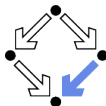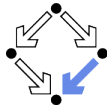# The Standard Library

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at
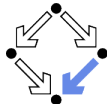
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.jku.at

1. General

2. Sequence Containers

3. Iterators

4. Adaptors

5. Associative Containers

6. Algorithms

The Standard Library

- Set of headers with declarations.

  `#include <name>`

  - Headers need not be physical files (do not use `<name.h>`).
- Almost all names are in namespace `std`.

  `using namespace std;`

  - Only exceptions are global operators `new` and `delete` (header `<new>`).
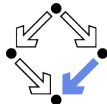- Provides lot of basic functionality.
  - Numerics.
  - Input/Output.
  - Containers, iterators, algorithms.
    - We will look at these in more detail.

For effective programming, it is important to know not only a programming language but also the associated basic libraries.

C Library Wrappers

For backward compatibility, the entire C standard library is included.

C++ Header	C Header
`<cstdio>`	`<stdio.h>`
`<cstdlib>`	`<stdlib.h>`
`<cstring>`	`<string.h>`
`<cmath>`	`<math.h>`
...	...

- Use of C++ header (places name in namespace `std`)
  ```
  #include <cstdio>
  int main() { std::printf("Hello, world"); }
  ```
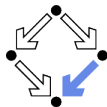- Use of C header (places name in global namespace)
  ```
  #include <stdio.h>
  int main() { printf("Hello, world"); }
  ```

The C++ library provides better alternatives for writing new applications.

Traits and Policies

The standard library makes heavy use of traits and policies.

- Trait: a class that provides information about a type.
    - By type definitions and/or static member data in the trait.
- Policy: a trait that also defines an operational interface for the type.
    - By static member functions in the policy.
- Often implemented as specializations of dummy templates.

```
template <type T> class Trait { };   // dummy trait template
template<> class Trait<int> { ... }; // trait for type "int"
```

    - Thus the trait for a type can be deduced from the name of a type.
- Mainly used as template arguments.

```
template<class C, class T = Trait<C> >
class Lib { ... C ... T::member ... };
```

    - Template thus receives required information about type parameter.
    - Since trait holds information, atomic type can be template argument.

Many standard types are instantiations of templates with traits/policies.

Example: **Class** `string`
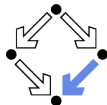
C++ strings are actually parameterized over the character type.

```
// header <string>
template<typename charT> struct char_traits;
template<> struct char_traits<char> { ... }

template<class charT, class traits = char_traits<charT>, ... >
class basic_string { ... }
typedef basic_string<char> string;
```

- **Wide** character type: `wchar_t`
    - Narrow character type `char` is only one byte large.
    - `wchar_t` is typically 32 bit large and may hold any Unicode character.
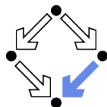   ```
        wchar_t pi = '\u03c0'; // greek character "pi"
        ```
- **Wide** strings: another string type provided by the library.
        ```
        template<> struct char_traits<wchar_t> {...}
        typedef basic_string<wchar_t> wstring;
        ```
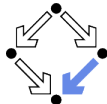
The whole library (also I/O) works with any character type.

## Example: Strings that Ignore Cases

```cpp
template<typename T> struct ci_char_traits { };
template<> struct ci_char_traits<char> {
  typedef char char_type; typedef int int_type;
  typedef std::streamoff off_type; typedef std::streampos pos_type;
  typedef std::mbstate_t state_type;
  static void assign(char_type& dst, const char_type src) { dst = src; }
  static char_type* assign(char* dst, std::size_t n, char c)
  { return static_cast<char_type*>(std::memset(dst, n, c)); }
  static bool eq(const char_type& c1, const char_type& c2)
  { return lower(c1) == lower(c2); }
  static bool lt(const char_type& c1, const char_type& c2)
  { return lower(c1) < lower(c2); }
  static int compare(const char_type* s1, const char_type* s2, std::size_t n) {
    for (size_t i = 0; i < n; i++) {
      char_type lc1 = lower(s1[i]); char_type lc2 = lower(s2[i]);
      if (lc1 < lc2) return -1; if (lc1 > lc2) return +1;
    }
    return 0;
  }
  static int_type lower(char_type c) { return std::tolower(to_int_type(c)); }
  ...
};
```
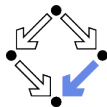
Example (Contd)

```
typedef std::basic_string<char, ci_char_traits<char> > ci_string;

int main()
{
  ci_string s1 = "Hello, World";
  ci_string s2 = "hello, world";
  std::cout << (s1 == s2); // "true";
}
```

Ray Lischner "C++ in a Nutshell".

book

Allocators

The standard library is also generic with respect to memory management.

- Allocator: a policy for managing dynamic memory.
  - Use of `new` and `delete` is not hard-wired in the standard library.
- The library provides a standard allocator

  ```
  // header <memory>
  template <class T> class allocator { ...}
  ```

- Standard library classes use this allocator by default

  ```
  // header <string>
  template<class charT, class traits = char_traits<charT>,
      class Alloc = allocator<charT> >
  class basic_string { ...}
  ```

- Other allocation schemes are possible
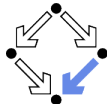
  ```
  template<> class allocator<int> {...} // globally used

  class MyCharAllocator {...}            // selectively used
  typedef basic_string<char, char_traits<char>,
      MyCharAllocator> mystring;
  ```

Example

```
template<typename T> class myallocator {
public:
  typedef std::size_t size_type; typedef std::ptrdiff_t difference_type;
  typedef T* pointer; typedef const T* const_pointer;
  typedef T& reference; typedef const T& const_reference;
  typedef T value_type;
  template <class U> struct rebind { typedef myallocator<U> other; };
  myallocator() throw() {}
  myallocator(const myallocator&) throw()          {}
  template <class U> myallocator(const myallocator<U>&) throw() {}
  ~myallocator() throw() {}
  pointer address(reference x) const {return &x;}
  const_pointer address(const_reference x) const {return &x;}
  pointer allocate(size_type n, void* hint = 0)
  { return static_cast<T*>(::operator new (n * sizeof(T)) ); }
  void deallocate(pointer p, size_type n)
  { ::operator delete(static_cast<void*>(p)); }
  size_type max_size() const throw()
  { return std::numeric_limits<size_type>::max() / sizeof(T); }
  void construct(pointer p, const T& val) { new(static_cast<void*>(p)) T(val); }
  void destroy(pointer p) { p->~T(); }
};
```

Example (Cntd)

```
template<> class myallocator<void> {
public:
  typedef void* pointer; typedef const void* const_pointer;
  typedef void value_type;
  template <class U> struct rebind { typedef myallocator<U> other; };
};

template<typename T>
bool operator==(const myallocator<T>&, const myallocator<T>&) { return true; }

template<typename T>
bool operator!=(const myallocator<T>&, const myallocator<T>&) { return false; }

int main() {
  std::list<int, myallocator<int> > data;
  data.push_back(10);
  data.push_back(20);
  return data.size();
}
```
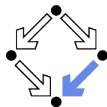
Ray Lischner "C++ in a Nutshell".   book

Standard Template Library (STL)

The core of SGI's STL was integrated into the C++ standard.

- Containers: template classes that hold arbitrary kinds of items.
  - Vectors, double ended queues, lists, (multi)sets, (multi)maps, bitsets.
    - Sequence containers: preserve order in which items are added.
    - Associative containers: fast search by sorting items according to keys.
    - Adaptors: template classes that provide abstract container interfaces.
      - Stacks, queues, priority queues.
- Iterators: (abstractions of) container pointers/indices.
  - Identify (ranges of) elements in container.
  - Same code may be used for processing different kinds of containers.
- Algorithms: template functions that implement common algorithms.
  - Processing, sorting, searching, merging, ...
    - Based on iterators, applicable to all kinds of containers.

The workhorse of generic programming in C++.

