Object-Oriented Programming for Scientific Computing

STL Iterators and Algorithms

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

16. Juni 2015

Motivation for Iterators

- How does one access the entries of an associative container, for example a `set`?
- How does one write an algorithm that works for all types of STL containers?
- This requires a general method to iterate over the elements of a container.
- It would be best if this would as well work for traditional C arrays.
- It should always be possible to use the special capabilities of a container (such as random access for a `vector`).

Iterators

An iterator

- is an object of a class that allows iterating over the elements in a container (container and iterator do not have the same class).
- is `Assignable`, `DefaultConstructible` and `EqualityComparable`.
- is pointing at a specific position in a container object.
- The next element of the container object can be reached using the `operator++` of the iterator.

Example of Iterators

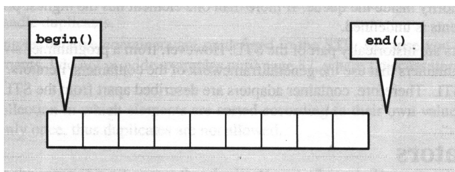

Abbildung: Iterator over a container

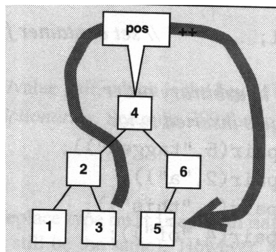

Abbildung: Iterator over a set

Iterators for Containers

- Each container defines the type of its iterator objects with a `typedef`:
  - `Container::iterator`: an iterator with read/write permission
  - `Container::const_iterator`: a readonly iterator
- In addition, each container has the following methods:
  - `begin()` returns an iterator pointing to the first element of the container object.
  - `end()` provides an iterator that points to the end of the container, i.e. one element after the last element of the container.
- For empty containers `begin()==end()` holds.

First Iterator Example: Header File

```
#include<iostream>

template<class T>
void print(const T& container)
{
    for(typename T::const_iterator i=container.begin();
                                   i!=container.end(); ++i)
        std::cout << *i << " ";
    std::cout << std::endl;
}

template<class T>
void push_back_a_to_z(T& container)
{
    for(char c='a'; c <='z'; ++c)
        container.push_back(c);
}
```

First Iterator Example: Source File

```
#include"iterator1.hh"
#include<list>
#include<vector>

int main(int argc, char** argv)
{
    std::list<char> listContainer;
    push_back_a_to_z(listContainer);
    print(listContainer);

    std::vector<int> vectorContainer;
    push_back_a_to_z(vectorContainer);
    print(vectorContainer);
}
```

