1 Notations

- The symbol const for `const`.
- The symbol ↶ for *function returned value*.
- Template class parameters lead by outlined character. For example: 𝕋, 𝕂ey, ℂompare. Interpreted in `template` definition context.
- Sometimes `class`, `typename` dropped.
- Template class parameters dropped, thus C sometimes used instead of C⟨𝕋⟩.
- "See example" by ☞, its output by ✇➠.

2 Containers

2.1 Pair

#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 Types

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

2.1.2 Functions & Operators

See also 2.2.3.

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

2.2 Containers — Common

Here X is any of
{**vector**, **deque**, **list**, **set**, **multiset**, **map**, **multimap**}

2.2.1 Types

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**

Iterators reference value_type (See 6).

2.2.2 Members & Operators

```
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 Comparison Operators

Let, X *v*, *w*. X may also be **pair** (2.1).

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

All done lexicographically and ↶`bool`.

2.3 Sequence Containers

S is any of {**vector**, **deque**, **list**}

2.3.1 Constructors

```
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 Members

```
S::iterator  // inserted copy
S::insert(S::iterator             before,
          const S::value_type&    val);

S::iterator  // inserted copy
S::insert(S::iterator             before,
          S::size_type            nVal,
          const S::value_type&    val);

S::iterator  // inserted copy
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,
↶ post erased        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 Vector

#include <vector>

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

See also 2.2 and 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 Deque

#include <deque>

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

Has all of **vector** functionality (see 2.4).

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

2.6 List

#include <list>

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

See also 2.2 and 2.3.

```
void  list::pop_front();

void  list::push_front(const 𝕋& x);

void  // move all x (&x ≠ this) before pos
list::splice(iterator pos, list⟨𝕋⟩& x);   ☞7.2

void  // move x's xElemPos before pos
list::splice (iterator   pos,
              list⟨𝕋⟩&   x,
              iterator   xElemPos);   ☞7.2

void  // move x's [xFirst,xLast) before pos
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);
  // after call: ∀ this iterator p, *p ≠ *(p + 1)
void list::unique();  // remove repeats
void  // as before but, ¬binPred(*p, *(p + 1))
list::unique(𝔹inaryPredicate binPred);
  // Assuming both this and x sorted
void list::merge(list⟨𝕋⟩& x);
  // merge and assume sorted by cmp
void list::merge(list⟨𝕋⟩& x, ℂompare cmp);
void list::reverse();
void list::sort();
void list::sort(ℂompare cmp);
```

2.7 Sorted Associative

Here A any of
{**set**, **multiset**, **map**, **multimap**}.

2.7.1 Types

For A=[multi]set, columns are the same

A::**key_type**	A::**value_type**
A::**keycompare**	A::**value_compare**

2.7.2 Constructors

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

2.7.3 Members

```
A::keycompare     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  // # erased
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::const_iterator  // ↶ first = k or end()
A::find(const A::key_type& k) const;
```

A::const_iterator // ↶ *first* ≥ *k*
A::**lower_bound**(const A::key_type& *k*) const ;

A::const_iterator // ↶ *first* > *k*
A::**upper_bound**(const A::key_type& *k*) const ;

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

2.8 Set

#include <set>

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

See also 2.2 and 2.7.

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

pair⟨set::iterator, bool⟩ // *bool = if new*
set::**insert**(const set::value_type& *x*);

2.9 Multiset

#include <multiset>

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

See also 2.2 and 2.7.

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

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

multiset::iterator // *inserted copy*
multiset::**insert**(const multiset::value_type& *x*);

2.10 Map

#include <map>

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

See also 2.2 and 2.7.

2.10.1 Types

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

2.10.2 Members

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

pair⟨map::iterator, bool⟩ // *bool = if new*
map::**insert**(const map::value_type& *x*);

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

map::iterator
map::**find**(const map::key_type& *k*);

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

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

pair⟨map::iterator, map::iterator⟩
map::**equal_range**(const map::key_type& *k*);

Example

```
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 Multimap

#include <multimap>

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

See also 2.2 and 2.7.

2.11.1 Types

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

2.11.2 Members

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

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

multimap::iterator multimap::**find**(
const multimap::key_type& *k*);

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

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

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

3 Container Adaptors

3.1 Stack Adaptor

#include <stack>

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

Default constructor. ℂontainer must have `empty()`, `size()`, `back()`, `push_back()`, `pop_back()`. So **vector**, **list** and **deque** can be used.

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 ;

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

Comparision Operators

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

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

3.2 Queue Adaptor

#include <queue>

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

Default constructor. ℂontainer must have `empty()`, `size()`, `back()`, `front()`, `push_back()` and `pop_front()`. So **list** and **deque** can be used.