Sequence Containers (cplusplus.com)
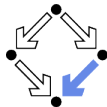
Sequence containers

|		`<array>`	`<vector>`	`<deque>`	`<forward_list>`	`<list>`
Headers		array	vector	deque	forward_list	list
Members	constructor	*implicit*	vector	deque	forward_list	list
	destructor	*implicit*	~vector	~deque	~forward_list	~list
	operator=	*implicit*	operator=	operator=	operator=	operator=
iterators	begin	begin	begin	begin	begin before_begin	begin
	end	end	end	end	end	end
	rbegin	rbegin	rbegin	rbegin		rbegin
	rend	rend	rend	rend		rend
const iterators	begin	cbegin	cbegin	cbegin	cbegin cbefore_begin	cbegin
	cend	cend	cend	cend	cend	cend
	crbegin	crbegin	crbegin	crbegin		crbegin
	crend	crend	crend	crend		crend
capacity	size	size	size	size		size
	max_size	max_size	max_size	max_size	max_size	max_size
	empty	empty	empty	empty	empty	empty
	resize		resize	resize	resize	resize
	shrink_to_fit		shrink_to_fit	shrink_to_fit		
	capacity		capacity			
	reserve		reserve			
element access	front	front	front	front	front	front
	back	back	back	back		back
	operator[]	operator[]	operator[]	operator[]		
	at	at	at	at		
modifiers	assign		assign	assign	assign	assign
	emplace		emplace	emplace	emplace_after	emplace
	insert		insert	insert	insert_after	insert
	erase		erase	erase	erase_after	erase
	emplace_back		emplace_back	emplace_back		emplace_back
	push_back		push_back	push_back		push_back
	pop_back		pop_back	pop_back		pop_back
	emplace_front			emplace_front	emplace_front	emplace_front
	push_front			push_front	push_front	push_front
	pop_front			pop_front	pop_front	pop_front
	clear		clear	clear	clear	clear
	swap	swap	swap	swap	swap	swap
list operations	splice				splice_after	splice
	remove				remove	remove
	remove_if				remove_if	remove_if
	unique				unique	unique
	merge				merge	merge
	sort				sort	sort
	reverse				reverse	reverse
observers	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator
	data	data	data			

Associative Containers (cplusplus.com)

map -> similar to mathematical functions
multimap -> similar to mathematical relations
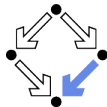
Associative containers

Headers		<set>		<map>		<unordered_set>		<unordered_map>	
Members		set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap
	constructor	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap
	destructor	~set	~multiset	~map	~multimap	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap
	assignment	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=
iterators	begin	begin	begin	begin	begin	begin	begin	begin	begin
	end	end	end	end	end	end	end	end	end
	rbegin	rbegin	rbegin	rbegin	rbegin				
	rend	rend	rend	rend	rend				
const iterators	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin
	cend	cend	cend	cend	cend	cend	cend	cend	cend
	crbegin	crbegin	crbegin	crbegin	crbegin				
	crend	crend	crend	crend	crend				
capacity	size	size	size	size	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size
	empty	empty	empty	empty	empty	empty	empty	empty	empty
	reserve					reserve	reserve	reserve	reserve
element access	at			at				at	
	operator[]			operator[]				operator[]	
modifiers	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace
	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint
	insert	insert	insert	insert	insert	insert	insert	insert	insert
	erase	erase	erase	erase	erase	erase	erase	erase	erase
	clear	clear	clear	clear	clear	clear	clear	clear	clear
	swap	swap	swap	swap	swap	swap	swap	swap	swap
operations	count	count	count	count	count	count	count	count	count
	find	find	find	find	find	find	find	find	find
	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range
	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound				
	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound				
observers	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator
	key_comp	key_comp	key_comp	key_comp	key_comp				
	value_comp	value_comp	value_comp	value_comp	value_comp				
	key_eq					key_eq	key_eq	key_eq	key_eq
	hash_function					hash_function	hash_function	hash_function	hash_function
buckets	bucket					bucket	bucket	bucket	bucket
	bucket_count					bucket_count	bucket_count	bucket_count	bucket_count
	bucket_size					bucket_size	bucket_size	bucket_size	bucket_size
	max_bucket_count					max_bucket_count	max_bucket_count	max_bucket_count	max_bucket_count
hash policy	rehash					rehash	rehash	rehash	rehash
	load_factor					load_factor	load_factor	load_factor	load_factor
	max_load_factor					max_load_factor	max_load_factor	max_load_factor	max_load_factor

Example: A Vector Program
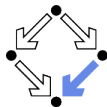
```
#include <iostream>
#include <vector>  vectors are arrays that can grow and shrink
#include <algorithm>
using namespace std;

void print(int i) { cout << i << endl; }

int main ()
{
  int values[] = {75, 23, 65, 42, 14};
  vector<int> container(values, values+5); // pointers as iterators
                         pointers to start of the array and to the end+1
  // iterate over container
  for (vector<int>::iterator it = container.begin(); it != container.end(); it++)
    cout << *it  << endl;

  // use algorithm for iteration
  for_each(container.begin(), container.end(), print);

  return 0;
}
```
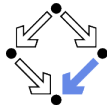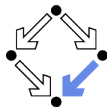
Example: A List Program

```cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

void print(int i) { cout << i << endl; }

int main ()
{
  int values[] = {75, 23, 65, 42, 14};
  list<int> container(values, values+5); // pointers as iterators

  // iterate over container
  for (list<int>::iterator it = container.begin(); it != container.end(); it++)
    cout << *it << " ";

  // use algorithm for iteration
  for_each(container.begin(), container.end(), print);

  return 0;
}
```

1. General

2. **Sequence Containers**

3. Iterators

4. Adaptors

5. Associative Containers

6. Algorithms
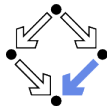
Sequence Containers

- Strict linear sequences of elements
    - `<vector>`: class template `vector`
        - Dynamic arrays.
    - `<deque>`: class template `deque` ("deck")
        - Double-ended queues.
    - `<list>`: class template `list`
        - Doubly-linked lists.
- Common operations
    - Basic: constructor, destructor, `operator=`.
    - Iterators: `begin`, `end`, `rbegin`, `rend`.
    - Capacity: `size`, `max_size`, `empty`, `resize`.
    - Sequential access: `front`, `back`.
    - Random access (not list) `operator[]`, `at`.
    - Modifiers: `assign`, `insert`, `erase`, `swap`, `clear`.
    - Modify end: `push_back`, `pop_back`.
    - Modify begin (not vector): `push_front`, `pop_front`.
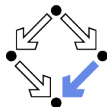
Similar interfaces, operations vary in performance.

Class Template vector

cplusplus.com: "C++ Reference".

```
template < class T, class Allocator = allocator<T> > class vector;
```

```
Vector containers are implemented as dynamic arrays; Just as regular arrays,
vector containers have their elements stored in contiguous storage locations,
which means that their elements can be accessed not only using iterators but
also using offsets on regular pointers to elements.
```

```
But unlike regular arrays, storage in vectors is handled automatically,
allowing it to be expanded and contracted as needed.
```

```
Vectors are good at:
```

```
    * Accessing individual elements by their position index (constant time).
    * Iterating over the elements in any order (linear time).
    * Add and remove elements from its end (constant amortized time).
```

```
Compared to arrays, they provide almost the same performance for these tasks,
plus they have the ability to be easily resized. Although, they usually
consume more memory than arrays when their capacity is handled automatically
(this is in order to accomodate for extra storage space for future growth).
```

Example
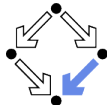
cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main () {
  vector<int> myvector (10);
  vector<int>::size_type sz = myvector.size();
  for (unsigned int i=0; i<sz; i++) myvector[i]=i;

  for (unsigned int i=0; i<sz/2; i++) { // reverse vector using operator[]
    int temp = myvector[sz-1-i];
    myvector[sz-1-i]=myvector[i];
    myvector[i]=temp;
  }

  for (unsigned int i=0; i<sz; i++) cout << " " << myvector[i];
  return 0;
}
```

if class, initialized with default constructor (annotation pointing to `vector<int> myvector (10);`)

vector of 10 uninitialized elements is allocated (annotation next to `vector<int> myvector (10);`)