Output:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113
    114 115 116 117 118 119 120 121 122
```

Iterator Concepts

- Iterators can have additional properties.
- These depend on the specific properties of the container.
- Thus it is possible to write more efficient algorithms for containers which have additional capabilities.
- Iterators can be grouped according to their capabilities.

Iterator	Capability
Input iterator	read forwards
Output iterator	write forwards
Forward iterator	iterate forwards
Bidirectional iterator	read and write backwards and forwards
Random access iterator	Read and write at arbitrary positions

Tabelle: Predefined Iterators

Iterator Concepts

Abbildung: Iterator concepts

Trivial Iterator

- A `Trivial Iterator` is an object that points to another object and can be dereferenced like a pointer. There is no guarantee that arithmetic operations are possible.
- Associated types: `value_type` is the type of the object pointed to by the iterator.
- Methods:

`ITERTYPE()`	Default constructor
`operator*()`	Dereferencing
`*i=t`	If the iterator `i` is not `const`, then assignment is possible.
`operator->()`	Access to methods and attributes of the referenced object

- Complexity guarantees: All operations have amortized constant complexity.

Input Iterator

- An `Input Iterator` is an object that points to another object, can be dereferenced like a pointer, and can be incremented to get an iterator to the next object.
- Associated types: `difference_type`: type to save the distance between two iterators
- Methods:

Expression	Effect
`x=*i`	Dereferences
`++i`	Takes a step forward
`(void)i++`	Takes a step forward, identical to `++i`.
`*i++`	identical to `T t=*i;++i;return t;`.

- Complexity guarantees: All operations have amortized constant complexity.

Output Iterator

- An `Output Iterator` is an object that can be written to and which can be incremented.
- `Output Iterators` are not comparable and do not have to define `value_type` and `difference_type`.
- May be compared to a continuous paper printer.
- Increment and assignment must alternate. The sequence has to start with an assignment, then an increment follows, then an assignment, and so forth.
- Methods:

Expression	Effect
`ITERTYPE(i)`	Copy constructor
`*i=value`	Writes a value to the location to which the iterator points
`++i`	Takes a step forward
`i++`	Takes a step forward, identical to `++i`.

- Complexity guarantees: All operations have amortized constant complexity.

Forward Iterator

- A `Forward Iterator` corresponds to the common notion of a linear sequence of values. With a forward iterator (as opposed to an `Output Iterator`) multiple passes through a container are possible.
- Defines no further methods in comparison to the `Input Iterator`.
- Incrementing does not invalidate earlier copies of the iterator.
- A forward iterator is no output iterator because `++i` does not always point to a writable location, e.g. if `i==end()`.
- Complexity guarantees: All operations have amortized constant complexity.
- Assurances: For two iterators `i` and `j` the following holds: if `i == j` then `++i == ++j`

Bidirectional Iterator

- Can be used forwards and backwards.
- Iteration of `list`, `set`, `multiset`, `map` and `multimap`
- Additional methods:

`--i`	Takes a step backwards
`i--`	Takes a step backwards

- Complexity guarantees: All operations have amortized constant complexity.
- Assurances: If `i` points to an element in the container, then `++i;--i;` and `--i;++i;` are null operations.

Random Access Iterator

- A Random Access Iterator is a bidirectional iterator that provides additional methods in order to make steps of any size both forwards and backwards in constant time. It allows in principle all of the possible pointer operations.
- Provided by vector, deque, string and array.
- Additional methods:

i+n	Returns an iterator to the nth next element
i-n	Returns an iterator to the nth previous element
i+=n	moves n elements forwards
i-=n	moves n elements backwards
i[n]	equivalent to *(i+n)
i-j	Returns the distance betweeni and j

- Complexity guarantees: All operations have amortized constant complexity.

Random Access Iterator

- Assurances:
  - If `i+n` is defined, then `i+=n;i-=n;` and `i-=n;i+=n;` are null operations
  - If `i-j` is defined, then `i == j + (i-j)` is true.
  - If `i` can be reached from `j` by a sequence of increments or decrements, then `i-j >= 0`.
  - Two iterators are `Comparable`.

Example for Vector

```
#include <iostream>
#include <vector>
#include <string>

int main()
{
    std::vector<double> a(7);
    std::cout << a.size() << std::endl;
    for (int i=0;i<a.size();++i)
        a[i] = i*0.1;
    double d = 4 * a[2];
    std::vector<double> c(a);
    std::cout << a.back() << " " << c.back() << std::endl;
    std::vector<std::string> b;
    b.resize(3);
    typedef std::vector<std::string>::reverse_iterator VectorRevIt;
    for (VectorRevIt i=b.rbegin();i!=b.rend();++i)
        std::cin >> *i;
    b.resize(4);
    b[3] = "blub";
    b.push_back("blob");
    typedef std::vector<std::string>::iterator VectorIt;
    for (VectorIt i=b.begin();i<b.end();++i)
        std::cout << *i << std::endl;
}
```

Example for List

```
#include <iostream>
#include <list>

int main()
{
    std::list<double> vals;
    for (int i=0;i<7;++i)
        vals.push_back(i*0.1);
    vals.push_front(-1);
    std::list<double> copy(vals);
    typedef std::list<double>::iterator ListIt;
    for (ListIt i=vals.begin();i!=vals.end();++i,++i)
        i=vals.insert(i,*i+0.05);
    std::cout << "vals size: " << vals.size() << std::endl;
    for (ListIt i=vals.begin();i!=vals.end();i=vals.erase(i))
        std::cout << *i << " ";
    std::cout << std::endl << "vals size: " << vals.size() << std::endl;
    typedef std::list<double>::reverse_iterator ListRevIt;
    for (ListRevIt i=copy.rbegin();i!=copy.rend();++i)
        std::cout << *i << " ";
    copy.clear();
    std::cout << std::endl << "copy size: " << copy.size() << std::endl;
}
```

Example for Set: Storage for Global Optimization