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**();

Comparision Operators

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

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

3.3 Priority Queue

#include <queue>

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

ℂontainer must provide random access iterator and have `empty()`, `size()`, `front()`, `push_back()` and `pop_back()`. So **vector** and **deque** can be used.

Mostly implemented as *heap*.

3.3.1 Constructors

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

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

3.3.2 Members

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**();

No comparision operators.

4 Algorithms

#include <algorithm>

STL algorithms use iterator type parameters. Their *names* suggest their category (See 6.1).

For abbreviation, the clause — `template ⟨class 𝔽oo, ...⟩` is dropped. The outlined leading character can suggest the `template` context.

Note: When looking at two sequences: $S_1 = [first_1, last_1)$ and $S_2 = [first_2, ?)$ or $S_2 = [?, last_2)$ — caller is responsible that function will not overflow S_2.

4.1 Query Algorithms

𝔽unction // *f not changing [first, last)*
for_each(𝕀nputIterator *first*,
𝕀nputIterator *last*,
𝔽unction *f*); ☞7.4

𝕀nputIterator // *first i so i==last or *i==val*
find(𝕀nputIterator *first*,
𝕀nputIterator *last*,
const 𝕋 *val*); ☞7.2

𝕀nputIterator // *first i so i==last or pred(i)*
find_if(𝕀nputIterator *first*,
𝕀nputIterator *last*,
ℙredicate *pred*); ☞7.7

𝔽orwardIterator // *first duplicate*
adjacent_find(𝔽orwardIterator *first*,
𝔽orwardIterator *last*);

𝔽orwardIterator // *first binPred-duplicate*
adjacent_find(𝔽orwardIterator *first*,
𝔽orwardIterator *last*,
𝔹inaryPredicate *binPred*);

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

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

// *↶ bi-pointing to first !=*
pair⟨𝕀nputIterator1, 𝕀nputIterator2⟩
mismatch(𝕀nputIterator1 *first1*,
𝕀nputIterator1 *last1*,
𝕀nputIterator2 *first2*);

// *↶ bi-pointing to first binPred-mismatch*
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 Mutating Algorithms

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 // $\forall s_i^k \in S_k \;\; r_i = bop(s_i^1, s_i^2)$
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 // *as above but using 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**(𝕆utputIterator *first*,
𝕊ize *n*,
const 𝕋& *value*);

void // *by calling gen()*
generate(𝔽orwardIterator *first*,
𝔽orwardIterator *last*,
𝔾enerator *gen*);

void // *n calls to gen()*
generate_n(𝕆utputIterator *first*,
𝕊ize *n*,
𝔾enerator *gen*);

All variants of **remove** and **unique** return iterator to *new end* or *past last copied*.

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

𝔽orwardIterator // *as above but using pred*
remove_if(𝔽orwardIterator *first*,
𝔽orwardIterator *last*,
ℙredicate *pred*);

𝕆utputIterator // *↶ past last copied*
remove_copy(𝕀nputIterator *first*,
𝕀nputIterator *last*,
𝕆utputIterator *result*,
const 𝕋& *value*);

𝕆utputIterator // *as above but using pred*
remove_copy_if(𝕀nputIterator *first*,
𝕀nputIterator *last*,
𝕆utputIterator *result*,
ℙredicate *pred*);

All variants of **unique** template functions remove *consecutive* (*binPred*-) duplicates. Thus useful after sort (See 4.3).

𝔽orwardIterator // *[↶,last) gets repetitions*
unique(𝔽orwardIterator *first*,
𝔽orwardIterator *last*);

𝔽orwardIterator // *as above but using binPred*
unique(𝔽orwardIterator *first*,
𝔽orwardIterator *last*,
𝔹inaryPredicate *binPred*);

𝕆utputIterator // *↶ past last copied*
unique_copy(𝕀nputIterator *first*,
𝕀nputIterator *last*,
𝕆utputIterator *result*,
const 𝕋& *result*);

𝕆utputIterator // *as above but using binPred*
unique_copy(𝕀nputIterator *first*,
𝕀nputIterator *last*,
𝕆utputIterator *result*,
𝔹inaryPredicate *binPred*);

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

𝕆utputIterator // *↶ past last copied*
reverse_copy(𝔹idirectionalIterator *first*,
𝔹idirectionalIterator *last*,
𝕆utputIterator *result*);

void // *with first moved to middle*
rotate(𝔽orwardIterator *first*,
𝔽orwardIterator *middle*,
𝔽orwardIterator *last*);

𝕆utputIterator // *first to middle* position
rotate_copy(𝔽orwardIterator *first*,
𝔽orwardIterator *middle*,
𝔽orwardIterator *last*,
𝕆utputIterator *result*);