```
9 8 7 6 5 4 3 2 1 0
```

Example
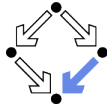
cplusplus.com: "C++ Reference".
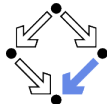
```cpp
#include <iostream>
#include <vector>
using namespace std;

int main () {
  vector<int> myvector; // default constructor creates empty vector

  myvector.push_back(10);
  while (myvector.back() != 0)
  {
    myvector.push_back ( myvector.back() -1 );
  }

  for (unsigned i=0; i<myvector.size() ; i++)
    cout << " " << myvector[i];
  return 0;
}

10 9 8 7 6 5 4 3 2 1 0
```

Example

cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main ()
{
  vector<int> myvector;          // vector with 10 elements
  for (unsigned int i=1;i<10;i++) myvector.push_back(i);

  myvector.resize(5);            // shrink to size 5
  myvector.resize(8,100);        // extend to size 8, fill with 100
  myvector.resize(12);           // extend to size 12, fill with 0

  for (unsigned int i=0;i<myvector.size();i++)
    cout << " " << myvector[i];
}

1 2 3 4 5 100 100 100 0 0 0 0
```

Class vector<bool>
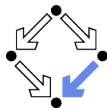
The vector class template has a special template specialization for the bool type. This specialization is provided to optimize for space allocation: In this template specialization, each element occupies only one bit (which is eight times less than the smallest type in C++: char).

The references to elements of a bool vector returned by the vector members are not references to bool objects, but a special member type which is a reference to a single bit, defined inside the vector<bool> class specialization as:

```
class vector<bool>::reference {
  friend class vector;
  reference();                              // no public constructor
public:
  ~reference();
  operator bool () const;                   // convert to bool
  reference& operator= ( const bool x );    // assign from bool
  reference& operator= ( const reference& x );  // assign from bit
  void flip();                              // flip bit value.
}
```

For a similar container class to contain bits, but with a fixed size, see bitset.

Class bitset

```cpp
#include <iostream>
#include <string>
#include <bitset>
using namespace std;

int main () {
  bitset<4> first (string("1001"));
  bitset<4> second (string("0011"));
  cout << (first^=second) << endl;       // 1010 (XOR,assign)
  cout << (first&=second) << endl;       // 0010 (AND,assign)
  cout << (first|=second) << endl;       // 0011 (OR,assign)
  cout << (~second) << endl;             // 1100 (NOT)
  cout << (second<<1) << endl;           // 0110 (SHL)
  cout << (second>>1) << endl;           // 0001 (SHR)
  cout << (first==second) << endl;       // false (0110==0011)
  cout << (first!=second) << endl;       // true  (0110!=0011)
  cout << (first&second) << endl;        // 0010
  cout << (first|second) << endl;        // 0111
  cout << (first^second) << endl;        // 0101
  return 0;
}
```

Class Template deque
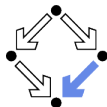
cplusplus.com: "C++ Reference".

```
template < class T, class Allocator = allocator<T> > class deque;
```

```
Deques may be implemented by specific libraries in different ways, but in all
cases they allow for the individual elements to be accessed through random
access iterators, with storage always handled automatically (expanding and
contracting as needed).

Deque sequences have the following properties:

    * Individual elements can be accessed by their position index.
    * Iteration over the elements can be performed in any order.
    * Elements can be efficiently added and removed from any of its ends
      (either the beginning or the end of the sequence).

Therefore they provide a similar functionality as the one provided by vectors,
but with efficient insertion and deletion of elements also at the beginning
of the sequence and not only at its end. On the drawback side, unlike
vectors, deques are not guaranteed to have all its elements in contiguous
storage locations, eliminating thus the possibility of safe access through
pointer arithmetics.
```

Example
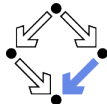
cplusplus.com: "C++ Reference".
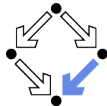
```cpp
#include <iostream>
#include <deque>
using namespace std;

int main ()
{
  deque<int> mydeque (2,100);      // two ints with a value of 100
  mydeque.push_front (200);
  mydeque.push_front (300);

  for (unsigned i=0; i<mydeque.size(); ++i)
    cout << " " << mydeque[i];
  return 0;
}

300 200 100 100
```

Example

cplusplus.com: "C++ Reference".

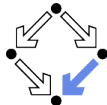```cpp
#include <iostream>
#include <deque>
using namespace std;

int main () {
  deque<int> mydeque;
  mydeque.push_back (100);
  mydeque.push_back (200);
  mydeque.push_back (300);

  while (!mydeque.empty()) {
    cout << " " << mydeque.front();
    mydeque.pop_front();
  }
  cout << "\nFinal size of mydeque is " << int(mydeque.size()) << endl;
  return 0;
}

100 200 300
Final size of mydeque is 0
```

Class Template `list`

cplusplus.com: "C++ Reference".

```
template < class T, class Allocator = allocator<T> > class list;
```

List containers are implemented as doubly-linked lists; doubly linked lists
can store each of the elements they contain in different and unrelated
storage locations. The ordering is kept by the association to each element of
a link to the element preceding it and a link to the element following it.
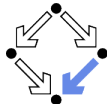
This provides the following advantages to list containers:

* Efficient insertion/removal of elements in the container (constant time).
* Efficient moving elements within the container (constant time).
* Iterating over the elements in forward or reverse order (linear time).

Compared to other base standard sequence containers (vectors and deques),
lists perform generally better in inserting, extracting and moving elements
in any position within the container, and therefore also in algorithms that
make intensive use of these, like sorting algorithms.

The main drawback of lists compared to these other sequence containers is that
they lack direct access to the elements by their position ...

Example

cplusplus.com: "C++ Reference".
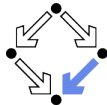
```cpp
// reversing vector
#include <iostream>
#include <list>
using namespace std;

int main ()
{
  list<int> mylist;
  for (int i=1; i<10; i++) mylist.push_back(i);

  mylist.reverse(); // additional member function of list

  for (list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
    cout << " " << *it;
  return 0;
}

9 8 7 6 5 4 3 2 1
```
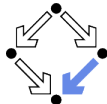
Iterators

cplusplus.com: "C++ Reference".

```
Header <iterator>
```

In C++, an iterator is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (at least, the increment (++) and dereference (*) operators).

The most obvious form of iterator is a pointer: A pointer can point to elements in an array, and can iterate through them using the increment operator (++). But other forms of iterators exist. For example, each container type (such as a vector) has a specific iterator type designed to iterate through its elements in an efficient way.

Notice that while a pointer is a form of iterator, not all iterators have the same functionality a pointer has; to distinguish between the requirements an iterator shall have for a specific algorithm, five iterator categories exist:

```
RandomAccess -> Bidirectional -> Forward -> Input
                                         -> Output
```

Iterator Categories

cplusplus.com: "C++ Reference".

* Input and output iterators are the most limited types of iterators, specialized in performing only sequential input or ouput operations.

* Forward iterators have all the functionality of input and output iterators, although they are limited to one direction in which to iterate through a range.

* Bidirectional iterators can be iterated through in both directions. All standard containers support at least bidirectional iterators types.

* Random access iterators implement all the functionalities of bidirectional iterators, plus, they have the ability to access ranges non-sequentially: offsets can be directly applied to these iterators without iterating through all the elements in between. This provides these iterators with the same functionality as standard pointers (pointers are iterators of this category).

Iterator Operations

category				characteristic	valid expressions
all categories				Can be copied and copy-constructed	`X b(a);` `b = a;`

				Can be incremented	`++a`
					`a++`
					`*a++`

Random Access	Bidirectional	Forward	Input	Accepts equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced to be the left side of an assignment operation	`*a = t` `*a++ = t`
				Can be default-constructed	`X a;` `X()`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`n - a`
					`a - b`

				Supports inequality comparisons (< and >) between iterators	`a < b` `a > b`

				Supports compound assignment operations +=, -=, <= and >=	`a += n`
					`a -= n`
					`a <= b`
					`a >= b`

				Supports offset dereference operator ([ ])	`a[n]`

Class <iterator>

cplusplus.com: "C++ Reference".

This is a base class template that can be used to derive iterator classes from it. It is not an iterator class and does not provide any of the functionality an iterator is expected to have.

This base class only provides some member types, which in fact are not required to be present in any iterator type (iterator types have no specific member requirements), but they might be useful, since they define the members needed for the default iterator_traits class template to generate the appropriate iterator_traits class automatically.

