate pred
unary_negate::operator()(
    ℙredicate::argument_type x);

unary_negate⟨ℙredicate⟩
not1(const ℙredicate pred);
```

```
template⟨class ℙredicate⟩
class binary_negate : public
  binary_function⟨
    ℙredicate::first_argument_type,
    ℙredicate::second_argument_type⟩;
    bool⟩;
```

```
binary_negate::binary_negate(
    ℙredicate pred);

bool  // 取反 pred
binary_negate::operator()(
    ℙredicate::first_argument_type   x
    ℙredicate::second_argument_type  y);

binary_negate⟨ℙredicate⟩
not2(const ℙredicate pred);
```

5.1.2 绑定

```
template⟨class 𝕆peration⟩
class binder1st: public
  unary_function⟨
    𝕆peration::second_argument_type,
    𝕆peration::result_type⟩;
```

```
binder1st::binder1st(
  const 𝕆peration&                    op,
  const 𝕆peration::first_argument_type y);

 // argument_type 来自 unary_function
𝕆peration::result_type
binder1st::operator()(
    const binder1st::argument_type x);

binder1st⟨𝕆peration⟩
bind1st(const 𝕆peration& op, const 𝕋& x);
```

```
template⟨class 𝕆peration⟩
class binder2nd: public
  unary_function⟨
    𝕆peration::first_argument_type,
    𝕆peration::result_type⟩;
```

```
binder2nd::binder2nd(
  const 𝕆peration&                     op,
  const 𝕆peration::second_argument_type y);

 // argument_type 来自 unary_function
𝕆peration::result_type
binder2nd::operator()(
    const binder2nd::argument_type x);

binder2nd⟨𝕆peration⟩
bind2nd(const 𝕆peration& op, const 𝕋& x);
```

☞ 7.7.

5.1.3 函数指针

```
template⟨class 𝔸rg, class ℝesult⟩
class pointer_to_unary_function :
  public unary_function⟨𝔸rg, ℝesult⟩;
```

```
pointer_to_unary_function⟨𝔸rg, ℝesult⟩
ptr_fun(ℝesult(*x)(𝔸rg));
```

```
template<class 𝔸rg1, class 𝔸rg2,
        class ℝesult>
class pointer_to_binary_function :
  public binary_function⟨𝔸rg1, 𝔸rg2,
                         ℝesult⟩;
```

```
pointer_to_binary_function⟨𝔸rg1, 𝔸rg2,
                           ℝesult⟩
ptr_fun(ℝesult(*x)(𝔸rg1, 𝔸rg2));
```

6 迭代器

#include <**iterator**>

6.1 迭代器分类

这里我们用:

`X`	迭代器类型.
`a`, `b`	迭代器值.
`r`	迭代器引用 (`X& r`).
`t`	一个 `T` 类型的值.

用空的 `struct` 标签.

6.1.1 输入、输出、前引用

struct **input_iterator_tag** {}☞ 7.8
struct **output_iterator_tag** {}
struct **forward_iterator_tag** {}

下表显示 **I**nput, **O**utput 和 **F**orward 迭代器.

表达式	条件	I	O	F

`X()`
`X u`
	可单用			●

`X(a)`	⇒`X(a) == a`	●		●
	`*a=t ⇔ *X(a)=t`		●	

`X u(a)`
`X u=a`
	⇒ `u == a`	●		●

	`u` copy of `a`		●	
`a==b`	equivalence relation	●		●
`a!=b`	⇔`!(a==b)`	●		●
`r = a`	⇒ `r == a`			●
`*a`	convertible to `T`. `a==b ⇔ *a==*b`	●		●
`*a=t`	(for *forward*, if `X` mutable)		●	●
`++r`	result is dereferenceable or *past-the-end*. `&r == &++r`	●	●	●
	convertible to `const X&`	●	●	
	convertible to `X&` `r==s⇔ ++r==++s`			●
`r++`	convertible to `X&` ⇔{`X x=r;++r;return x;`}	●	●	●
`*++r` `*r++`	convertible to `T`	●	●	●

☞ 7.7.

6.1.2 Bidirectional Iterators

struct **bidirectional_iterator_tag** {}

The **forward** requirements and:

`--r`	Convertible to const `X&`. If ∃ `r=++s` then `--r` refers same as `s`. `&r==&--r`. `--(++r)==r`. (`--r == --s` ⇒ `r==s`.
`r--`	⇔ {`X x=r; --r; return x;`}.

6.1.3 Random Access Iterator

struct **random_access_iterator_tag** {}

The **bidirectional** requirements and
(`m,n` iterator's *distance* (integral) value):

`r+=n`	⇔ {`for (m=n; m-->0; ++r);`
	`for (m=n; m++<0; --r);`
	`return r;`} //but time = $O(1)$.

`a+n`	⇔ `n+a` ⇔ {`X x=a; return a+=n]`}
`r-=n`	⇔ `r += -n`.
`a-n`	⇔ `a+(-n)`.
`b-a`	Returns iterator's *distance* value n, such that `a+n == b`.
`a[n]`	⇔ `*(a+n)`.
`a<b`	Convertible to `bool`, `<` total ordering.
`a<b`	Convertible to `bool`, `>` opposite to `<`.
`a<=b`	⇔ `!(a>b)`.
`a>=b`	⇔ `!(a<b)`.

6.2 Stream Iterators

