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

Summer Semester 2018

Motivation for Iterators

Reasons for Defining Iterators:

- How does one access the entries of associative containers, for example a `set`?
- How does one write algorithms that work 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 also work for traditional C arrays (for legacy code that doesn't use STL `array` and `vector`).
- 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 are instances of different classes).
- is `Assignable`, `DefaultConstructible` and `EqualityComparable`.
- is pointing at a specific position in a container object (or data stream).
- The next element of the container can be reached using the `operator++` of the iterator.

Note: Directly using iterators can be quite verbose. For this reason, range-based for loops (similar to those in Python) have been introduced in C++11 (later). However, behind the scenes these are still based on iterators, which makes iterators an important topic for user-defined classes.

Example of Iterators

Two illustrations for the inner workings of iterators:

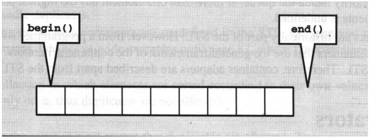

Abbildung: Iterating over a vector

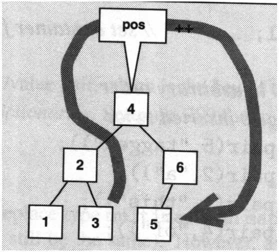

Abbildung: Iterating over a set

Iterators for Containers

Each STL container implements corresponding iterators.

- 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 entry *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()
{
    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
```

Iterator Concepts

Iterators can have additional properties, depending 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 forward (once)
Output iterator	write forward (once)
Forward iterator	iterate forward (multipass)
Bidirectional iterator	read and write backward and forward
Random access iterator	read and write at arbitrary positions

Tabelle: Predefined Iterators

Iterator Concepts

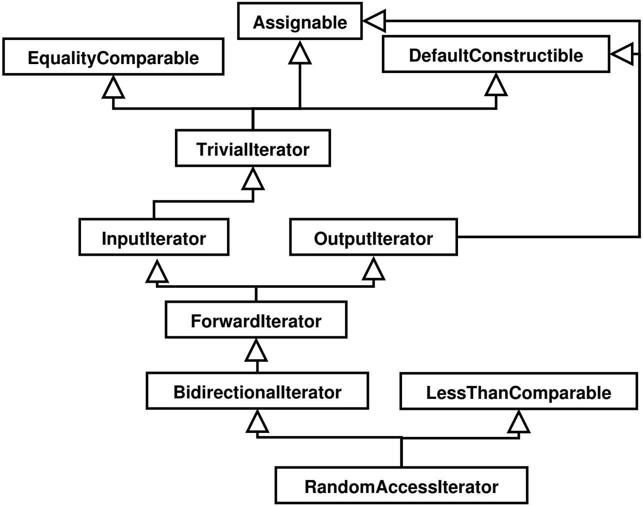

Abbildung: Iterator concepts

TrivialIterator

`TrivialIterator`, the most basic type of 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`, the type of object pointed to.
- Available methods:

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

- Complexity guarantees: All operations have amortized constant complexity.

Note: Apart from using `element_type` instead of `value_type` to designate the type of objects, this interface is fulfilled by the smart pointers introduced in C++11.

InputIterator

`InputIterator`, an extension of `TrivialIterator`:

- Is an object that points to another object, can be dereferenced like a pointer, and can additionally be incremented to get an iterator to the next object.
- Associated types: `difference_type`, type to store the distance between two iterators
- Additional methods:

`++i`	Takes a step forward
`i++`	Takes a step forward, identical to `++i` (but see below)
`*i++`	Identical to `T t = *i; ++i; return t;`

- Complexity guarantees: All operations have amortized constant complexity.

Note: The postfix version of `operator++` returns the *previous* iterator state. Since this can involve copying possibly large referenced objects, the prefix variant is generally preferred in generic algorithms.

OutputIterator

The special case `OutputIterator`:

- Is an object that can be written to and which can be incremented.
- Is not comparable and doesn't have to define `value_type` and `difference_type`.
- May be compared to those old continuous paper printers.
- Increment and assignment must alternate. The sequence has to start with an assignment, then an increment follows, then an assignment, and so forth.
- Available methods:

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

- Complexity guarantees: All operations have amortized constant complexity.

ForwardIterator

The concept `ForwardIterator`:

- Corresponds to the common notion of a linear sequence of values. With a `ForwardIterator` multiple passes through a container are possible (as opposed to `InputIterator` and `OutputIterator`).
- Defines no additional methods, only gives further guarantees.
- Incrementing does not invalidate earlier copies of the iterator (in contrast to `InputIterator` and `OutputIterator`).
- Complexity guarantees: All operations have amortized constant complexity.
- Assurances: For two `ForwardIterators` `i` and `j` the following holds: if `i == j` then `++i == ++j`

`BidirectionalIterator`

`BidirectionalIterator`, an extension of `ForwardIterator`:

- Can be used forward and backward.
- Is provided by `list`, `set`, `multiset`, `map` and `multimap`.
- Additional methods:

`--i`	Takes a step backward
`i--`	Takes a step backward

- 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 (no-op).

RandomAccessIterator

RandomAccessIterator, an extension of BidirectionalIterator:

- Provides additional methods in order to make steps of any size, both forward and backward, in constant time. In principle, it allows all 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 forward
`i -= n`	Moves n elements backward
`i[n]`	Equivalent to `*(i+n)`
`i - j`	Returns the distance between i and j

- Complexity guarantees: All operations have amortized constant complexity.

`RandomAccessIterator`

- Assurances:
  - If `i + n` is defined, then `i += n; i -= n;` and `i -= n; i += n;` are null operations (no-op).
  - If `i - j` is defined, then `i == j + (i-j)` is true.
  - Two iterators are `Comparable` (`LessThanComparable`).

Note: Through the use of traits (later), raw pointers can also be used as `RandomAccessIterators`. Thus, all suitable STL algorithms are readily available for C-style legacy code.

Iterator 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] = "foo";
    b.push_back("bar");
    typedef std::vector<std::string>::iterator VectorIt;
    for (VectorIt i = b.begin(); i < b.end(); ++i)
        std::cout << *i << std::endl;
}
```