```
#include<vector>
#include<set>

class Result
{
    double residual_;
    std::vector<double> parameter_;
  public:
    bool operator<(const Result& other) const
    {
        if (other.residual_<=residual_)
            return false;
        else
            return true;
    }
    double Residual() const
    {
        return residual_;
    }
    Result(double res) : residual_(res)
    {};
};
```

Example for Set: Storage for Global Optimization

```
#include<iostream>
#include<set>
#include"result.h"

int main()
{
    std::multiset<Result> valsMSet;
    for (int i=0;i<7;++i)
        valsMSet.insert(Result(i*0.1));
    for (int i=0;i<7;++i)
        valsMSet.insert(Result(i*0.2));
    typedef std::multiset<Result>::iterator MultiSetIt;
    for (MultiSetIt i=valsMSet.begin();i!=valsMSet.end();++i)
        std::cout << i->Residual() << " ";
    std::cout << std::endl << "valsMSet size: " << valsMSet.size() <<
         std::endl;
    std::set<Result> vals(valsMSet.begin(),valsMSet.end());
    typedef std::set<Result>::iterator SetIt;
    for (SetIt i=vals.begin();i!=vals.end();++i)
        std::cout << i->Residual() << " ";
    std::cout << std::endl << "vals size: " << vals.size() << std::endl;
}
```

Example Set: Memory for Global Optimization

Output:

```
0 0 0.1 0.2 0.2 0.3 0.4 0.4 0.5 0.6 0.6 0.8 1 1.2
valsMSet size: 14
0 0.1 0.2 0.3 0.4 0.5 0.6 0.8 1 1.2
vals size: 10
```

Example for Map: Parameter Management

```
#include <iostream>
#include <map>

template<typename T>
bool GetValue(const std::map<std::string,T>& container,std::string
    key, T& value)
{
    typename std::map<std::string,T>::const_iterator
        element=container.find(key);
    if (element!=container.end())
    {
        value=element->second;
        return(true);
    }
    else
        return(false);
}
```

Example for Map: Parameter Management

```
template<typename T>
T GetValue(const std::map<std::string,T>& container,std::string
    key,bool abort=true,T defValue=T())
{
    typename std::map<std::string,T>::const_iterator
        element=container.find(key);
    if (element!=container.end())
        return(element->second);
    else
    {
        if (abort)
        {
            std::cerr << "GetValue:␣key␣\"" << key << "\"␣not␣
                found";
            std::cerr << std::endl << std::endl << "Available␣
                keys:␣" << std::endl;
            for(element=container.begin();element!=container.end();++e
                std::cerr << element->first << std::endl;
            throw "No␣Value␣found";
        }
    }
    return(defValue);
}
```

STL Algorithms

- The STL defines many algorithms that can be applied to the objects of containers, for example: search, sort, copy, ...
- These are global functions, not methods of the containers.
- Iterators are used for input and output.
- The header `algorithm` must be included.
- In the header `numeric` there are algorithms that perform calculations.

Example

```
#include<vector>
#include<iostream>
#include<algorithm>

template<typename T>
void print (const T &elem)
{
  std::cout << elem << " ";
}

int add(int &elem)
{
  elem+=5;
}

int main()
{
  std::vector<int> coll(7,3);
  std::for_each(coll.begin(), coll.end(), print<int>);
  std::cout << std::endl;
  std::for_each(coll.begin(), coll.end(), add);
  std::for_each(coll.begin(), coll.end(), print<int>);
  std::cout << std::endl;
  std::cout << std::endl;
}
```

Iterator Ranges

- All algorithms operate on one or more sets of elements, which is/are limited by iterators. This is also called a range.
- A range is bounded by the iterators: `[begin,end)`. `begin` points to the first element and `end` to the first element after the last.
- This can also be a subset of a container.
- The user is responsible for ensuring that this is a valid / meaningful set, i.e. that one gets from `begin` to `end` when iterating over the elements.
- For algorithms that expect more than one iterator range, the end is only given for the first range. For all others it is assumed that they (can) contain the same number of elements:

```
std::copy(coll1.begin(), coll1.end(), coll2.begin())
```

Algorithms with Suffix

Sometimes there are additional versions of an algorithm, which are characterized by a suffix. This makes it easier for the compiler and the programmer to distinguish the different versions.

_if suffix

- The suffix _if is added when two versions of an algorithm exist that do not differ in the number of arguments, but in their meaning.
- In the version without a suffix the last argument is a value for comparison with the elements.
- The version with suffix `_if` expects a predicate, i.e. a function that returns `bool` (see below) as a parameter. This is evaluated for all the elements.
- Not all algorithms have a version with `_if` suffix, e.g. if the number of arguments for the different versions differs.
- Example: `find` and `find_if`.

Algorithms with Suffix

_copy suffix

- Without suffix the content of each element is changed, with the suffix elements are copied and then modified.
- This version of the algorithm always has an additional argument (an iterator for the location of the copy).
- Example: `reverse` and `reverse_copy`.

This can also be combined, e.g. `_copy_if`

for_each

The easiest and most common algorithm is most likely `for_each(b,e,f)`. Here the functor `f(x)` is called for each element in the range `[b:e)`. Since references can be passed to f, it is possible to change values in the range.

Algorithms that don't Change the Content

`count(b,e,v)`	Number of elements in range `[b:e)` which are the same as `v`	integer
`count_if(b,e,v,f)`	Number of elements in range `[b:e)` for which `f(*p)` is true	integer
`all_of(b,e,f)`	True if condition `f(*p)` true for all elements in range `[b:e)`	bool
`any_of(b,e,f)`	True if condition `f(*p)` true for at least one element in range `[b:e)`	bool
`none_of(b,e,f)`	True if condition `f(*p)` true for none of the elements in range `[b:e)`	bool
`min_element(b,e)`	Smallest element in range `[b:e)`	iterator
`min_element(b,e,f)`	Smallest element in range `[b:e)`	iterator
`max_element(b,e)`	Largest element in range `[b:e)`	iterator
`max_element(b,e,f)`	Largest element in range `[b:e)`	iterator
`minmax_element(b,e)`	Iterators to smallest and largest element in range `[b:e)`	pair(min,max)
`minmax_element(b,e,f)`	Iterators to smallest and largest element in range `[b:e)`	pair(min,max)

Search Algorithms

`find(b,e,v)`	First item in range `[b:e)` with value `v`, or `e`	iterator
`find_if(b,e,f)`	First item in range `[b:e)` for which `f(*p)` is true, or `e`	iterator
`find_if_not(b,e,f)`	First item in range `[b:e)` for which `f(*p)` is false, or `e`	iterator
`find_first_of(b,e,b2,e2)`	First item in range `[b:e)` for which element from range `[b2:e2)` is the same, or `e`	iterator
`find_first_of(b,e,b2,e2,f)`	First element `p` in range `[b:e)` for which `f(*p,*q)` is true for element `q` from range `[b2:e2)`, or `e`	iterator
`find_end(b,e,b2,e2)`	Last element in range `[b:e)` that is the same as element from range `[b2:e2)`, or `e`	iterator
`find_end_of(b,e,b2,e2,f)`	Last element `p` in range `[b:e)` for which `f(*p,*q)` is true for element `q` from range `[b2:e2)`, or `e`	iterator