```
template⟨class 𝕋,
        class 𝔻istance=ptrdiff_t⟩
class istream_iterator :
    pubic iterator⟨input_iterator_tag, 𝕋, 𝔻istance⟩;
```

```
 // end of stream  ☞7.4
istream_iterator::istream_iterator();

istream_iterator::istream_iterator(
    istream& s);  ☞7.4

istream_iterator::istream_iterator(
    const istream_iterator⟨𝕋, 𝔻istance⟩&);

istream_iterator::~istream_iterator();

const 𝕋& istream_iterator::operator*() const;

istream_iterator&  // Read and store 𝕋 value
istream_iterator::operator++() const;

bool  // all end-of-streams are equal
operator==(const istream_iterator,
           const istream_iterator);
```

```
template⟨class 𝕋⟩
class ostream_iterator :
    public iterator⟨output_iterator_tag, void, ...⟩;
```

```
 // If delim ≠ 0 add after each write
ostream_iterator::ostream_iterator(
    ostream&      s,
    const char*   delim=0);

ostream_iterator::ostream_iterator(
    const ostream_iterator s);

ostream_iterator&  // Assign & write (*o=t)
ostream_iterator::operator*() const;

ostream_iterator&
ostream_iterator::operator=(
    const ostream_iterator s);

ostream_iterator&  // No-op
ostream_iterator::operator++();

ostream_iterator&  // No-op
ostream_iterator::operator++(int);
```

☞ 7.4.

6.3 类型定义 & 适配器

```
template⟨ℂategory, 𝕋,
        𝔻istance=ptrdiff_t,
        ℙointer=𝕋*, ℝeference= 𝕋&⟩
class iterator {
    ℂategory   iterator_category;
    𝕋          value_type;
    𝔻istance   difference_type;
    ℙointer    pointer;
    ℝeference  reference;}
```

6.3.1 Traits

```
template⟨𝕀⟩
class iterator_traits {
  𝕀::iterator_category
                  iterator_category;
  𝕀::value_type       value_type;
  𝕀::difference_type  difference_type;
  𝕀::pointer          pointer;
  𝕀::reference        reference;}
```

Pointer specilaizations: ☞ 7.8