Iterator 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;
    // one of the few valid uses of the comma operator (!)
    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 for Set: Storage 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

```cpp
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::endl << std::endl;
            std::cerr << "Available keys: " << std::endl;
            for (element = container.begin(); element != container.end(); ++element)
                std::cerr << element->first << std::endl;
            throw "value not found";
        }
    }
    return defValue;
}
```

C++11: Range-based for Loop

C++11 introduces a range-based for loop, which is implemented using iterators. Instead of the verbose loop:

```
typedef std::vector<std::string>::iterator VectorIt;
for (VectorIt i = b.begin(); i < b.end(); ++i)
  std::cout << *i << std::endl;
```

one can now just write:

```
for (const std::string& elem : b)
  std::cout << elem << std::endl;
```

It is therefore no longer necessary to use iterators in the most common situation: iterating over the full container in one forward pass. However, iterators remain useful for cases when more complex access patterns are required (e.g. visiting every second element, or moving back and forth).

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 extensively for input and output.
- The header `algorithm` must be included. The header `numeric` defines additional algorithms that perform calculations.
- Some of the algorithms expect functors as arguments, which can be user-supplied. Some standard functors can be found in the header `functional`. One can also use C++11 lambda functions (later) as argument.

Example for STL Algorithms

```cpp
#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;
}
```

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 two iterators: `[begin,end)`, with `begin` pointing 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.

Algorithms with Suffix

_if (cont.)
- 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`.

_copy suffix
- Without suffix the content of each element is changed, with suffix the 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`.

Other possible suffixes: `_n` to specify a number of elements instead of a range, `_until` to find e.g. max sorted subsequences instead of just checking for sortedness.

This can also be combined, e.g. `_copy_if` to create a copy and use a predicate.

Predefined Functors

The following functors / predicates can be found in header `<functional>`:

`negate<T>`	`- v`
`plus<T>`	`v1 + v2`
`minus<T>`	`v1 - v2`
`multiplies<T>`	`v1 * v2`
`divides<T>`	`v1 / v2`
`modulus<T>`	`v1 % v2`
`equal_to<T>`	`v1 == v2`
`not_equal_to<T>`	`v1 != v2`
`less<T>`	`v1 < v2`
`less_equal<T>`	`v1 <= v2`

`greater<T>`	`v1 > v2`
`greater_equal<T>`	`v1 >= v2`
`logical_not<T>`	`!v`
`logical_and<T>`	`v1 && v2`
`logical_or<T>`	`v1 || v2`
`bit_and<T>`	`v1 & v2`
`bit_or<T>`	`v1 | v2`
`bit_xor<T>`	`v1 ^ v2`
`bit_not<T>`	`~v` (C++14)