```
template <class Category, class T, class Distance = ptrdiff_t,
          class Pointer = T*, class Reference = T&>
  struct iterator {
    typedef T         value_type;
    typedef Distance  difference_type;
    typedef Pointer   pointer;
    typedef Reference reference;
    typedef Category  iterator_category;
  };
```

Category
    category to which the iterator belongs to.

Example

cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <iterator>
using namespace std;

class myiterator : public iterator<input_iterator_tag, int> {
  int* p;
public:
  myiterator(int* x) :p(x) {}
  myiterator(const myiterator& mit) : p(mit.p) {}
  myiterator& operator++() {++p;return *this;}
  myiterator& operator++(int) {p++;return *this;}
  bool operator==(const myiterator& rhs) {return p==rhs.p;}
  bool operator!=(const myiterator& rhs) {return p!=rhs.p;}
  int& operator*() {return *p;}
};

int main () {
  int numbers[]={10,20,30,40,50};
  myiterator beginning(numbers); myiterator end(numbers+5);
  for (myiterator it=beginning; it!=end; it++) cout << *it << " ";
}
```

Containers and Iterators
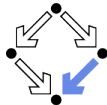
cplusplus.com: "C++ Reference".

```
{vector,deque,list}::{begin,end}                          public member function

      iterator begin ();
const_iterator begin () const;

Returns an iterator referring to the first element in the container.

      iterator end ();
const_iterator end () const;

Returns an iterator referring to the past-the-end element in the
vector container.

Both iterator and const_iterator are member types.

* In the vector and dequeue class template, these are random access iterators.
* In the list class template, these are bidirectional iterators.
```
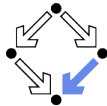
Example

cplusplus.com: "C++ Reference".

```
#include <iostream>
#include <vector>
using namespace std;

int main ()
{
  vector<int> myvector;
  for (int i=1; i<=5; i++) myvector.push_back(i);

  for ( vector<int>::iterator it=myvector.begin() ; it < myvector.end(); it++ )
    cout << " " << *it;
  return 0;
}

1 2 3 4 5
```

Advancing Iterators

cplusplus.com: "C++ Reference".

template function   header <iterator>

```
template <class InputIterator, class Distance>
  void advance (InputIterator& i, Distance n);
```

Advances the iterator i by n elements.

If i is a Random Access Iterator, the function uses once operator+ or
operator-, otherwise, the function uses repeatedly the increase or decrease
operator (operator++ or operator--) until n elements have been advanced.

Complexity

* Constant for random access iterators.
* Linear on n for other categories of iterators.

Example

cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

int main () {
  list<int> mylist;
  for (int i=0; i<10; i++) mylist.push_back (i*10);

  list<int>::iterator it = mylist.begin();

  advance (it,5);

  cout << "The sixth element in mylist is: " << *it << endl;
  return 0;
}

The sixth element in mylist is: 50
```

Container Construction with Iterators

cplusplus.com: "C++ Reference".

```
template <class InputIterator> vector
  ( InputIterator first, InputIterator last, const Allocator& = Allocator() );
template <class InputIterator> deque
  ( InputIterator first, InputIterator last, const Allocator& = Allocator() );
template < class InputIterator > list
  ( InputIterator first, InputIterator last, const Allocator& = Allocator() );
```

first is an element of the vector, last is the end + 1 element (which is not an element of the list)

```
Iteration constructor: Iterates between first and last, setting a copy of each
of the sequence of elements as the content of the container.
```

With input iterators, we can initialize a container by a range of elements
from another container.

Example

cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <list>
using namespace std;

int main () {
  list<int> first;                               // empty list of ints
  list<int> second (4,100);                      // four ints with value 100
  list<int> third (second.begin(),second.end()); // iterating through second
  list<int> fourth (third);                      // a copy of third

  // the iterator constructor can also be used to construct from arrays:
  int myints[] = {16,2,77,29};
  list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  for (list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
    cout << *it << " ";
}
```

16 2 77 29

Reverse Iterators
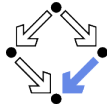
cplusplus.com: "C++ Reference".

```
template <class Iterator> class reverse_iterator;          header <iterator>
```

This class reverses the direction a bidirectional or random access iterator
iterates through a range.

A copy of the original iterator (the base iterator) is kept internally and
used to reflect all operations performed on the reverse_iterator: whenever
the reverse_iterator is incremented, its base iterator is decreased, and vice
versa. The base iterator can be obtained at any moment by calling member base.

Notice however that when an iterator is reversed, the reversed version does
not point to the same element in the range, but to the one preceding it. This
is so, in order to arrange for the past-the-end element of a range: An
iterator pointing to a past-the-end element in a range, when reversed, is
changed to point to the last element (not past it) of the range (this would
be the first element of the range if reversed). And if an iterator to the
first element in a range is reversed, the reversed iterator points to the
element before the first element (this would be the past-the-end element of
the range if reversed).

Example

cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int main () {
  vector<int> myvector;
  for (int i=0; i<10; i++) myvector.push_back(i);
  typedef vector<int>::iterator iter_int;
                                                    // ? 0 1 2 3 4 5 6 7 8 9 ?
  iter_int begin (myvector.begin());                //   ^
  iter_int end (myvector.end());                    //                       ^
  reverse_iterator<iter_int> rev_end (begin);       // ^
  reverse_iterator<iter_int> rev_iterator (end);    //                     ^

  for ( ; rev_iterator < rev_end ; ++rev_iterator) cout << *rev_iterator << " ";
  return 0;
}

9 8 7 6 5 4 3 2 1 0
```
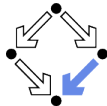
Containers and Reverse Iterators
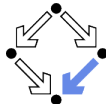
cplusplus.com: "C++ Reference".

```
{vector,deque,list}::{rbegin,rend}                              public member function

      reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;

Returns a reverse iterator referring to the last element in the container.

      reverse_iterator rend ();
const_reverse_iterator rend () const;

Both reverse_iterator and const_reverse_iterator are member types defined as
reverse_iterator<iterator> and reverse_iterator<const_iterator> respectively.

* In the vector and dequeue class template, these are
    reverse random access iterators.
* In the list class template, these are
    reverse bidirectional iterators.
```

Example
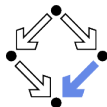
cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main ()
{
  vector<int> myvector;
  for (int i=1; i<=5; i++) myvector.push_back(i);

  vector<int>::reverse_iterator rit;
  for ( rit=myvector.rbegin() ; rit < myvector.rend(); ++rit )
    cout << " " << *rit;
  return 0;
}

5 4 3 2 1
```

Container Operations with Iterators

```
{vector,deque,list}::*                                          public member functions

template <class InputIterator> void assign (InputIterator f, InputIterator l);
void assign (size_type n, const T& u);
```

  Assigns new content to the container, dropping all the elements contained in
  the container object and replacing them by those specified by the parameters.

```
iterator insert (iterator p, const T& x );
void insert (iterator p, size_type n, const T& x);
template <class InputIterator>
void insert (iterator p, InputIterator f, InputIterator l);
```

  The container is extended by inserting new elements before position p. This
  effectively increases the container size by the amount of elements inserted.

```
iterator erase ( iterator position );
iterator erase ( iterator first, iterator last );
```

  Removes from the list container either a single element (position) or a
  range of elements ([first,last)). This effectively reduces the list size by
  the number of elements removed, calling each element's destructor before.