Search Algorithms

`adjacent_find(b,e)`	First element that is equal to its neighbor in range `[b:e)`, or `e`	iterator
`adjacent_find(b,e,f)`	First element `p` for which `f(*p,*(p+1))` is true, or `e`	iterator
`search(b,e,b2,e2)`	First element in `[b:e)`, so that the next `b2-e2` elements are like those of the range `[b2:e2)`, or `e`	iterator
`search(b,e,b2,e2,f)`	First element in `[b:e)`, so that `f(*p,*q)` is true for the next `b2-e2` elements of range `[b2:e2)`, or `e`	iterator
`search_n(b,e,n,v)`	First element, so that this and the following `n-1` elements are equal to `v`, or `e`	iterator
`search_n(b,e,n,v,f)`	First element, so that `f(*p,v)` is true for this and the following `n-1` elements, or `e`	iterator

Comparison Algorithms

`equal(b,e,b2)`	True if all elements of the two ranges `[b:e]` and `[b2:b2+(b-e))` are equal	`bool`
`equal(b,e,b2,f)`	True if `f(*p,*q)` is true for all elements of the two ranges	`bool`
`mismatch(b,e,b2)`	First elements in `[b:e)` and `[b2:b2+(b-e))` which aren't equal, or twice `e`	`pair<iterator>`
`mismatch(b,e,b2,f)`	First elements in `[b:e)` and `[b2:b2+(b-e))` for which `f(*p,*q)` is false, or twice `e`	`pair<iterator>`
`lexicographical_compare(b,e,b2,e2)`	Lexicographical comparison of two ranges	`bool`
`lexicographical_compare(b,e,b2,e2,f)`	Lexicographical comparison of two ranges using the criterion `f`	`bool`

Copying and Moving

`copy(b,e,out)`	Copy range `[b:e)` into `[out:out+(e-b))`
`copy_backward(b,e,out)`	Copy range `[b:e)` in reverse order into `out:out+(e-b)`
`copy_n(b,n,out)`	Copy all elements in range `[b:(b+n)]` into `[out:(out+n))`
`copy_if(b,e,out,f)`	Copy all elements in range `[b:e)` into `[out:out+(e-b))` for which `f(*p)` is true
`move(b,e,out)`	Move range `[b:e)` into `[out:out+(e-b))` (C++11)
`move_backward(b,e,out)`	Move range `[b:e)` in reverse order into `out:out+(e-b)` (C++11)
`swap_ranges(b,e,b2)`	Swap elements in range `[b:e)` with those in range `[b2:b2+(b-e))`

Setting and Replacing Values

`fill(b,e,v)`	Set all elements in range `[b:e)` equal to `v`
`fill_n(b,n,v)`	Set first `n` elements from `b` equal to `v`
`generate(b,e,f)`	Set all elements in range `[b:e)` equal to `f()`
`generate_n(b,n,f)`	Set first `n` elements from `b` equal to `f()`
`replace(b,e,v,v2)`	Replace all elements in range `[b:e)` which are equal to `v` with `v2`
`replace_if(b,e,f,v2)`	Replace all elements in range `[b:e)` for which `f(*p)` is true with `v2`
`replace_copy(b,e,out,v,v2)`	Create copy of all elements in range `[b:e)` replacing elements which are equal to `v` with `v2`, return iterator to end of copy
`replace_copy_if(b,e,out,f,v2)`	Create copy of all elements in range `[b:e)` replacing elements for which `f(*p)` is true with `v2`, return iterator to end of copy

Changing Values

`transform(b,e,out,f)`	Apply operation `*q=f(*p)` to each element `p` in range `[b:e)` and write results `q` into range `[out:out+(e-b))`
`transform(b,e,b2,out,f)`	Apply operation `*q=f(*p1,*p2)` to each pair of `p1` in range `[b:e)` and `p2` in range `[b2:b2+(e-b))` and write the results `q` into range `[out:out+(e-b))`

transform vs. for_each