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

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

```
𝔹idirectionalIterator // begin with true
partition(𝔹idirectionalIterator  first,
          𝔹idirectionalIterator  last,
          ℙredicate              pred);
```

```
𝔹idirectionalIterator // begin with true
stable_partition(
    𝔹idirectionalIterator  first,
    𝔹idirectionalIterator  last,
    ℙredicate              pred);
```

4.3 Sort and Application

```
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) sorted,
partial_sort( // [middle,last) eq-greater
    ℝandomAccessIterator  first,
    ℝandomAccessIterator  middle,
    ℝandomAccessIterator  last);
```

```
void // as above but using comp(e_i, e_j)
partial_sort(
    ℝandomAccessIterator  first,
    ℝandomAccessIterator  middle,
    ℝandomAccessIterator  last,
    ℂompare               comp);
```

```
ℝandomAccessIterator // ↶ post last sorted
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);
```

Let $n = position - first$, **nth_element** partitions $[first, last)$ into: $L = [first, position)$, e_n, $R = [position + 1, last)$ such that $\forall l \in L, \forall r \in R \quad l \leq e_n \leq r$. $\leq$ means $\ngtr$.

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

```
void // as above but using comp(e_i, e_j)
nth_element(
    ℝandomAccessIterator  first,
    ℝandomAccessIterator  position,
    ℝandomAccessIterator  last,
    ℂompare               comp);
```

4.3.1 Binary Search

```
bool // this section assumes sorted range
binary_search(𝔽orwardIterator  first,
              𝔽orwardIterator  last,
              const 𝕋&         value);
```

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

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

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

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

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

equal_range returns iterators pair that lower_bound and upper_bound return.

```
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 Merge

Assuming $S_1 = [first_1, last_1)$ and $S_2 = [first_2, last_2)$ are sorted, stably merge them into $[result, result + N)$ where $N = |S_1| + |S_2|$.

```
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 // ranges [first,middle) [middle,last)
inplace_merge( // into [first,last)
    𝔹idirectionalIterator  first,
    𝔹idirectionalIterator  middle,
    𝔹idirectionalIterator  last);
```

```
void // as above but using comp
inplace_merge(
    𝔹idirectionalIterator  first,
    𝔹idirectionalIterator  middle,
    𝔹idirectionalIterator  last,
    ℂompare                comp);
```

4.3.3 Functions on Sets

Can work on *sorted associcative* containers (see 2.7). For **multiset** the interpretation of: *union*, *intersection* and *difference* is by: *maximum*, *minimum* and *subtraction* of occurrences respectively.

Let $S_i = [first_i, last_i)$ for $i = 1, 2$.

```
bool // S1 ⊇ S2
includes(𝕀nputIterator1  first1,
         𝕀nputIterator1  last1,
         𝕀nputIterator2  first2,
         𝕀nputIterator2  last2);
```

```
bool // as above but using comp
includes(𝕀nputIterator1  first1,
         𝕀nputIterator1  last1,
         𝕀nputIterator2  first2,
         𝕀nputIterator2  last2,
         ℂompare         comp);
```

```
𝕆utputIterator // S1 ∪ S2, ↶past end
set_union(𝕀nputIterator1  first1,
          𝕀nputIterator1  last1,
          𝕀nputIterator2  first2,
          𝕀nputIterator2  last2,
          𝕆utputIterator  result);
```

```
𝕆utputIterator // as above but using comp
set_union(𝕀nputIterator1  first1,
          𝕀nputIterator1  last1,
          𝕀nputIterator2  first2,
          𝕀nputIterator2  last2,
          𝕆utputIterator  result,
          ℂompare         comp);
```

```
𝕆utputIterator // S1 ∩ S2, ↶past end
set_intersection(𝕀nputIterator1  first1,
                 𝕀nputIterator1  last1,
                 𝕀nputIterator2  first2,
                 𝕀nputIterator2  last2,
                 𝕆utputIterator  result);
```

```
𝕆utputIterator // as above but using comp
set_intersection(𝕀nputIterator1  first1,
                 𝕀nputIterator1  last1,
                 𝕀nputIterator2  first2,
                 𝕀nputIterator2  last2,
                 𝕆utputIterator  result,
                 ℂompare         comp);
```

```
𝕆utputIterator // S1 \ S2, ↶past end
set_difference(𝕀nputIterator1  first1,
               𝕀nputIterator1  last1,
               𝕀nputIterator2  first2,
               𝕀nputIterator2  last2,
               𝕆utputIterator  result);
```

```
𝕆utputIterator // as above but using comp
set_difference(𝕀nputIterator1  first1,
               𝕀nputIterator1  last1,
               𝕀nputIterator2  first2,
               𝕀nputIterator2  last2,
               𝕆utputIterator  result,
               ℂompare         comp);
```