```
template⟨𝕋⟩
class iterator_traits⟨𝕋*⟩ {
  random_access_iterator_tag
          iterator_category ;
  𝕋        value_type;
  ptrdiff_t difference_type;
  𝕋*       pointer;
  𝕋&       reference;}
```

template⟨𝕋⟩
class **iterator_traits**⟨const 𝕋*⟩ {
  random_access_iterator_tag
    iterator_category ;
  𝕋   **value_type**;
  ptrdiff_t **difference_type**;
  const 𝕋* **pointer**;
  const 𝕋& **reference**;}

6.3.2 Reverse Iterator

Transform $[i \nearrow j) \mapsto [j-1 \searrow i-1)$.

template⟨𝕀ter⟩
class **reverse_iterator** : public iterator⟨
  iterator_traits⟨𝕀ter⟩::iterator_category,
  iterator_traits⟨𝕀ter⟩::value_type,
  iterator_traits⟨𝕀ter⟩::difference_type,
  iterator_traits⟨𝕀ter⟩::pointer,
  iterator_traits⟨𝕀ter⟩::reference⟩;

Denote
 RI = **reverse_iterator**
 𝔸𝕀 = ℝandomAccessIterator.

Abbreviate:
typedef RI<𝔸𝕀, 𝕋,
  ℝeference, 𝔻istance⟩ **self**;

// Default constructor ⇒ singular value
self::RI();

explicit *// Adaptor Constructor*
self::RI(𝔸𝕀*i*);

𝔸𝕀 self::**base**(); *// adpatee's position*

// so that: &(RI(i)) == &*(i-1)* ℝeference
self::**operator***();

self *// position to & return base()-1*
RI::**operator++**();

self& *// return old position and move*
RI::**operator++**(int); *// to base()-1*

self *// position to & return base()+1*
RI::**operator−−**();

self& *// return old position and move*
RI::**operator−−**(int); *// to base()+1*

bool *// ⇔ s0.base() == s1.base()*
operator==(const self& *s0*, const self& *s1*);

reverse_iterator Specific

self *// returned value positioned at base()-n*
reverse_iterator::**operator+**(
  𝔻istance *n*) const;

self& *// change & return position to base()-n*
reverse_iterator::**operator+=**(𝔻istance *n*);

self *// returned value positioned at base()+n*
reverse_iterator::**operator−**(
  𝔻istance *n*) const;

self& *// change & return position to base()+n*
reverse_iterator::**operator−=**(𝔻istance *n*);

ℝeference *// *(*this + n)*
reverse_iterator::**operator[]**(𝔻istance *n*);

𝔻istance *// r0.base() - r1.base()*
operator−(const self& *r0*, const self& *r1*);

self *// n + r.base()*
operator−(𝔻istance *n*, const self& *r*);

bool *// r0.base() < r1.base()*
operator<(const self& *r0*, const self& *r1*);

6.3.3 Insert Iterators

template⟨class ℂontainer⟩
class **back_insert_iterator** :
  public output_iterator;

template⟨class ℂontainer⟩
class **front_insert_iterator** :
  public output_iterator;

template⟨class ℂontainer⟩
class **insert_iterator** :
  public output_iterator;

Here 𝕋 will denote the ℂontainer::value_type.

Constructors

explicit *// ∃ ℂontainer::push_back(const 𝕋&)*
back_insert_iterator::back_insert_iterator(
  ℂontainer& *x*);

explicit *// ∃ ℂontainer::push_front(const 𝕋&)*
front_insert_iterator::front_insert_iterator(
  ℂontainer& *x*);

// ∃ ℂontainer::insert(const 𝕋&)
insert_iterator::insert_iterator(
  ℂontainer   *x*,
  ℂontainer::iterator *i*);

Denote
 InsIter = **back_insert_iterator**
 insFunc = **push_back**
 iterMaker = **back_inserter** ☞7.4
or
 InsIter = **front_insert_iterator**
 insFunc = **push_front**
 iterMaker = **front_inserter**
or
 InsIter = **insert_iterator**
 insFunc = **insert**

Member Functions & Operators

InsIter& *// calls x.insFunc(val)*
InsIter::**operator=**(const 𝕋& *val*);

InsIter& *// return *this*
InsIter::**operator***();

InsIter& *// no-op, just return *this*
InsIter::**operator++**();

InsIter& *// no-op, just return *this*
InsIter::**operator++**(int);

Template Function

InsIter *// return InsIter⟨ℂontainer⟩(x)*
iterMaker(ℂontainer& *x*);

// return insert_iterator⟨ℂontainer⟩(x, i)
insert_iterator⟨ℂontainer⟩
inserter(ℂontainer& *x*, 𝕀terator *i*);

7 示例

7.1 向量

```
// safe get
int  vi(const vector<unsigned>& v, int i) {
  return(i < (int)v.size() ? (int)v[i] : -1);
}

// safe set
void vin(vector<int>& v, unsigned i, int n) {
  int  nAdd = i - v.size() + 1;
  if (nAdd > 0) v.insert(v.end(), nAdd, n);
  else v[i] = n;
}
```

7.2 链表分割

```
void lShow(ostream& os, const list<int>& l) {
  ostream_iterator<int> osi(os, " ");
  copy(l.begin(), l.end(), osi); os << endl;
}

void lmShow(ostream& os, const char* msg,
  const list<int>& l,
  const list<int>& m) {
  os << msg << (m.size() ? ":\n" : ": ");
  lShow(os, l);
  if (m.size()) lShow(os, m);
} // lmShow

list<int>::iterator p(list<int>& l, int val) {
  return find(l.begin(), l.end(), val);
}
```