```
#include<algorithm>
#include<cstdlib>
#include<iterator>
#include<list>
#include<iostream>
#include<functional>

int myrand()
{
  return 1 + (int) (10.0 * (rand() / (RAND_MAX + 1.0)));
}

template<typename T>
void print(const char* prefix,const T& coll){
  std::cout<<prefix;
  std::copy(coll.begin(), coll.end(),
            std::ostream_iterator<int>(std::cout, " "));
  std::cout<<std::endl;
```

transform vs. for_each

```
template<typename T>
void multAssign(T& t){
  t=std::bind1st(std::multiplies<T>(),10)(t);
}

int main(){
  std::list<int> coll;
   std::generate_n(std::back_inserter(coll), 9,
                        myrand);
  print("initial: ", coll);
```

transorm vs. for_each

```
  std::for_each(coll.begin(),coll.end(),
                multAssign<int>);
  print("for_each: ", coll);
  std::transform(coll.begin(),coll.end(), coll.begin(),
                 std::bind1st(std::multiplies<int>(),10));
  print("transform: ", coll);
}
```

Output:

```
initial: 9 4 8 8 10 2 4 8 3
for_each: 90 40 80 80 100 20 40 80 30
transform: 900 400 800 800 1000 200 400 800 300
```

Deletion Algorithms

`remove(b,e,v)`	Remove all items in range `[b:e)` which are equal to `v`
`remove_if(b,e,f)`	Remove all items in range `[b:e)` for which `f(*p)` is true
`remove_copy(b,e,out,v)`	Create copy of range `[b:e)` with all elements equal to `v` removed
`remove_copy_if(b,e,f)`	Create copy of range `[b:e)` with all elements with `f(*p)` true removed
`unique(b,e)`	Remove all consecutive duplicates
`unique(b,e,f)`	Remove all consecutive elements for which `f(*p,*(p+1))` is true
`unique_copy(b,e,out)`	Create duplicate free copy
`unique_copy(b,e,out,f)`	Create duplicate free copy

- Elements will be overwritten with the following elements that are not removed, and the relative order of the remaining elements is preserved.
- The functions return an iterator that points to the location after the remaining range. The elements between this iterator and the iterator `e` are no longer valid, but may still be accessed. They can be removed by calling the `erase()` method of the container.

Example for Deletion

```
#include<list>
#include<algorithm>
#include<iostream>
#include<iterator>

template<typename T>
void print(T coll)
{
  typedef typename T::value_type value_type;
  std::copy(coll.begin(), coll.end(),
       std::ostream_iterator<value_type>(std::cout, " "));
  std::cout<<std::endl;
}

int main(){
  std::list<int> coll;

  for(int i=0; i<6; ++i){
    coll.push_front(i);
    coll.push_back(i);}

  std::cout<<"pre:      "; print(coll);
```

Example for Deletion

```
std::list<int>::iterator newEnd = remove(coll.begin(),
    coll.end(), 3);

std::cout<<"post:    "; print(coll);

coll.erase(newEnd, coll.end());

std::cout<<"removed: "; print(coll);}
```

Output of sample program:

```
pre:     5 4 3 2 1 0 0 1 2 3 4 5
post:    5 4 2 1 0 0 1 2 4 5 4 5
removed: 5 4 2 1 0 0 1 2 4 5
```

Swap Algorithms

`reverse(b,e)`	Reverse order of elements in range `[b:e)`
`reverse_copy(b,e,out)`	Create copy of range `[b:e)` with reversed order
`rotate(b,m,e)`	Move all elements cyclically `m` elements to the left
`rotate_copy(b,m,e,out)`	Create copy with all elements cyclically moved `m` elements to the left
`random_shuffle(b,e)`	Create random order of range content
`random_shuffle(b,e,f)`	Create random order of range content with random number generator `f`
`partition(b,e,f)`	Move all elements with `f(*p)` true to the front
`stable_partition(b,e,f)`	As `partition`, but keep order within partitions

Algorithms that Change Content and Associative Containers

- Iterators of associative containers do not allow assignment, because the unchangeable key is part of `value_type`.
- Therefore, they can not be used as a target of a content changing algorithm.
- Using them will cause a compiler error.
- Instead of deletion algorithms the container method `erase` can be used.
- Results can be saved in such containers by using an insert iterator adapter (see below).

Algorithms versus Container Methods

- While the STL algorithms can be generally applied to any container, they often do not have the optimal complexity for a given container.
- If speed is an issue container methods should be used.
- To remove all items with a value of 4 from a `list` the method call `coll.remove(4)` should be used for example instead of