(`v`, `v1` and `v2` are objects of type `T`)

C++11: `std::function` and `std::bind`

C++11 adds a general-purpose function wrapper class `function` that can store:

- Ordinary free functions
- Functors (function objects)
- Lambda expressions (later)
- Bind expressions (see below)

It also add a function template `bind` that:

- Is based on variadic templates (later)
- Modifies other functions / function objects
- Fixes (binds) one or several arguments

While `bind` removes function arguments, the function `mem_fn` adds an argument:

- It makes the implicit object argument of methods explicit
- The result is a free function with additional (object) argument
- There are automatic versions for pointers / smart pointers
- Can also be used to access (public) data members of objects

Example for Functors and Binding

```
#include <functional>

struct Foo
{
  int data;
  bool check(int number) const
  {
    return number == data;
  }
};

int f(int a, int b, int c) {return a + b*c;}
int g(int d)               {return d*d;}

int main()
{
  Foo foo;
  foo.data = 8;
```

Example for Functors and Binding

```
  std::function<bool(Foo,int)> checkMember
      = std::mem_fn(&Foo::check);
  std::function<int(Foo)> getData
      = std::mem_fn(&Foo::data);

  checkMember(foo,getData(foo));

  using std::placeholders::_1;
  std::function<bool(int)> greaterThanFive
      = std::bind(std::greater<int>(), _1, 5);

  bool probably = greaterThanFive(getData(foo));

  using std::placeholders::_2;
  std::function<int(int,int)> h
      = std::bind(f, _2, std::bind(g, _1), _1);
  // h(x,y) = f(y,g(x),x) = y + x^3

  int result = h(10,getData(foo)); // 1008
}
```

for_each

The simplest 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.

Note: The `for_each` algorithm can be replaced by C++11 range-based for loops, which can be just as fast and more readable, but algorithms that don't just operate on individual elements (e.g. search or aggregation) remain important also in C++11 and above.

For the remaining algorithms, the same notation will be used:

- `b` and `e` for beginning and end of range
- `v` for value, `f` for function / predicate
- `p` and `q` for iterators, `n` for number
- `out` for iterator to output location

Counting and Checking

`count(b,e,v)`	Number of elements in range [`b`:`e`) which are `v`
`count_if(b,e,f)`	Number of elements in range [`b`:`e`) for which `f(*)` is true
`all_of(b,e,f)`	True iff `f(*)` is true for all elements in range [`b`:`e`) (C++11)
`any_of(b,e,f)`	True iff `f(*)` is true for at least one element in range [`b`:`e`) (C++11)
`none_of(b,e,f)`	True iff `f(*)` is true for none of the elements in range [`b`:`e`) (C++11)

Minima and Maxima

`min_element(b,e)`	Smallest element in range [`b`:`e`)
`min_element(b,e,f)`	Element in range [`b`:`e`) with smallest `f(*)`
`max_element(b,e)`	Largest element in range [`b`:`e`)
`max_element(b,e,f)`	Element in range [`b`:`e`) with largest `f(*)`
`minmax_element(b,e)`	Smallest and largest element in range [`b`:`e`) (C++11)
`minmax_element(b,e,f)`	Elements in range [`b`:`e`) with smallest and largest `f(*)` (C++11)

Note: These functions return iterators resp. pairs of iterators to elements.

Search Algorithms

`find(b,e,v)`	First element in range [`b`:`e`) with value `v`
`find_if(b,e,f)`	First element in range [`b`:`e`) for which `f(*)` is true
`find_if_not(b,e,f)`	First element in range [`b`:`e`) for which `f(*)` is false (C++11)
`find_first_of(b,e,b2,e2)`	First element in range [`b`:`e`) for which element from range [`b2`:`e2`) is the same
`find_first_of(b,e,b2,e2,f)`	First element in range [`b`:`e`) for which `f(*,*)` is true for element from range [`b2`:`e2`)

Note: These functions return iterators to elements, or e if nothing was found. All algorithms that take more than one range effectively treat the two ranges as a range of pairs, i.e. they iterate over both ranges simultaneously.