Example
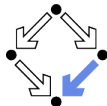
cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <list>
#include <vector>
using namespace std;

int main () {
  list<int> mylist;
  for (int i=1; i<=5; i++) mylist.push_back(i); // 1 2 3 4 5
  list<int>::iterator it = mylist.begin();
  ++it;          // it points now to number 2          ^
  mylist.insert (it,10);                              // 1 10 2 3 4 5
  // "it" still points to number 2                           ^
  mylist.insert (it,2,20);                            // 1 10 20 20 2 3 4 5
  --it;          // it points now to the second 20               ^
  vector<int> myvector (2,30);
  mylist.insert (it,myvector.begin(),myvector.end());
                                                      // 1 10 20 30 30 20 2 3 4 5
                                                      //              ^
  for (it=mylist.begin(); it!=mylist.end(); it++) cout << " " << *it;
  return 0;
}
```

Example

cplusplus.com: "C++ Reference".

```
...
int main () {
  list<unsigned int> mylist;
  list<unsigned int>::iterator it1,it2;
  for (unsigned int i=1; i<10; i++) mylist.push_back(i*10);
                               // 10 20 30 40 50 60 70 80 90
  it1 = it2 = mylist.begin(); // ^^
  advance (it2,6);            // ^                   ^
  ++it1;                      //    ^                ^
  it1 = mylist.erase (it1);   // 10 30 40 50 60 70 80 90
                              //    ^             ^
  it2 = mylist.erase (it2);   // 10 30 40 50 60 80 90
                              //    ^             ^
  ++it1;                      //       ^          ^
  --it2;                      //       ^       ^
  mylist.erase (it1,it2);     // 10 30 60 80 90
                              //       ^
  for (it1=mylist.begin(); it1!=mylist.end(); ++it1) cout << " " << *it1;
}
```

10 30 60 80 90

Stream Iterators

cplusplus.com: "C++ Reference".

```
template <class T, class charT=char, class traits=char_traits<charT>,
          class Distance = ptrdiff_t> class istream_iterator;
```

Istream iterators are a special input iterator class designed to read
successive elements from an input stream ... whenever operator++ is used on
the iterator, it extracts an element (with >>) from the stream.

A special value for this iterator exists: the end-of-stream; When an iterator
is set to this value has either reached the end of the stream (operator void*
applied to the stream returns false) or has been constructed using its default
constructor (without associating it with any basic_istream object).

```
template <class T, class charT=char, class traits=char_traits<charT>,
          class Distance = ptrdiff_t> class ostream_iterator;
```

Ostream iterators are a special output iterator class designed to write into
successive elements of an output stream ... whenever an assignment operator
is used on the ostream_iterator (even when dereferenced) it inserts a new
element into the stream. Optionally, a delimiter can be specified on
construction which is written to the stream after each element is inserted.

Example
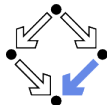
cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <iterator>
#include <string>
using namespace std;

int main () {
  istream_iterator<char> eos;        // end-of-range iterator
  istream_iterator<char> iit (cin);  // stdin iterator
  string mystring;
  cout << "Please, enter your name: ";
  while (iit!=eos && *iit!='\n') {
    mystring += *iit;
    iit++;
  }
  cout << "Your name is " << mystring << ".\n";
  return 0;
}
```

type of elements that are read or written

inputstream

```
Please, enter your name: HAL 9000
Your name is HAL 9000.
```

Example

cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int main () {
  vector<int> myvector;
  for (int i=1; i<10; ++i) myvector.push_back(i*10);

  ostream_iterator<int> out_it (cout,", ");
  for (vector<int>::iterator it = myvector.begin(); it != myvector.end(); it++)
  {
    *out_it = *it;
    out_it++;        printing as soon as *out_it gets assigned a value
  }
  return 0;
}
```
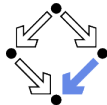
10, 20, 30, 40, 50, 60, 70, 80, 90,
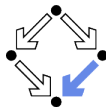
Adaptors

cplusplus.com: "C++ Reference".

stack, queue and priority_queue are implemented as container adaptors. Container adaptors are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as deque or list) to handle the elements. The underlying container is encapsulated in such a way that its elements are accessed by the members of the container class independently of the underlying container class used.

			Container Adaptors		
Headers			**<stack>**	**<queue>**	
Members			**stack**	**queue**	**priority_queue**
	constructor	*	constructor	constructor	constructor
capacity	size	O(1)	size	size	size
	empty	O(1)	empty	empty	empty
element access	front	O(1)		front	
	back	O(1)		back	
	top	O(1)	top		top
modifiers	push	O(1)	push	push	push
	pop	O(1)	pop	pop	pop

Class Template `stack`

cplusplus.com: "C++ Reference".

```
template < class T, class Container = deque<T> > class stack;
```

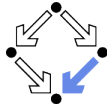Stacks are a type of container adaptors, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from the end of the container. ... Elements are pushed/popped from the "back" of the specific container, which is known as the top of the stack.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The only requirement is that it supports the following operations:

```
    * back()
    * push_back()
    * pop_back()
```

Therefore, the standard container class templates vector, deque and list can be used. By default, if no container class is specified for a particular stack class, the standard container class template deque is used.

Example
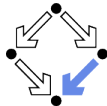
cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <stack>
using namespace std;

int main () {
  stack<int> mystack;
  for (int i=0; i<5; ++i) mystack.push(i);

  while (!mystack.empty()) {
     cout << " " << mystack.top();
     mystack.pop();
  }
  return 0;
}

4 3 2 1 0
```

Class Template queue

cplusplus.com: "C++ Reference".

```
template < class T, class Container = deque<T> > class queue;
```

queues are a type of container adaptors, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other. ... Elements are pushed into the "back" of the specific container and popped from its "front".

The underlying container may be one of the standard container class template or some other specifically designed container class. The only requirement is that it supports the following operations:

    * front()
    * back()
    * push_back()
    * pop_front()

Therefore, the standard container class templates deque and list can be used. By default, if no container class is specified for a particular queue class, the standard container class template deque is used.

Example

cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main () {
  queue<int> myqueue;
  int myint;

  do {                                 // enter integers (0 to end)
    cin >> myint;
    myqueue.push (myint);
  } while (myint);

  while (!myqueue.empty()) {          // print integers (in the same order
    cout << " " << myqueue.front(); //    in which they were entered)
    myqueue.pop();
  }
  return 0;
}
```

Class Template priority_queue

cplusplus.com: "C++ Reference".

```
template < class T, class Container = vector<T>,
 class Compare = less<typename Container::value_type> >  class priority_queue;
```

Priority queues are a type of container adaptors, specifically designed such
that its first element is always the greatest of the elements it contains,
according to some strict weak ordering condition. This context is similar to
a heap where only the max heap element can be retrieved and elements can be
inserted indefinitely. ... Elements are popped from the "back" of the
specific container, which is known as the top of the priority queue.

The underlying container may be any ... container class. The only requirement
is that it must be accessible through random access iterators and it must
support the following operations:

    * front()
    * push_back()
    * pop_back()

Therefore, the standard container class templates vector and deque can be
used. By default ... the standard container class template vector is used.
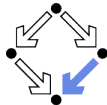
Class Template `priority_queue`

cplusplus.com: "C++ Reference".

```
template < class T, class Container = vector<T>,
 class Compare = less<typename Container::value_type> > class priority_queue;

template <class T> struct less : binary_function <T,T,bool> {
  bool operator() (const T& x, const T& y) const {return x<y;}
};
```

Compare is a class such that the expression comp(a,b), where comp is an object
of this class and a and b are elements of the container, returns true if a is
to be placed earlier than b in a strict weak ordering operation. This can
either be a class implementing a function call operator or a pointer to a
function. This defaults to less<T>, which returns the same as applying the
less-than operator (a<b).

The priority_queue object uses this expression when an element is inserted or
removed from it (using push or pop, respectively) to grant that the element
popped is always the greater in the priority queue.

Example
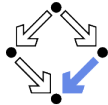
cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main () {
  priority_queue<int> mypq;
  mypq.push(30);
  mypq.push(100);
  mypq.push(25);
  mypq.push(40);

  while (!mypq.empty()) {
     cout << " " << mypq.top();
     mypq.pop();
  }
  return 0;
}

100 40 30 25
```
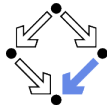
Associative Containers

- Elements organized for fast access by keys
  - `<set>`: class templates `set` and `multiset`.
    - (Multi)sets of elements (elements themselves are the keys).
  - `<map>`: class templates `map` and `multimap`.
    - Mappings of keys to (sets of) values.
- Common operations
  - Most operations of sequence containers.
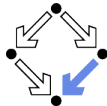
    Except sequential access, random access, modification of begin and end of container.
  - Observers: `key_comp`, `value_comp`.
  - Miscellaneous operations: `find`, `count`, `lower_bound`, `upper_bound`, `equal_range`.

