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.
- Template class parameters dropped, thus C sometimes used instead of C<𝕋>.
- A "See example" note by ☞. Example 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;

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

void X::**erase**(X::iterator *elemPosition*);
void X::**erase**(X::const_iterator *first*,
X::const_iterator *last*);

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*);

void S::**push_back**(const 𝕋& 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` $\neq$ `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:* $\forall$ `this` iterator p, $*p \neq *(p+1)$
void list::**unique**(𝔹inaryPredicate *binPred*);
void // *as before but,* $\neg binPred(*p, *(p+1))$
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 is 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::**key_compare**	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::key_compare A::**key_comp**();
A::value_compare A::**value_comp**();
A::iterator A::**insert**(A::iterator *hint*,
A::valueType *val*);

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

A::size_type // *# erased*
A::**erase**(A::key_type k);
void A::**erase**(A::iterator p);
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> // 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.h>

```
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::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;
```

Example

```
typedef map<string, int, less<string> > 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.h>

```
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::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 Container Adaptors

3.1 Stack Adaptor

#include <stack>

```
template<class Container>
class stack;
```

Default constructor. Container must have `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>
#include <stack> // *Gnu*

```
template<class Container>
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>
#include <stack> // *Gnu*

```
template<class Container,
         class Compare>
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  begin,
    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 Non Mutating 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);

// [first2, last2) ⊑ [first1, last1)
ForwardIterator1
search(ForwardIterator1 first1,
       ForwardIterator1 last1,
       ForwardIterator2 first2,
       ForwardIterator2 last2);

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

4.2 Mutating Algorithms

```
OutputIterator // ↶ first2 + (last1 − first1)
copy(InputIterator  first1,
     InputIterator  last1,
     OutputIterator first2);

// ↶ last2 − (last1 − first1)
BidirectionalIterator2
copy_backward(
    BidirectionalIterator1 first1,
    BidirectionalIterator1 last1,
    BidirectionalIterator2 last2);

void swap(T& x, T& y);

ForwardIterator2 // ↶ first2 + #[first1, last1)
swap_ranges(ForwardIterator1 first1,
            ForwardIterator1 last1,
            ForwardIterator2 first2);

OutputIterator // ↶ result + (last1 − first1)
transform(InputIterator  first,
          InputIterator  last,
          OutputIterator result,
          UnaryOperation op);    ☞7.6

OutputIterator // ∀ s_i^k ∈ 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 // ↶ result2 + #[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(ForwardIterator first,
            Size            n,
            const T&        value);

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

void // n calls to gen()
generate_n(ForwardIterator first,
           Size            n,
           Generator       gen);
```

All variants of **remove** and **unique** template functions 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 usefull 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 // asbutusing 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 *result*);

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 *resulLast*,
Compare *comp*);

Let $n = position - first$, **nth_element** partitions $[first, last)$ into: $L = [first, position)$, e_n, $R = [position + 1, last)$ such that $\forall l \in L, \forall r \in R \quad l \not> e_n \le r$.

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
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 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 *substraction* of occurrences respectably.

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

bool // $S_1 \supseteq S_2$
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 // $S_1 \cup S_2$, ↶*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 // $S_1 \cap S_2$, ↶*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 // $S_1 \setminus S_2$, ↶*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 // $S_1 \triangle S_2$, ⌒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

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 **times**<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`.

6.1.1 Input, Output, Forward

```
template<class 𝕋, class 𝔻istance>
class input_iterator;
```

```
class output_iterator;
```

```
template<class 𝕋, class 𝔻istance>
class forward_iterator;
```

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

```
template<class 𝕋, class 𝔻istance>
class bidirectional_iterator;
```

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

```
template<class 𝕋, class 𝔻istance>
class random_access_iterator;
```

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 :
      input_iterator<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 output_iterator<𝕋>;
```

```
// 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 Adaptors Iterators

6.3.1 Reverse Iterators

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

```
template<class BidirectionalIterator,
            class T, class Reference= &T,
            class Distance = ptrdiff_t>
class
 reverse_bidirectional_iterator :
  public
   bidirectional_iterator<T, Distance>;
```

```
template<class RandomAccessIterator,
            class T, class Reference= &T,
            class Distance = ptrdiff_t>
class
 reverse_iterator :
  public
   random_access_iterator<T, Distance>;
```

Denote
  RI = **reverse_bidirectional_iterator**,
  AI = BidirectionalIterator,
or
  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*);

6.3.2 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); cout<<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<float_complex> roots;
 float  arg = 2.*M_PI/(float)n;
 float_complex  r, r1 = polar((float)1., arg);
 for (r = r1; --n;  r *= r1)
   roots.push_back(r);
 copy(roots.begin(), roots.end(),
      ostream_iterator<float_complex>(cout,
                                       "\n"));
} // unitRoots

 {ofstream("primes.txt") << "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

```
void eqrange_example() {
 // 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%4)==3) cout << endl;
 } cout << endl;
} // eqrange_example
```

☞

```
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

float normNP(const float* xb,
       const float* 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 {
  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); }

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 <=4; 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
```