Search Algorithms

`find_end(b,e,b2,e2)`	Last element in range `[b:e)` that is the same as element from range `[b2:e2)`
`find_end(b,e,b2,e2,f)`	Last element in range `[b:e)` for which `f(*,*)` is true for element from range `[b2:e2)`
`adjacent_find(b,e)`	First element that is equal to its successor in range `[b:e)`
`adjacent_find(b,e,f)`	First element for which `f(*,*)` is true with its successor

Search Algorithms

`search(b,e,b2,e2)`	First element in `[b:e)`, so that next `e2 - b2` elements are like those of range `[b2:e2)`
`search(b,e,b2,e2,f)`	First element in `[b:e)`, so that `f(*,*)` is true for the next `e2 - b2` elements
`search(b,e,searcher)`	Use custom Searcher class (C++17)
`search_n(b,e,n,v)`	First element for which this and the following `n-1` elements are equal to `v`
`search_n(b,e,n,v,f)`	First element for which `f(*,v)` is true for itself and the next `n-1` elements

Comparison Algorithms

`equal(b,e,b2)`	True iff all elements of the two ranges `[b:e]` and `[b2:b2+(b-e))` are equal
`equal(b,e,b2,f)`	True iff `f(*,*)` is true for all elements of the two ranges
`mismatch(b,e,b2)`	First elements in `[b:e)` and `[b2:b2+(b-e))` which aren't equal
`mismatch(b,e,b2,f)`	First elements in the two ranges for which `f(*,*)` is false
`lexicographical_compare(b,e,b2,e2)`	Lexicographical comparison of two ranges
`lexicographical_compare(b,e,b2,e2,f)`	Lexicographical comparison of two ranges using criterion `f`

C++14: All versions of `equal` and `mismatch` can have additional argument `e2` (end of second range)

Copying, Moving and Removing

`copy(b,e,out)`	Copy range `[b:e)` into `[out:out+(e-b))`
`copy_backward(b,e,out)`	Copy range, start at end
`copy_n(b,n,out)`	Copy all elements in range `[b:(b+n))` into `[out:(out+n))` (C++11)
`copy_if(b,e,out,f)`	Copy all elements for which `f(*)` is true (C++11)
`move(b,e,out)`	Move range `[b:e)` into `[out:out+(e-b))` (C++11)
`move_backward(b,e,out)`	Move range, but start at end (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 of range equal to `v`
`fill_n(b,n,v)`	Set first `n` elements equal to `v`
`generate(b,e,f)`	Set all elements of range equal to `f()`
`generate_n(b,n,f)`	Set first `n` elements equal to `f()`
`replace(b,e,v,v2)`	Replace all elements which equal to `v` with `v2`
`replace_if(b,e,f,v2)`	Replace all elements for which `f(*)` is true with `v2`
`replace_copy(b,e,out,v,v2)`	Create copy, replacing elements which are equal to `v` with `v2`
`replace_copy _if(b,e,out,f,v2)`	Create copy, replacing elements for which `f(*)`is true with `v2`

Note: The generator `f` of `generate` and `generate_n` can be a functor with internal state, and therefore produce a sequence of numbers / objects.

Changing Values

`for_each(b,e,f)`	Apply operation `f(*)` to each element, overwrite the element with the result
`for_each_n(b,n,f)`	Apply operation `f(*)` to first `n` elements, overwriting with result (C++17)
`transform(b,e,out,f)`	Apply operation `f(*)` to each element in range `[b:e)` and write results into range `[out:out+(e-b))`
`transform(b,e,b2,out,f)`	Apply operation `f(*,*)` to each pair of elements from range `[b:e)` and `[b2:b2+(e-b))` and write results 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. * (rand() / (RAND_MAX + 1.)));
}

template<typename T>
void print(const char* prefix, const T& coll)
{
    std::cout << prefix;
    // ostream_iterator: represent cout as iterator range
    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::bind(std::multiplies<T>(),10,std::placeholders::_1)(t);
}

int main()
{
  std::list<int> coll;
  // back_inserter: represent push_back() as iterator range
   std::generate_n(std::back_inserter(coll), 9, myRand);
  print("initial: ", coll);
```

transform 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::bind(std::multiplies<int>(),10,
                std::placeholders::_1));
  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 equal to `v`
`remove_if(b,e,f)`	Remove all items in range for which `f(*)` is true
`remove_copy(b,e,out,v)`	Create copy `[b:e)` with all elements equal to `v` removed
`remove_copy_if(b,e,f)`	Create copy with all elements with `f(*)` true removed
`unique(b,e)`	Remove all consecutive duplicates
`unique(b,e,f)`	Remove all consecutive elements for which `f(*,*)` is true
`unique_copy(b,e,out)`	Create duplicate free copy
`unique_copy(b,e,out,f)`	Create duplicate free copy based on `f`

- Elements will be overwritten with following elements that are not removed, and the relative order of 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

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

template<typename T>
void print(T coll)
{
  using value_type = typename T::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);
  }
```