Chosen according to required mathematical functionality.

Class Templates set **and** multiset

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> > class (multi)set;
```

Sets are a kind of associative containers that stores unique elements, and in
which the elements themselves are the keys. Internally, the elements in a
set are always sorted from lower to higher following a specific strict weak
ordering criterion set on container construction. Sets are typically
implemented as binary search trees. Therefore, the main characteristics of
set as an associative container are:

* Unique element values: no two elements in the set can compare equal to
  each other. For a similar associative container allowing for multiple
  equivalent elements, see multiset.

* The element value is the key itself. For a similar associative container
  where elements are accessed using a key, but map to a value different than
  this key, see map.

* Elements follow a strict weak ordering at all times. Unordered associative
  arrays, like unordered_set, are available in implementations following TR1.

Multisets ... allow for multiple keys with equal values.

Example

cplusplus.com: "C++ Reference".
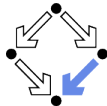
```cpp
#include <iostream>
#include <set>
using namespace std;

int main () {
  int myints[] = {75,23,65,23,42,13}; // 23 occurs twice
  set<int> myset (myints,myints+6);
  myset.insert(23);                    // once more 23 is inserted

  for ( set<int>::iterator it=myset.begin() ; it != myset.end(); it++ )
    cout << " " << *it;
  return 0;
}

13 23 42 65 75
```

Example
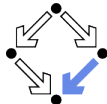
cplusplus.com: "C++ Reference".

```
#include <iostream>
#include <set>
using namespace std;

int main () {
  int myints[] = {75,23,65,23,42,13}; // 23 occurs twice
  multiset<int> myset (myints,myints+6);
  myset.insert(23);                    // once more 23 is inserted

  for ( multiset<int>::iterator it=myset.begin() ; it != myset.end(); it++ )
    cout << " " << *it;
  return 0;
}

13 23 23 23 42 65 75
```

Class Templates `map` and `multimap`

```
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key,T> > > class (multi)map;
```

Maps are a kind of associative containers that stores elements formed by the
combination of a key value and a mapped value. In a map, the key value is
generally used to uniquely identify the element, while the mapped value is some
sort of value associated to this key. Types of key and mapped value may differ.
Internally, the elements in the map are sorted from lower to higher key value
following a specific strict weak ordering criterion set on construction.
Therefore, the main characteristics of a map as an associative container are:

* Unique key values: no two elements in the map have keys that compare equal
  to each other. For a similar associative container allowing for multiple
  elements with equivalent keys, see multimap.
* Each element is composed of a key and a mapped value. For a simpler
  associative container where the element value itself is its key, see set.
* Elements follow a strict weak ordering at all times.

Maps ... implement the direct access operator (operator[]) which allows for
direct access of the mapped value.

Multimaps ... allow different elements to have the same key value.

Example

cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <map>
using namespace std;

int main () {
  map<char,int> mymap;
  map<char,int>::iterator it;

  mymap['b'] = 100;
  mymap['a'] = 200;
  mymap['c'] = 300;

  for ( it=mymap.begin() ; it != mymap.end(); it++ )
    cout << (*it).first << " => " << (*it).second << endl;
  return 0;
}

a => 200
b => 100
c => 300
```
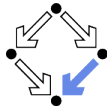
Example

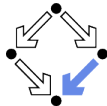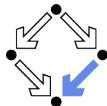cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <map>
using namespace std;

int main () {
  multimap<char,int> mymultimap;
  multimap<char,int>::iterator it;

  mymultimap.insert (pair<char,int>('a',10));
  mymultimap.insert (pair<char,int>('b',20));
  mymultimap.insert (pair<char,int>('b',150));

  for ( it=mymultimap.begin() ; it != mymultimap.end(); it++ )
    cout << (*it).first << " => " << (*it).second << endl;
  return 0;
}

a => 10
b => 20
b => 150
```

Member Function `find`

```
iterator find ( const key_type& x ) const;

Searches the container for an element with a value of x and returns an
iterator to it if found, otherwise it returns an iterator to the
element past the end of the container.

#include <iostream>
#include <map>
using namespace std;

int main ()
{
  map<char,int> mymap;
  map<char,int>::iterator it;

  mymap['a']=50; mymap['b']=100;

  it = mymap.find('b');
  if (it != mymap.end()) mymap.erase (it);
  cout << "a => " << mymap.find('a')->second << '\n';
  return 0;
}
```

Member Function `count`

```
size_type count ( const key_type& x ) const;

Searches the container for an element with a key of x and returns the number
of times the element appears in the container.

#include <iostream>
#include <set>
using namespace std;

int main () {
  set<int> myset;
  for (int i=1; i<5; i++) myset.insert(i*3);     // set: 3 6 9 12
  for (int i=0;i<10; i++) {
    cout << i;
    if (myset.count(i)>0)
      cout << " is an element of myset.\n";
    else cout << " is not an element of myset.\n";
  }
  return 0;
}

0 is not an element of myset. ...
```

Member Functions lower/upper_bound

iterator lower/upper_bound (const key_type& x);

lower_bound returns an iterator pointing to the first element in the container whose key does not compare less than x (using the container's comparison object), i.e. it is either equal or greater. upper_bound returns an iterator pointing to the first element in the container whose key compares greater than x.

```cpp
#include <iostream>
#include <map>
using namespace std;
int main () {
  map<char,int> mymap;
  mymap['a']=20; mymap['b']=40; mymap['c']=60; mymap['d']=80; mymap['e']=100;
  map<char,int>::iterator itlow=mymap.lower_bound ('b'); // itlow points to b
  map<char,int>::iterator itup=mymap.upper_bound ('d');  // itup  points to e
  mymap.erase(itlow,itup);                               // erases [itlow,itup)
  for (map<char,int>::iterator it=mymap.begin() ; it != mymap.end(); it++ )
    cout << (*it).first << " => " << (*it).second << " ";
  return 0;
}
```

a => 20 e => 100

Member Function equal_range
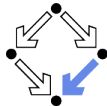
```
pair<iterator,iterator> equal_range ( const key_type& x ) const;
```

Returns the bounds of a range that includes all the elements in the container
with a key that compares equal to x. If x does not match any key in the
container, the range has a length of zero, with both iterators pointing to
the nearest value greater than x, if any, or to the element past the end of
the container if x is greater than all the elements in the container.

