

A quick overview of the Standard Template Library

Advanced Programming and Algorithmic Design

Alberto Sartori

November 27, 2018

Outline

- 1 The C++ standard library
- 2 Iterators
- 3 Containers
- 4 Algorithms
- 5 Function objects

1 The C++ standard library

2 Iterators

3 Containers

4 Algorithms

5 Function objects

What is the standard library?

The standard library is the set of components specified by the ISO C++ standard (~ 800 dense pages) and shipped with identical behavior (modulo performance) by every C++ implementation.

The C++ Programming Language

Part IV: The Standard Library

857

30.	Standard Library Summary	859
31.	STL Containers	885
32.	STL Algorithms	927
33.	STL Iterators	953
34.	Memory and Resources	973
35.	Utilities	1009
36.	Strings	1033
37.	Regular Expressions	1051
38.	I/O Streams	1073
39.	Locales	1109
40.	Numerics	1159
41.	Concurrency	1191
42.	Threads and Tasks	1209
43.	The C Standard Library	1253
44.	Compatibility	1267

The header files

Containers		
<code><vector></code>	One-dimensional resizable array	§31.4.2
<code><deque></code>	Double-ended queue	§31.4.2
<code><forward_list></code>	Singly-linked list	§31.4.2
<code><list></code>	Doubly-linked list	§31.4.2
<code><map></code>	Associative array	§31.4.3
<code><set></code>	Set	§31.4.3
<code><unordered_map></code>	Hashed associative array	§31.4.3.2
<code><unordered_set></code>	Hashed set	§31.4.3.2
<code><queue></code>	Queue	§31.5.2
<code><stack></code>	Stack	§31.5.1
<code><array></code>	One-dimensional fixed-size array	§34.2.1
<code><bitset></code>	Array of bool	§34.2.2

The header files

General Utilities		
<code><utility></code>	Operators and pairs	§35.5, §34.2.4.1
<code><tuple></code>	Tuples	§34.2.4.2
<code><type_traits></code>	Type traits	§35.4.1
<code><typeindex></code>	Use a <code>type_info</code> as a key or a hash code	§35.5.4
<code><functional></code>	Function objects	§33.4
<code><memory></code>	Resource management pointers	§34.3
<code><scoped_allocator></code>	Scoped allocators	§34.4.4
<code><ratio></code>	Compile-time rational arithmetic	§35.3
<code><chrono></code>	Time utilities	§35.2
<code><ctime></code>	C-style date and time	§43.6
<code><iterator></code>	Iterators and iterator support	§33.1

The header files

Algorithms		
<code><algorithm></code>	General algorithms	§32.2
<code><cstdlib></code>	<code>bsearch()</code> , <code>qsort()</code>	§43.7

The header files

Diagnostics		
<code><exception></code>	Exception class	§30.4.1.1
<code><stdexcept></code>	Standard exceptions	§30.4.1.1
<code><cassert></code>	Assert macro	§30.4.2
<code><cerrno></code>	C-style error handling	§13.1.2
<code><system_error></code>	System error support	§30.4.3

The header files

Strings and Characters		
<code><string></code>	String of T	Chapter 36
<code><cctype></code>	Character classification	§36.2.1
<code><cwctype></code>	Wide-character classification	§36.2.1
<code><cstring></code>	C-style string functions	§43.4
<code><cwchar></code>	C-style wide-character string functions	§36.2.1
<code><cstdlib></code>	C-style allocation functions	§43.5
<code><cuchar></code>	C-style multibyte characters	
<code><regex></code>	Regular expression matching	Chapter 37

The header files

Input/Output		
<code><iosfwd></code>	Forward declarations of I/O facilities	§38.1
<code><iostream></code>	Standard <code>iostream</code> objects and operations	§38.1
<code><ios></code>	<code>iostream</code> bases	§38.4.4
<code><streambuf></code>	Stream buffers	§38.6
<code><istream></code>	Input stream template	§38.4.1
<code><ostream></code>	Output stream template	§38.4.2
<code><iomanip></code>	Manipulators	§38.4.5.2
<code><sstream></code>	Streams to/from strings	§38.2.2
<code><cctype></code>	Character classification functions	§36.2.1
<code><fstream></code>	Streams to/from files	§38.2.1
<code><cstdio></code>	<code>printf()</code> family of I/O	§43.3
<code><cwchar></code>	<code>printf()</code> -style I/O of wide characters	§43.3