Example for Deletion

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

  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
```

Permutation Algorithms

`reverse(b,e)`	Reverse order of elements in range
`reverse_copy(b,e,out)`	Create copy of range with reversed order
`rotate(b,m,e)`	Move 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
`is_permutation(b,e,b2)`	True iff range `[b,e)` is permutation of range starting at `b2` (C++11)
`is_permutation(b,e,b2,f)`	True iff range `[b,e)` is permutation of range starting at `b2`, with two elements equivalent if `f(*,*)` is true (C++11)
`next_permutation(b,e)`	Change content of range `[b,e)` to lexicographically next permutation
`prev_permutation(b,e)`	Change content of range `[b,e)` to lexicographically previous permutation

There are also versions of `next_permutation` and `prev_permutation` that take an argument `f` that is used for sorting (comparison operator).

Shuffle Algorithms

`random_shuffle(b,e)`	Create random order of range content (removed in C++17)
`random_shuffle(b,e,f)`	Create random order of range content with RNG `f` (removed in C++17)
`shuffle(b,e,f)`	Create random order of range content with RNG `f` (C++11)
`sample(b,e,out,n,f)`	Create `n` random samples from `[b,e)` using RNG `f` and copy to out (C++17)

The old algorithm `random_shuffle` is based on the C method `random`, deprecated in C++11, and removed in C++17.

The new algorithm `shuffle` is based on C++11 random number generators with significantly more flexibility (later).

Modifying Algorithms 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, and 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 (similar to the iterator adapters in previous examples).

Algorithms versus Container Methods

- While the STL algorithms can be applied to any container, they often do not have the optimal complexity for a given container.
- Container methods should be used if
  - speed is an issue (benchmark!)
  - the type of container is known beforehand
  - the container provides the functionality
- For example: to remove all items with a value of 4 from a `list` the method call `coll.remove(4)` should be used instead of

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

Partitioning Algorithms

`is_partitioned(b,e,f)`	True iff all elements with `f(*)` true appear before those with false (C++11)

`partition(b,e,f)`
`partition_copy`
`(b,e,out,out2,f)`
	Move all elements with `f(*)` true to front
	Copy all elements with `f(*)` true to `out`, the rest to `out2` (C++11)

`stable_partition(b,e,f)`	Move all elements with `f(*)` true to front, preserving relative order in partitions
`partition_point(b,e,f)`	First element for which `f(*)` is false (C++11)

Heap Algorithms

Note: A heap (max heap) is a range `[p,q)` so that
`p[floor((i-1)/2)] >= p[i]` for all distances `i`
(a convenient structure that is roughly sorted in descending order)

`is_heap(b,e)`	True iff elements in range are a heap (C++11)
`is_heap_until(b,e)`	Last iterator `p` so that `[b,p)` is a heap (C++11)
`make_heap(b,e)`	Convert elements in range `[b:e)` into a heap
`push_heap(b,e)`	Insert element at `e-1` in heap `[b:e-1)`, so that in the end `[b:e)` is a heap
`pop_heap(b,e)`	Delete largest / first element from heap and move it to `e-1`, 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`.

Sorting Algorithms

`is_sorted(b,e)`	True iff elements are in non-descending order (C++11)
`is_sorted_until(b,e)`	First element that is not in non-descending order (C++11)
`sort(b,e)`	Sort all elements in range (based on Quicksort)
`stable_sort(b,e)`	Sort all items in range while maintaining the order of equal elements (based on Mergesort)
`partial_sort(b,m,e)`	Sort until the first `m-b` elements have the right order (based on Heapsort)
`partial_sort_copy(b,e,b2,e2)`	Create copy of smallest `e2-b2` elements
`nth_element(b,p,e)`	Partial sort that puts correct element in position `p`, larger elements in `[p,e)`, and smaller ones in `[b,p)` (based on Introselect)

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

Algorithms for Presorted Ranges: Search

These algorithms assume that the range `[b,e)` is already sorted.

`binary_search(b,e,v)`	True iff range contains element equal to `v`
`lower_bound(b,e,v)`	First element in range greater than or equal to `v`
`upper_bound(b,e,v)`	First element in range greater than `v`
`equal_range(b,e,v)`	Pair of iterators to range equal to `v` in `[b:e)`