```
#include <iostream>
#include <set>
using namespace std;
int main () {
  int myints[]= {77,30,16,2,30,30};
  multiset<int> mymultiset (myints, myints+6); // 2 16 30 30 30 77
  pair<multiset<int>::iterator,multiset<int>::iterator>
    ret = mymultiset.equal_range(30);            //         ^         ^
  for (multiset<int>::iterator it=ret.first; it!=ret.second; ++it)
    cout << " " << *it;
  return 0;
}

30 30 30
```
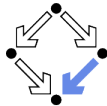
1. **General**

2. **Sequence Containers**

3. **Iterators**

4. **Adaptors**

5. **Associative Containers**

6. **Algorithms**

Header <algorithm>
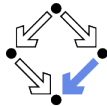
The standard library comes with a rich set of (so-called) algorithms.

- Algorithm: a template function operating on a range of elements.
    - A range is a sequence of objects accessible by iterators/pointers.
        - Iterator type is argument of function template.
        - Iterators of this type are arguments to function.
            ```
            template<class InIter, class T>
            InIter find(InIter first, InIter last, const T& value);
                            included                excluded
            ```
- Works on any object that provides suitable iterators/pointers.
    - Containers, plain arrays, streams.
- Algorithms and containers are mostly orthogonal.
    - New algorithms can be written without modifying containers.
        - Algorithms will be automatically applicable on containers.
    - New containers can be developed without modifying algorithms.
        - Containers can be immediately processed by algorithms.

When processing containers, remember the already available algorithms.

Header `<functional>`
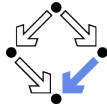
Many algorithms operate on function objects.

- Function object: any object that provides function application.
  - Any function and any object that provides `operator()`.
    ```
    struct F { int operator()(int a) {return a;} };
    F f; int x = f(0); // function-like syntax with object f
    ```
- `<functional>` provides a collection of function object templates.
  - Unary function objects inherit from `unary_function`.
    ```
    template <class Arg, class Result>
      struct unary_function {
        typedef Arg argument_type;
        typedef Result result_type;
      };
    ```
  - Binary function objects inherit from `binary_function`.
    ```
    template <class Arg1, class Arg2, class Result>
      struct binary_function {
        typedef Arg1 first_argument_type;
        typedef Arg2 second_argument_type;
        typedef Result result_type;
      };
    ```

Example
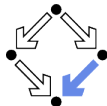
```cpp
#include <iostream>
#include <functional>
using namespace std;

struct Compare : public binary_function<int,int,bool> {
  bool operator() (int a, int b) {return (a==b);}
};

int main () {
  Compare::first_argument_type input1;
  Compare::second_argument_type input2;
  cout << "Please enter first number: ";  cin >> input1;
  cout << "Please enter second number: "; cin >> input2;
  cout << "Numbers " << input1 << " and " << input2;
  Compare Compare_object;
  Compare::result_type result = Compare_object (input1,input2);
  if (result)
    cout << " are equal.\n";
  else
    cout << " are not equal.\n";
  return 0;
}
```

Function Objects
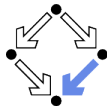
cplusplus.com: "C++ Reference".

plus	Addition function object class
minus	Subtraction function object class
multiplies	Multiplication function object class
divides	Division function object class
modulus	Modulus function object class
negate	Negative function object class
equal_to	Function object class for equality comparison
not_equal_to	Function object class for non-equality comparison
greater	Function object class for greater-than inequality comparison
less	Function object class for less-than inequality comparison
greater_equal	Function object class for greater-than-or-equal-to comparison
less_equal	Function object class for less-than-or-equal-to comparison
logical_and	Logical AND function object class
logical_or	Logical OR function object class
logical_not	Logical NOT function object class
...	

Example
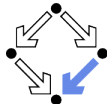
cplusplus.com: "C++ Reference".

```cpp
template <class T> struct less : binary_function <T,T,bool> {
  bool operator() (const T& x, const T& y) const
    {return x<y;}
};
```

Objects of this class can be used with some standard algorithms such as sort, merge or lower_bound.

```cpp
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

int main () {
  int foo[]={10,20,5,15,25};
  sort (foo, foo+5, less<int>() ); // 5 10 15 20 25
  return 0;
}
```

Non-Modifying Sequence Operations

cplusplus.com: "C++ Reference".

for_each	Apply function to range
find	Find value in range
find_if	Find element in range
find_end	Find last subsequence in range
find_first_of	Find element from set in range
adjacent_find	Find equal adjacent elements in range
count	Count appearances of value in range
count_if	Return number of elements in range satisfying condition
mismatch	Return first position where two ranges differ
equal	Test whether the elements in two ranges are equal
search	Find subsequence in range
search_n	Find succession of equal values in range

These operations do not modify the contents of the sequence.

Algorithm `for_each`

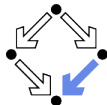clusplus.com: "C++ Reference".

```
template <class InputIterator, class Function>
Function for_each (InputIterator first, InputIterator last, Function f);

Applies function f to each of the elements in the range [first,last).

The behavior of this template function is equivalent to:

template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f)
  {
    while ( first!=last ) f(*first++);
    return f;
  }
```

Example

clusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

void myfunction (int i) { cout << " " << i; }
struct myclass { void operator() (int i) {cout << " " << i;} };

int main () {
  vector<int> myvector;
  myvector.push_back(10); myvector.push_back(20); myvector.push_back(30);

  for_each (myvector.begin(), myvector.end(), myfunction);

  myclass myobject;
  for_each (myvector.begin(), myvector.end(), myobject);
  return 0;
}
```
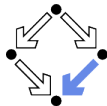
10 20 30 10 20 30

Algorithm find

clusplus.com: "C++ Reference".

```
template <class InputIterator, class T>
InputIterator find ( InputIterator first, InputIterator last, const T& value );
```

Returns an iterator to the first element in the range [first,last) that
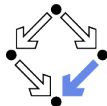compares equal to value, or last if not found.

The behavior of this function template is equivalent to:

```
template<class InputIterator, class T>
InputIterator find ( InputIterator first, InputIterator last, const T& value )
  {
    for ( ;first!=last; first++) if ( *first==value ) break;
    return first;
  }
```

Complexity

At most, performs as many comparisons as the number of elements in the
range [first,last).

Example

clusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int myints[] = { 10, 20, 30 ,40 };
  int *p = find(myints,myints+4,30);
  ++p;
  cout << "The element following 30 is " << *p << endl;

  vector<int> myvector (myints,myints+4);
  vector<int>::iterator it = find (myvector.begin(), myvector.end(), 30);
  ++it;
  cout << "The element following 30 is " << *it << endl;
  return 0;
}

The element following 30 is 40
The element following 30 is 40
```
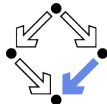
Algorithm `find_if`

clusplus.com: "C++ Reference".

```
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

Returns an iterator to the first element in the range [first,last) for which
applying pred to it, is true.

The behavior of this function template is equivalent to:

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred)
  {
    for ( ; first!=last ; first++ ) if ( pred(*first) ) break;
    return first;
  }
```

Example
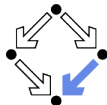
clusplus.com: "C++ Reference".
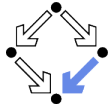
```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool IsOdd (int i) { return ((i%2)==1); }

int main () {
  vector<int> myvector;
  myvector.push_back(10);
  myvector.push_back(25);
  myvector.push_back(40);
  myvector.push_back(55);

  vector<int>::iterator it = find_if (myvector.begin(), myvector.end(), IsOdd);
  cout << "The first odd value is " << *it << endl;
  return 0;
}

The first odd value is 25
```
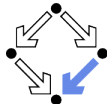
Modifying Sequence Operations

copy	Copy range of elements
copy_backward	Copy range of elements backwards
swap	Exchange values of two objects
swap_ranges	Exchange values of two ranges
iter_swap	Exchange values of objects pointed by two iterators
transform	Apply function to range
replace Replace	value in range
replace_if	Replace values in range
replace_copy	Copy range replacing value
replace_copy_if	Copy range replacing value
fill	Fill range with value
fill_n	Fill sequence with value
generate	Generate values for range with function
generate_n	Generate values for sequence with function
remove	Remove value from range
remove_if	Remove elements from range
remove_copy	Copy range removing value
remove_copy_if	Copy range removing values
unique	Remove consecutive duplicates in range
unique_copy	Copy range removing duplicates
reverse	Reverse range
reverse_copy	Copy range reversed
rotate	Rotate elements in range
rotate_copy	Copy rotated range
random_shuffle	Rearrange elements in range randomly
partition	Partition range in two
stable_partition	Parition range in two - stable ordering

These operations modify the contents of the sequence.

Algorithm copy

```
template <class InIter, class OutIter>
OutIter copy ( InIter first, InIter last, OutIter result );

Copies the elements in the range [first,last) into a range beginning at result.
Returns an iterator to the last element in the destination range.