```
OutputIterator // S1△S2, ↶past end
set_symmetric_difference(
    InputIterator1   first1,
    InputIterator1   last1,
    InputIterator2   first2,
    InputIterator2   last2,
    OutputIterator   result);

OutputIterator // as above but using comp
set_symmetric_difference(
    InputIterator1   first1,
    InputIterator1   last1,
    InputIterator2   first2,
    InputIterator2   last2,
    OutputIterator   result,
    Compare          comp);
```

4.3.4 Heap

Maxheap—largest element is first.

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

void // as above but using comp
push_heap(RandomAccessIterator first,
          RandomAccessIterator last,
          Compare              comp);

void // first is popped
pop_heap(RandomAccessIterator first,
         RandomAccessIterator last);

void // as above but using comp
pop_heap(RandomAccessIterator first,
         RandomAccessIterator last,
         Compare              comp);

void // [first,last) arbitrary ordered
make_heap(RandomAccessIterator first,
          RandomAccessIterator last);

void // as above but using comp
make_heap(RandomAccessIterator first,
          RandomAccessIterator last,
          Compare              comp);

void // sort the [first,last) heap
sort_heap(RandomAccessIterator first,
          RandomAccessIterator last);

void // as above but using comp
sort_heap(RandomAccessIterator first,
          RandomAccessIterator last,
          Compare              comp);
```

4.3.5 Min and Max

```
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 Permutations

To get all permutations, start with ascending sequence end with descending.

```
bool // ↶ iff available
next_permutation(
    BidirectionalIterator first,
    BidirectionalIterator last);

bool // as above but using comp
next_permutation(
    BidirectionalIterator first,
    BidirectionalIterator last,
    Compare               comp);

bool // ↶ iff available
prev_permutation(
    BidirectionalIterator first,
    BidirectionalIterator last);

bool // as above but using comp
prev_permutation(
    BidirectionalIterator first,
    BidirectionalIterator last,
    Compare               comp);
```

4.3.7 Lexicographic Order

```
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 Computational

#include <numeric>

T // $\sum_{[first,last)}$ ☞7.6

```
accumulate(InputIterator first,
           InputIterator last,
           T             initVal);

T // as above but using binop
accumulate(InputIterator   first,
           InputIterator   last,
           T               initVal,
           BinaryOperation binop);
```

T // $\sum_i e_i^1 \times e_i^2$ for $e_i^k \in S_k, (k = 1, 2)$

```
inner_product(InputIterator1 first1,
              InputIterator1 last1,
              InputIterator2 first2,
              T              initVal);
```

T // Similar, using $\sum^{(sum)}$ and $\times_{mult}$

```
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 // as above but using binop
partial_sum(
    InputIterator   first,
    InputIterator   last,
    OutputIterator  result,
    BinaryOperation binop);
```

OutputIterator // $r_k = s_k - s_{k-1}$ for $k > 0$

adjacent_difference(// $r_0 = s_0$

```
    InputIterator  first,
    InputIterator  last,
    OutputIterator result);

OutputIterator // as above but using binop
adjacent_difference(
    InputIterator   first,
    InputIterator   last,
    OutputIterator  result,
    BinaryOperation binop);
```

5 Function Objects

#include <functional>

```
template<class Arg, class Result>
struct unary_function {
  typedef Arg argument_type;
  typedef Result result_type;}
```

Derived unary objects:

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;}
```

Following derived template objects accept two operands. Result obvious by the name.

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 Function Adaptors

5.1.1 Negators

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 // *negate pred*
binary_negate::**operator()**(
  ℙredicate::first_argument_type *x*
  ℙredicate::second_argument_type *y*);

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

5.1.2 Binders

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 from 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 from 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 Pointers to Functions

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 Iterators

#include <**iterator**>

6.1 Iterators Categories

Here, we will use:

- `X` iterator type.
- `a, b` iterator values.
- `r` iterator reference (`X& r`).
- `t` a value type `T`.

Imposed by empty `struct` tags.

6.1.1 Input, Output, Forward

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

In table follows requirements check list for **I**nput, **O**utput and **F**orward iterators.

Expression; Requirements		I	O	F
`X()` `X u`	might be singular			•
`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* $\neq 0$ *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 Typedefs & Adaptors

```
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**(); // *adaptee'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 Examples

7.1 Vector

```
// 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 List Splice

```
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 Compare Object Sort

```
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 Stream Iterators

```
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 Binary Search

```
 // 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 Transform & Numeric

```
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 Iterator and Binder

```
// 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 Iterator Traits

```
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=496
mid(random):
mid=496
```