The header files

Localization		
<code><locale></code>	Represent cultural differences	Chapter 39
<code><locale></code>	Represent cultural differences C-style	
<code><codecvt></code>	Code conversion facets	§39.4.6

The header files

Language Support		
<code><limits></code>	Numeric limits	§40.2
<code><climits></code>	C-style numeric scalar-limit macros	§40.2
<code><cfloat></code>	C-style numeric floating-point limit macros	§40.2
<code><stdint></code>	Standard integer type names	§43.7
<code><new></code>	Dynamic memory management	§11.2.3
<code><typeinfo></code>	Run-time type identification support	§22.5
<code><exception></code>	Exception-handling support	§30.4.1.1
<code><initializer_list></code>	initializer_list	§30.3.1
<code><cstdlib></code>	C library language support	§10.3.1
<code><cstdlibarg></code>	Variable-length function argument lists	§12.2.4
<code><csetjmp></code>	C-style stack unwinding	
<code><stdlib></code>	Program termination	§15.4.3
<code><ctime></code>	System clock	§43.6
<code><csignal></code>	C-style signal handling	

The header files

Numerics		
<code><complex></code>	Complex numbers and operations	§40.4
<code><valarray></code>	Numeric vectors and operations	§40.5
<code><numeric></code>	Generalized numeric operations	§40.6
<code><cmath></code>	Standard mathematical functions	§40.3
<code><cstdlib></code>	C-style random numbers	§40.7
<code><random></code>	Random number generators	§40.7

The header files

Concurrency		
<code><atomic></code>	Atomic types and operations	§41.3
<code><condition_variable></code>	Waiting for an action	§42.3.4
<code><future></code>	Asynchronous task	§42.4.4
<code><mutex></code>	Mutual exclusion classes	§42.3.1
<code><thread></code>	Threads	§42.2

The header files

C Compatibility		
<code><cstdint></code>	Aliases for common integer types	§43.7
<code><stdbool></code>	C <code>bool</code>	
<code><complex></code>	<code><complex></code>	
<code><cfenv></code>	Floating-point environment	
<code><stdalign></code>	C alignment	
<code><ctgmath></code>	C “type generic math”: <code><complex></code> and <code><cmath></code>	

The header files

Library Supported Language Features		
<code><new></code>	<code>new</code> and <code>delete</code>	§11.2
<code><typeinfo></code>	<code>typeid()</code> and <code>type_info</code>	§22.5
<code><iterator></code>	Range- <code>for</code>	§30.3.2
<code><initializer_list></code>	<code>initializer_list</code>	§30.3.1

We will focus on the STL



- 1 The C++ standard library
- 2 **Iterators**
- 3 Containers
- 4 Algorithms
- 5 Function objects

What is an Iterator?

Design pattern

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

A generalization of a pointer

- indirect access (`operator*()`, `operator->()`)
- operations for moving to point to a new element (`operator++()`, `operator--()`)