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

Algorithms for Presorted Ranges: Set Operations

These algorithms assume that the range `[b,e)` is already sorted.

`merge(b,e,b2,e2,out)`	Merge two sorted ranges
`inplace_merge(b,m,e)`	Merge two consecutive sorted ranges `[b:m)` and `[m:e)` into `[b:e)`
`includes(b,e,b2,e2)`	True iff all elements from range `[b:e)` are also in range `[b2:e2)`
`set_union(b,e,b2,e2,out)`	Sorted union of ranges `[b:e)` and `[b2:e2)`
`set_intersection (b,e,b2,e2,out)`	Sorted intersection of ranges `[b:e)` and `[b2:e2)`
`set_difference (b,e,b2,e2,out)`	Sorted set of elements in `[b:e)` but not in `[b2:e2)`
`set_symmetric_ difference(b,e,b2,e2,out)`	Elements that are either in range `[b:e)` or range `[b2:e2)` but not in both

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

Numerical Algorithms

All remaining algorithms are not in `<algorithm>`, but in `<numeric>`.

`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 two elements from ranges `[b:e)` and `[b2:b2+(e-b))` with operator `f`, composition of results with operator `f2` and initial value `i`

Numerical Algorithms

`adjacent_difference(b,e,out)`	First element of `out` is that of `b`, afterwards difference between consecutive elements
`adjacent_difference(b,e,out,f)`	First element of `out` is that of `b`, afterwards `f(*,*)` of consecutive elements
`partial_sum(b,e,out)`	First element of `out` is that of `b`, afterwards elements of `out` are sum of predecessor and element from `[b:e)`
`partial_sum(b,e,out,f)`	First element of `out` is that of `b`, afterwards elements of `out` are result of `f(*,*)` with predecessor and element from `[b:e)` as input
`iota(b,e,v)`	Assigns value `++v` to each element of range `[b:e)` (C++11)

C++17: Execution Policies

The last few years brought a consistent shift into the direction of parallelization due to the wide availability of multicore CPUs and GPUs (see introductory slides), and C++17 acknowledges that by adding explicit support for execution policies.

The following policies may be specified as an optional first argument for most of the mentioned STL algorithms:

`std::execution::seq` (of class `std::execution::sequenced_policy`):
Forced sequential execution — the algorithm has to be executed sequentially, even in parallel programs.

`std::execution::par` (of class `std::execution::parallel_policy`):
The algorithm may be parallelized through implicit thread generation. Element access in same thread is indeterminately sequenced.

`std::execution::par_unseq` (of class `std::execution::parallel_unsequenced_policy`):
The algorithm may be parallelized through implicit thread generation. Additionally, element access need no longer be sequenced per thread, enabling the use of vectorization on suitable architecture (interleaving of operations).

C++17: Additional Numerical Algorithms

While standard STL algorithms include function overloads for execution policies, the numerical algorithms have been renamed, because there are subtle differences for out-of-order evaluations when the underlying operations are not associative.

`reduce(b,e,i)`	Similar to `accumulate`, but allows out-of-order operations and therefore parallelization / optimization
`reduce(b,e)`	As above, but uses default constructor for initial value
`reduce(b,e,i,f)`	As above, but uses binary operator `f` for composition
`transform_reduce(b,e,b2,i)`	Similar to `inner_product`, but allows out-of-order operations
`transform_reduce (b,e,b2,i,f,f2)`	Similar to `inner_product` with operators `f(*,*)` and `f2(*,*)`
`transform_reduce(b,e,i,f,f2)`	Applies transformation `f2` to each element and reduces using operator `f`

C++17: Additional Numerical Algorithms

`inclusive_scan(b,e,out)`	Similar to `partial_sum`, but allows out-of-order operations and therefore parallelization / optimization
`inclusive_scan(b,e,i,f)`	As above, but uses binary operator `f` for composition (generalized noncommutative sum)
`inclusive_scan(b,e,b2,i)`	As above, but with additional initial value `i`
`exclusive_scan(b,e,out)`	Same as `inclusive_scan`, but omits i-th element in i-th sum
`exclusive_scan(b,e,i,f)`	Same as `inclusive_scan`, but omits i-th element in i-th generalized sum
`exclusive_scan(b,e,b2,f,i)`	As above

There are also algorithms `transform_inclusive_scan` and `transform_exclusive_scan` which take an additional argument `f2` that is used for transformation as in the case of `transform_reduce`.