1 Notations

- The symbol ↶ for *function returned value*.
- Template class parameters lead by outlined character. For example: 𝕋, 𝕂ey, ℂmp. Interpreted in `template` definition context.
- Sometimes `class`, `typename` dropped.
- Template class parameters dropped, thus C sometimes used instead of C⟨𝕋⟩.
- Allocator parameters are omitted for simplicity.

2 Containers

2.1 Pair⟨𝕋1, 𝕋2⟩

#include <utility>

struct **pair** {
    𝕋1 **first**; 𝕋2 **second**;
    pair();
    pair(const 𝕋1& a, const 𝕋2& 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.

2.2.2 Members & Operators

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

X::iterator X::**begin**();
X::iterator X::**end**();
X::reverse_iterator X::**rbegin**();
X::reverse_iterator X::**rend**();

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 v, w be of type X. 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*);

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*,
↶ *past erased* S::const_iterator *last*);

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

void S::**pop_back**();

S::reference S::**front**();

S::reference S::**back**();

2.4 Vector⟨𝕋⟩

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*);

2.5 Deque⟨𝕋⟩

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

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

2.6 List⟨𝕋⟩

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*);
void // *move x's xElemPos before pos*
list::**splice** (iterator *pos*, list⟨𝕋⟩& x,
    iterator *xElemPos*);
void // *move x's [xFirst,xLast) before pos*
list::**splice** (iterator *pos*, list⟨𝕋⟩& x,
    iterator *xFirst*, iterator *xLast*);
void list::**remove**(const 𝕋& *value*);
void list::**remove_if**(ℙred *pred*);
// *after call:* ∀ `this` *iterator* $p, *p \neq *(p+1)$
void list::**unique**(); // *remove repeats*
void // *as before but,* ¬*binPred*$(*p, *(p+1))$
list::**unique**(𝔹inPred *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*, ℂmp *cmp*);
void list::**reverse**();
void list::**sort**();
void list::**sort**(ℂmp *cmp*);

2.7 Sorted Associative

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(ℂmp *c*=ℂmp())
A::A(A::const_iterator *first*, A::const_iterator *last*,
    ℂmp *c*=ℂmp());

2.7.3 Members

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 // # *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::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 3.3.1*
A::**equal_range**(const A::key_type& *k*) const;

2.8 Set⟨𝕂ey, ℂmp=less⟨𝕂ey⟩⟩

See also 2.2 and 2.7.

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

2.9 Map⟨𝕂ey, 𝕋, ℂmp=less⟨𝕂ey⟩⟩

See also 2.2 and 2.7.

2.9.1 Types

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

2.9.2 Members

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

3 Algorithms

#include <algorithm>

STL algorithms use iterator type parameters and function objects. Their *names* suggest their category:

- InIter – input iterator
- OutIter – output iterator
- FwIter – forward iterator
- BiIter – bidirectional iterator
- RndIter – random access iterator
- Pred – unary predicate
- BinPred – binary predicate
- UnOp – unary operation
- Cmp – comparation

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.

3.1 Non-modifying Algorithms

```
Function // f not changing [first, last)
for_each(InIter first, InIter last, Function f);

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

InIter // first i so i==last or pred(i)
find_if(InIter first, InIter last, Pred pred);

FwIter // first duplicate
adjacent_find( FwIter first, FwIter last);

FwIter // first binPred-duplicate
adjacent_find( FwIter first, FwIter last,
               BinPred binPred);

Size // ↶# equal val
count( FwIter first, FwIter last,
       const T val);

Size // ↶# satisfying pred
count_if( FwIter first, FwIter last,
          Pred pred);

// ↶ bi-pointing to first !=
pair⟨InIter1, InIter2⟩
mismatch( InIter1 first1, InIter1 last1,
          InIter2 first2);

// ↶ bi-pointing to first binPred-mismatch
pair⟨InIter1, InIter2⟩
mismatch( InIter1 first1, InIter1 last1,
          InIter2 first2, BinPred binPred);

bool
bool equal( InIter1 first1, InIter1 last1,
            InIter2 first2);

bool
equal( InIter1 first1, InIter1 last1,
       InIter2 first2, BinPred binPred);
```

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

```
FwIter1
search( FwIter1 first1, FwIter1 last1,
        FwIter2 first2, FwIter2 last2);
```

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

```
FwIter1
search( FwIter1 first1, FwIter1 last1,
        FwIter2 first2, FwIter2 last2,
        BinPred binPred);
```

3.2 Modifying Algorithms

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

```
copy( InIter first1, InIter last1,
      OutIter first2);
```

BiIter2 // ↶ $last_2 - (last_1 - first_1)$

```
copy_backward( BiIter1 first1,
               BiIter1 last1,
               BiIter2 last2);

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

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

```
transform( InIter first, InIter last,
           OutIter result, UnOp op);

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

void
replace_if( FwIter first, FwIter last,
            Pred& pred, const T& newVal);

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

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

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

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

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

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

FwIter // as above but using pred
remove_if( FwIter first, FwIter last,
           Pred pred);
```

All variants of **unique** template functions remove *consecutive* (*binPred*-) duplicates. Thus usefull after sort (See 3.3).

```
FwIter // [↶,last) gets repetitions
unique( FwIter first, FwIter last);

FwIter // as above but using binPred
unique( FwIter first, FwIter last,
        BinPred binPred);

void reverse( BiIter first, BiIter last);

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

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

void // rand() returns double in $[0, 1)$

```
random_shuffle(
    RndIter first, RndIter last,
    RandomGenerator rand);

BiIter // begin with true
partition( BiIter first, BiIter last,
           Pred pred);

BiIter // begin with true
stable_partition(
    BiIter first, BiIter last, Pred pred);
```

3.3 Sort and Application

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

void sort(RndIter first, RndIter last,
          Cmp comp);

void stable_sort(RndIter first, RndIter last);

void stable_sort(RndIter first, RndIter last,
                 Cmp comp);
```

3.3.1 Binary Search

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

bool
binary_search(FwIter first, FwIter last,
              const T& value, Cmp comp);

FwIter
lower_bound(FwIter first, FwIter last,
            const T& value);

FwIter
lower_bound(FwIter first, FwIter last,
            const T& value, Cmp comp);

FwIter
upper_bound(FwIter first, FwIter last,
            const T& value);

FwIter
upper_bound(FwIter first, FwIter last,
            const T& value, Cmp comp);
```

equal_range returns iterators pair that lower_bound and upper_bound return.

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

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

3.3.2 Min and Max

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

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

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

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

FwIter
min_element( FwIter first, FwIter last);

FwIter
min_element( FwIter first, FwIter last,
             Cmp comp);

FwIter
max_element( FwIter first, FwIter last);

FwIter
max_element( FwIter first, FwIter last,
             Cmp comp);
```

3.3.3 Permutations

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

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

bool // as above but using comp
next_permutation(
    BiIter first, BiIter last, Cmp comp);

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

bool // as above but using comp
prev_permutation(
    BiIter first, BiIter last, Cmp comp);
```