The behavior of this function template is equivalent to:

template<class InIter, class OutIter>
OutIter copy ( InIter first, InIter last, OutIter result )
{
  while (first!=last) *result++ = *first++;
  return result;
}

If both ranges overlap in such a way that result points to an element in the
range [first,last), the function copy_backward should be used instead.
```
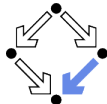
Example

clusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int myints[]={10,20,30,40,50,60,70};
  vector<int> myvector;

  myvector.resize(7);   // allocate space for 7 elements
  copy ( myints, myints+7, myvector.begin() );

  for (vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    cout << " " << *it;
  return 0;
}

10 20 30 40 50 60 70
```
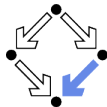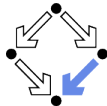
Algorithm transform

```
template < class InIter, class OutIter, class UnaryOp >
OutIter transform ( InIter first1, InIter last1, OutIter result, UnaryOp op );

template < class InIter1, class InIter2, class OutIter, class BinaryOp >
OutIter transform ( InIter1 first1, InIter1 last1,
                    InIter2 first2, OutIter result, BinaryOp binary_op );
```

The first version applies op to all the elements in the input range
([first1,last1)) and stores each returned value in the range beginning at
result. The second version uses as argument for each call to binary_op one
element from the first input range ([first1,last1)) and one element from the
second input range (beginning at first2). The behavior of this function
template is equivalent to:

```
template < class InIter, class OutIter, class UnaryOperator >
OutIter transform ( InIter first1, InIter last1,
                            OutIter result, UnaryOperator op )
{
  while (first1 != last1)
    *result++ = op(*first1++);  // or: *result++=binary_op(*first1++,*first2++);
  return result;
}
```

Example

clusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int op_increase (int i) { return ++i; }
int op_sum (int i, int j) { return i+j; }

int main () {
  vector<int> first, second;
  for (int i=1; i<6; i++) first.push_back (i*10); // first: 10 20 30 40 50
  second.resize(first.size());                     // allocate space

  transform (first.begin(), first.end(), second.begin(), op_increase);
                                                    // second: 11 21 31 41 51

  transform (first.begin(), first.end(), second.begin(), first.begin(), op_sum);
                                                    // first:  21 41 61 81 101
  return 0;
}
```
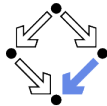
Algorithm generate

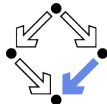clusplus.com: "C++ Reference".

```
template <class ForwardIterator, class Generator>
void generate ( ForwardIterator first, ForwardIterator last, Generator gen );

Sets the value of the elements in the range [first,last) to the value returned
by successive calls to gen.

The behavior of this function template is equivalent to:

template <class ForwardIterator, class Generator>
void generate ( ForwardIterator first, ForwardIterator last, Generator gen )
{
  while (first != last)  *first++ = gen();
}
```

Example

clusplus.com: "C++ Reference".
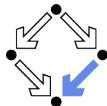
```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <cstdlib>
using namespace std;

int RandomNumber () { return (rand()%100); }
struct c_unique { int c; c_unique() {c=0;} int operator()() {return ++c;} };

int main () {
  vector<int> myvector (8);  // e.g.: 57 87 76 66 85 54 17 15
  generate (myvector.begin(), myvector.end(), RandomNumber);

  c_unique UniqueNumber;     // 1 2 3 4 5 6 7 8
  generate (myvector.begin(), myvector.end(), UniqueNumber);
  return 0;
}
```

Sorting and Operations on Sorted Ranges

cplusplus.com: "C++ Reference".

sort	Sort elements in range
stable_sort	Sort elements preserving order of equivalents
partial_sort	Partially Sort elements in range
partial_sort_copy	Copy and partially sort range
nth_element	Sort element in range
lower_bound	Return iterator to lower bound
upper_bound	Return iterator to upper bound
equal_range	Get subrange of equal elements
binary_search	Test if value exists in sorted array
merge	Merge sorted ranges
inplace_merge	Merge consecutive sorted ranges
includes	Test whether sorted range includes another one
set_union	Union of two sorted ranges
set_intersection	Intersection of two sorted ranges
set_difference	Difference of two sorted ranges
set_symmetric_difference	Symmetric difference of two sorted ranges

Algorithm sort
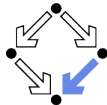
clusplus.com: "C++ Reference".

```
template <class RandomAccessIterator>
void sort (RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

```
Sorts the elements in the range [first,last) into ascending order. The
elements are compared using operator< for the first version, and comp for the
second. Elements that would compare equal to each other are not guaranteed to
keep their original relative order.
```

```
Complexity
```

```
Approximately N*logN comparisons on average (where N is last-first).
```

```
In the worst case, up to N2, depending on specific sorting algorithm used by
library implementation.
```

Example
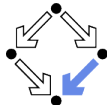
cplusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i,int j) { return (i<j); }
struct myclass { bool operator() (int i,int j) { return (i<j);} };

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  vector<int> myvector (myints, myints+8);      // 32 71 12 45 26 80 53 33

  sort (myvector.begin(), myvector.begin()+4);  // (12 32 45 71) 26 80 53 33

                                                // 12 32 45 71 (26 33 53 80)
  sort (myvector.begin()+4, myvector.end(), myfunction);

  myclass myobject;                             // (12 26 32 33 45 53 71 80)
  sort (myvector.begin(), myvector.end(), myobject);
  return 0;
}
```
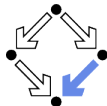
Algorithm `lower_bound`

clusplus.com: "C++ Reference".

```
template <class FwdIterator, class T> FwdIterator lower_bound
  (FwdIterator first, FwdIterator last, const T& val);
template <class FwdIterator, class T, class Compare> FwdIterator lower_bound
  (FwdIterator first, FwdIterator last, const T& val, Compare comp);
```

```
Returns an iterator pointing to the first element in the range [first,last)
which does not compare less than val.
```

```
The elements are compared using operator< for the first version, and comp
for the second. The elements in the range shall already be sorted according
to this same criterion (operator< or comp), or at least partitioned with
respect to val. The function optimizes the number of comparisons performed
by comparing non-consecutive elements of the sorted range, which is
specially efficient for random-access iterators.
```

```
Complexity
On average, logarithmic in the distance between first and last: Performs
approximately log2(N)+1 element comparisons (where N is this distance). On
non-random-access iterators, the iterator advances produce themselves an
additional linear complexity in N on average.
```

Example
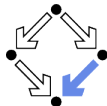
clusplus.com: "C++ Reference".

```cpp
#include <iostream>     // std::cout
#include <algorithm>    // std::lower_bound, std::upper_bound, std::sort
#include <vector>       // std::vector
using namespace std;

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20
  sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30

  vector<int>::iterator low,up;
  low=lower_bound (v.begin(), v.end(), 20); //             ^
  up=upper_bound (v.begin(), v.end(), 20);  //                      ^

  cout << "lower_bound at position " << (low- v.begin()) << '\n';
  cout << "upper_bound at position " << (up - v.begin()) << '\n';
  return 0;
}

lower_bound at position 3
upper_bound at position 6
```
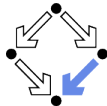
Algorithm merge

clusplus.com: "C++ Reference".

```
template <class InIter1, class InIter2, class OutIter>
OutIter merge ( InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2,
                OutIter result [, Compare comp] );
```

Combines the elements in the sorted ranges [first1,last1) and [first2,last2),
into a new range beginning at result with its elements sorted. The comparison
for sorting uses either operator< for the first version, or comp for the second.
For the function to yield the expected result, the elements in the both ranges
shall already be ordered according to the same strict weak ordering criterion
(operator< or comp). The resulting range is also sorted according to it.

The behavior of this function template is equivalent to:

```
template <class InIter1, class InIter2, class OutIter>
OutIter merge ( InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2,
                OutIter result ) {
  while (true) {
    *result++ = (*first2<*first1)? *first2++ : *first1++;
    if (first1==last1) return copy(first2,last2,result);
    if (first2==last2) return copy(first1,last1,result); } }
```

Example
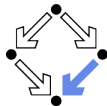
clusplus.com: "C++ Reference".

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);

  sort (first,first+5);
  sort (second,second+5);
  merge (first,first+5,second,second+5,v.begin());

  for (vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
    cout << " " << *it;
  return 0;
}

 5 10 10 15 20 20 25 30 40 50
```