```
  static int prim[] = { 2, 3, 5, 7 };
  static int perf[] = { 6, 28, 496 };
  const list<int> lPrimes(prim + 0, prim + 4);
  const list<int> lPerfects(perf + 0, perf + 3);
  list<int> l(lPrimes), m(lPerfects);
  lmShow(cout, "primes & perfects", l, m);
  l.splice(l.begin(), m);
  lmShow(cout, "splice(l.beg, m)", l, m);
  l = lPrimes; m = lPerfects;
  l.splice(l.begin(), m, p(m, 28));
  lmShow(cout, "splice(l.beg, m, ^28)", l, m);
  m.erase(m.begin(), m.end()); // <=>m.clear()
  l = lPrimes;
  l.splice(p(l, 3), l, p(l, 5));
  lmShow(cout, "5 before 3", l, m);
  l = lPrimes;
  l.splice(l.begin(), l, p(l, 7), l.end());
  lmShow(cout, "tail to head", l, m);
  l = lPrimes;
  l.splice(l.end(), l, l.begin(), p(l, 3));
  lmShow(cout, "head to tail", l, m);
```

➠

```
primes & perfects:
2 3 5 7
6 28 496
splice(l.beg, m): 6 28 496 2 3 5 7
splice(l.beg, m, ^28):
28 2 3 5 7
6 496
5 before 3: 2 5 3 7
tail to head: 7 2 3 5
head to tail: 3 5 7 2
```

7.3 比较对象排序

```
class ModN {
public:
  ModN(unsigned m) : _m(m) {}
  bool operator ()(const unsigned& u0,
    const unsigned& u1) {
    return ((u0 % _m) < (u1 % _m));
  }
private: unsigned _m;
}; // ModN
```

```
  ostream_iterator<unsigned> oi(cout, " ");
  unsigned  q[6];
  for (int n = 6, i = n - 1; i >= 0; n = i--)
    q[i] = n*n*n*n;
  cout << "four-powers:   ";
  copy(q + 0, q + 6, oi);
  for (unsigned b = 10; b <= 1000; b *= 10) {
    vector<unsigned>  sq(q + 0, q + 6);
    sort(sq.begin(), sq.end(), ModN(b));
    cout << endl << "sort mod " << setw(4) << b << ":
    copy(sq.begin(), sq.end(), oi);
  } cout << endl;
```

➠

```
four-powers:   1 16 81 256 625 1296
sort mod   10: 1 81 625 16 256 1296
sort mod  100: 1 16 625 256 81 1296
sort mod 1000: 1 16 81 256 1296 625
```

7.4 流迭代器

```
void unitRoots(int n) {
  cout << "unit " << n << "-roots:" << endl;
  vector<complex<float> > roots;
  float  arg = 2.*M_PI / (float)n;
  complex<float> r, r1 = polar((float)1., arg);
  for (r = r1; --n; r *= r1)
    roots.push_back(r);
  copy(roots.begin(), roots.end(),
    ostream_iterator<complex<float> >(cout,
    "\n"));
} // unitRoots
```

```
  {ofstream o("primes.txt"); o << "2 3 5"; }
  ifstream pream("primes.txt");
  vector<int> p;
  istream_iterator<int>  priter(pream);
  istream_iterator<int>  eosi;
  copy(priter, eosi, back_inserter(p));
  for_each(p.begin(), p.end(), unitRoots);
```

✇➠

```
unit 2-roots:
(-1.000,-0.000)
unit 3-roots:
(-0.500,0.866)
(-0.500,-0.866)
unit 5-roots:
(0.309,0.951)
(-0.809,0.588)
(-0.809,-0.588)
(0.309,-0.951)
```

7.5 二分查找

```
  // first 5 Fibonacci
  static int fb5[] = { 1, 1, 2, 3, 5 };
  for (int n = 0; n <= 6; ++n) {
    pair<int*, int*> p =
      equal_range(fb5, fb5 + 5, n);
    cout << n << ":[" << p.first - fb5 << ','
      << p.second - fb5 << ") ";
    if (n == 3 || n == 6) cout << endl;
  }
```

✇➠