```
coll.erase(remove(coll.begin(),coll.end(), 4), coll.end);
```

Sorting Algorithms

`sort(b,e)`	Sort all elements in the range `[b:e)` (based on Quicksort)
`stable_sort(b,e)`	Sort all items in the range `[b:e)` while maintaining the order of equal elements (based on Mergesort)
`partial_sort(b,m,e)`	Sort until the first `m` elements have the right order (based on Heapsort)
`partial_sort_copy(b,e,b2,e2)`	Create copy of smallest `e2-b2` elements with correct order (based on Heapsort)

All these algorithms are also available as a version with comparison operator `f`.

Heaps

`make_heap(b,e)`	Convert the elements in range `[b:e)` into a heap
`push_heap(b,e)`	Insert element `*(e-1)` in heap `[b:e-1)`, so that in the end `[b:e)` is a heap
`pop_heap(b,e)`	Delete element `*(e-1)` from heap, so that in the end `[b:e-1)` is a heap
`sort_heap(b,e)`	Sort the heap in ascending order (is afterwards no longer a heap)

All these algorithms are also available as a version with comparison operator `f`.

Search in Presorted Ranges

`binary_search(b,e,v)`	Check if range `[b:e)` contains element equal to `v`	bool
`lower_bound(b,e,v)`	First element in range `[b:e)` that is greater than or equal to `v`, or `e`	iterator
`upper_bound(b,e,v)`	First element in range `[b:e)` that is greater than `v`, or `e`	iterator
`equal_range(b,e,v)`	Iterators to the range equal to `v` in range `[b:e)`, or twice `e`	`pair<iterator>`

All these algorithms are also available as a version with comparison operator `f`.

Merge Presorted Ranges

`merge(b,e,b2,e2,out)`	Merge two ranges
`merge(b,e,b2,e2,out,f)`	Merge two ranges
`inplace_merge(b,m,e)`	Merge two consecutive sorted ranges `[b:m)` and `[m:e)` into `[b:e)`
`inplace_merge(b,m,e,f)`	Merge two consecutive sorted ranges `[b:m)` and `[m:e)` into `[b:e)`

Algorithms for Presorted Ranges: Sets

`includes(b,e,b2,e2)`	Check if all elements from range `[b:e)` are also in range `[b2:e2)`	bool
`set_union(b,e,b2,e2,out)`	Sorted union of ranges `[b:e)` and `[b2:e2)`	Iterator to last element
`set_intersection(b,e,b2,e2,out)`	Sorted intersection of ranges `[b:e)` and `[b2:e2)`	Iterator to last element
`set_difference(b,e,b2,e2,out)`	Sorted set of elements in `[b:e)` but not in `[b2:e2)`	Iterator to last element
`set_symmetric_ difference(b,e,b2,e2,out)`	Elements that are either in range `[b:e)` or range `[b2:e2)` but not in both	Iterator to last element

All these algorithms are also available as a version with comparison operator `f`.

Numerical Algorithms

`accumulate(b,e,i)`	Composition of all elements in range `[b:e)` with operator `plus` and initial value `i`
`accumulate(b,e,i,f)`	Composition of all elements in range `[b:e)` with binary operator `f` and initial value `i`
`inner_product(b,e,b2,i)`	Composition of ranges `[b:e)` and `[b2:b2+(e-b))` as a scalar product with initial value `i`
`inner_product(b,e,b2,i,f,f2)`	Composition of `(*p)` and `(*q)` from ranges `[b:e)` and `[b2:b2+(e-b))` with operator `f`, and composition of results with operator `f2` and initial value `i`

Numerical Algorithms

`adjacent_difference(b,e,out)`	First element of `out` is `*b`, afterwards difference `*p-*(p-1)`
`adjacent_difference(b,e,out,f)`	First element of `out` is `*b`, afterwards `f(*p,*(p-1))`
`partial_sum(b,e,out)`	First element of `out` is `*b`, afterwards elements `q` of `out` are `*(q-1)+p`
`partial_sum(b,e,out)`	First element of `out` is `*b`, afterwards elements `q` of `out` are `f(*(q-1),p)`
`iota(b,e,v))`	Assigns value `++v` to each element of range `[b:e)`