```
0:[0,0) 1:[0,2) 2:[2,3) 3:[3,4)
4:[4,4) 5:[4,5) 6:[5,5)
```

7.6 转换 & 计算

```
template <class T>
class AbsPwr : public unary_function < T, T > {
public:
  AbsPwr(T p) : _p(p) {}
  T operator()(const T& x) const {
    return pow(fabs(x), _p);
  }
private: T _p;
}; // AbsPwr

template<typename InpIter> float
normNP(InpIter xb, InpIter xe, float p) {
  vector<float>  vf;
  transform(xb, xe, back_inserter(vf),
    AbsPwr<float>(p > 0. ? p : 1.));
  return((p > 0.)
    ? pow(accumulate(vf.begin(), vf.end(), 0.),
    1. / p)
    : *(max_element(vf.begin(), vf.end())));
} // normNP

float distNP(const float* x, const float* y,
  unsigned n, float p) {
  vector<float>  diff;
  transform(x, x + n, y, back_inserter(diff),
    minus<float>());
  return normNP(diff.begin(), diff.end(), p);
} // distNP
```

```
  float  x3y4[] = { 3., 4., 0. };
  float  z12[] = { 0., 0., 12. };
  float  p[] = { 1., 2., M_PI, 0. };
  for (int i = 0; i < 4; ++i) {
    float d = distNP(x3y4, z12, 3, p[i]);
    cout << "d_{" << p[i] << "}=" << d << endl;
  }
```

✇➠

```
d_{1}=19
d_{2}=13
d_{3.14159}=12.1676
d_{0}=12
```

7.7 迭代和绑定

```
// self-refering int
class Interator : public
  iterator < input_iterator_tag, int, size_t > {
  int  _n;
public:
  Interator(int n = 0) : _n(n) {}
  int operator*() const { return _n; }
  Interator& operator++() {
    ++_n;  return *this;
  }
  Interator  operator++(int) {
    Interator t(*this);
    ++_n;   return t;
  }
}; // Interator
bool operator==(const Interator& i0,
  const Interator& i1) {
  return (*i0 == *i1);
}
bool operator!=(const Interator& i0,
  const Interator& i1) {
  return !(i0 == i1);
}

struct Fermat : public
  binary_function < int, int, bool > {
  Fermat(int p = 2) : n(p) {}
  int n;
  int nPower(int t) const
  { // t^n
    int i = n, tn = 1;
    while (i--) tn *= t;
    return tn;
  } // nPower
  int nRoot(int t) const {
    return (int)pow(t + .1, 1. / n);
  }
  int xNyN(int x, int y) const {
    return(nPower(x) + nPower(y));
  }
  bool operator()(int x, int y) const {
    int zn = xNyN(x, y), z = nRoot(zn);
    return(zn == nPower(z));
  }
}; // Fermat
```

```
  for (int n = 2; n <= Mp; ++n) {
    Fermat fermat(n);
    for (int x = 1; x < Mx; ++x) {
      binder1st<Fermat>
        fx = bind1st(fermat, x);
      Interator iy(x), iyEnd(My);
      while ((iy = find_if(++iy, iyEnd, fx))
        != iyEnd) {
        int  y = *iy,
          z = fermat.nRoot(fermat.xNyN(x, y));
        cout << x << '^' << n << " + "
          << y << '^' << n << " = "
          << z << '^' << n << endl;
        if (n > 2)
          cout << "Fermat is wrong!" << endl;
      }
    }
  }
```

✇➠

```
3^2 + 4^2 = 5^2
5^2 + 12^2 = 13^2
6^2 + 8^2 = 10^2
7^2 + 24^2 = 25^2
```

7.8 特征迭代

```
template <class Itr>
typename iterator_traits<Itr>::value_type
mid(Itr b, Itr e, input_iterator_tag) {
  cout << "mid(general):\n";
  Itr bm(b);  bool  next = false;
  for (; b != e; ++b, next = !next) {
    if (next) { ++bm; }
  }
  return *bm;
} // mid<input>

template <class Itr>
typename iterator_traits<Itr>::value_type
mid(Itr b, Itr e,
random_access_iterator_tag) {
  cout << "mid(random):\n";
  Itr bm = b + (e - b) / 2;
  return *bm;
} // mid<random>

template <class Itr>
typename iterator_traits<Itr>::value_type
mid(Itr b, Itr e) {
  typename
    iterator_traits<Itr>::iterator_category t;
  mid(b, e, t);
} // mid

template <class Ctr>
void fillmid(Ctr& ctr) {
  static int perfects[5] =
  { 6, 14, 496, 8128, 33550336 },
  *pb = &perfects[0];
  ctr.insert(ctr.end(), pb, pb + 5);
  int m = mid(ctr.begin(), ctr.end());
  cout << "mid=" << m << "\n";
} // fillmid
```

```
  list<int> l;  vector<int> v;
  fillmid(l);   fillmid(v);
```

✇➠

```
mid(general):
mid=0
mid(random):
mid=0
```