Iterators in the STL

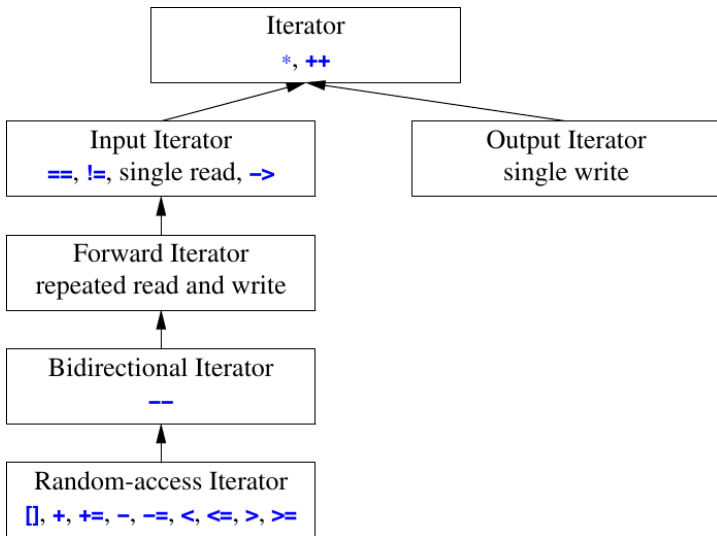
Their role

- Iterators are the glue that ties the standard-library algorithms to their data
- Iterators are the mechanism used to minimize an algorithm's dependence on the data structures on which it operates.

Alex Stepanov

The reason that STL containers and algorithms work so well together is that they know nothing of each other.

Iterator categories



Does our iterator work?

```
template <typename T>
class List<T>::Iterator {
    ...
};
```

Does our iterator work?

```
#include <iterator>
```

```
...
```

```
template <typename T>
```

```
class List<T>::Iterator : public
```

```
    std::iterator<std::forward_iterator_tag, T> {
```

```
    ...
```

```
};
```



```
template <typename Cat ,  
          typename T,  
          typename Dist = ptrdiff_t ,  
          typename Ptr = T* ,  
          typename Ref = T&>  
struct iterator{  
    using value_type = T;  
    using difference_type = Dist;  
    using pointer = Ptr;  
    using reference = Ref;  
    using iterator_category = Cat;  
};
```

- 1 The C++ standard library
- 2 Iterators
- 3 Containers
- 4 Algorithms
- 5 Function objects

Containers

Definition

A container holds a sequence of objects

Two categories

- Sequence containers: provide access to sequences of elements
- Associative containers: provide associative lookup based on a key

Associative containers

- Ordered
- Unordered

Sequence containers

Sequence Containers

vector<T,A>	A contiguously allocated sequence of T s; the default choice of container
list<T,A>	A doubly-linked list of T ; use when you need to insert and delete elements without moving existing elements
forward_list<T,A>	A singly-linked list of T ; ideal for empty and very short sequences
deque<T,A>	A double-ended queue of T ; a cross between a vector and a list; slower than one or the other for most uses

Ordered associative containers

Ordered Associative Containers (§iso.23.4.2)

C is the type of the comparison; **A** is the allocator type

<code>map<K,V,C,A></code>	An ordered map from K to V ; a sequence of (K , V) pairs
<code>multimap<K,V,C,A></code>	An ordered map from K to V ; duplicate keys allowed
<code>set<K,C,A></code>	An ordered set of K
<code>multiset<K,C,A></code>	An ordered set of K ; duplicate keys allowed

Unordered associative containers

Unordered Associative Containers (§iso.23.5.2)

H is the hash function type; **E** is the equality test; **A** is the allocator type

<code>unordered_map<K,V,H,E,A></code>	An unordered map from K to V
<code>unordered_multimap<K,V,H,E,A></code>	An unordered map from K to V ; duplicate keys allowed
<code>unordered_set<K,H,E,A></code>	An unordered set of K
<code>unordered_multiset<K,H,E,A></code>	An unordered set of K ; duplicate keys allowed

Array

array:

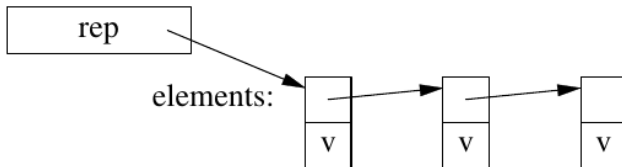
elements

Vector

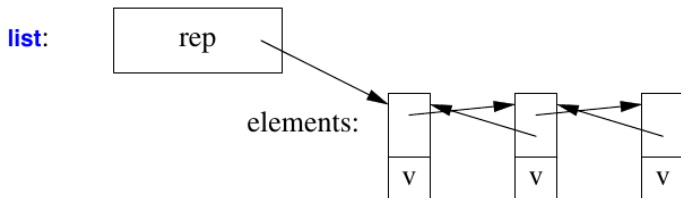


Forward list

forward_list:

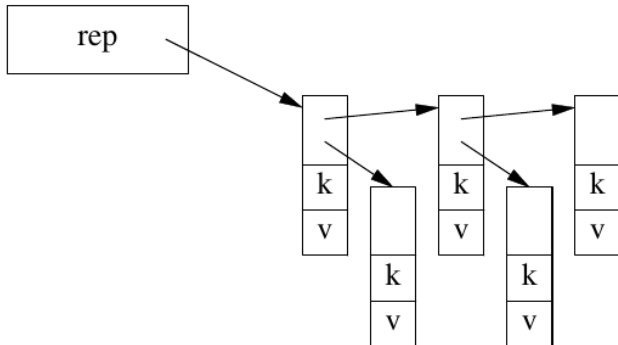


List

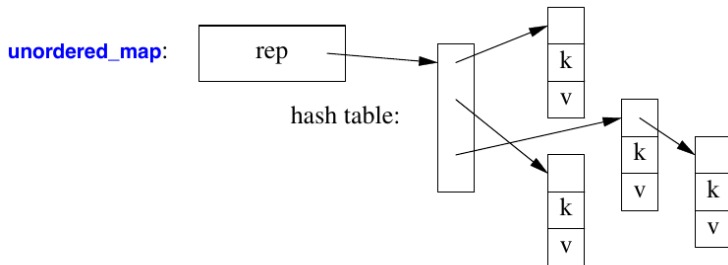


Map

map:



Unordered map



Operations and types

Container:

value_type, size_type, difference_type, pointer, const_pointer, reference, const_reference
 iterator, const_iterator, ?reverse_iterator, ?const_reverse_iterator, allocator_type
 begin(), end(), cbegin(), cend(), ?rbegin(), ?rend(), ?crbegin(), ?crend(), =, ==, !=
 swap(), ?size(), max_size(), empty(), clear(), get_allocator(), constructors, destructor
 ?<, ?<=, ?>, ?>=, ?insert(), ?emplace(), ?erase()

Sequence container:

assign(), front(), resize()
 ?back(), ?push_back()
 ?pop_back(), ?emplace_back()

Associative container:

key_type, mapped_type, ?[], ?at()
 lower_bound(), upper_bound(), equal_range()
 find(), count(), emplace_hint()

push_front(), pop_front()
 emplace_front()

[], at()
 shrink_to_fit()

Ordered container:

key_compare
 key_comp()
 value_comp()

Hashed container:

key_equal(), hasher
 hash_function()
 key_equal()
 bucket interface

List:

remove()
 remove_if(), unique()
 merge(), sort()
 reverse()

deque

data()
 capacity()
 reserve()

vector

splice()

insert_after(), erase_after()
 emplace_after(), splice_after()

list

forward_list

map

multimap

set

unordered_set

multiset

unordered_multimap

unordered_multiset

Operation complexity

Standard Container Operation Complexity					
	[] §31.2.2	List §31.3.7	Front §31.4.2	Back §31.3.6	Iterators §33.1.2
vector	const	$O(n)+$		const+	Ran
list		const	const	const	Bi
forward_list		const	const		For
deque	const	$O(n)$	const	const	Ran
stack				const	
queue			const	const	
priority_queue			$O(\log(n))$	$O(\log(n))$	
map	$O(\log(n))$	$O(\log(n))+$			Bi
multimap		$O(\log(n))+$			Bi
set		$O(\log(n))+$			Bi
multiset		$O(\log(n))+$			Bi
unordered_map	const+	const+			For
unordered_multimap		const+			For
unordered_set		const+			For
unordered_multiset		const+			For
string	const	$O(n)+$	$O(n)+$	const+	Ran
array	const				Ran
built-in array	const				Ran
valarray	const				Ran
bitset	const				

Prime numbers

```
#include <vector>

int main(){
    std::vector<int> primes;

    primes.emplace_back(2);

    for (int i=3; i<=max; ++i)
        if (is_prime(i))
            primes.emplace_back(i);

    for (const auto& x: primes)
        std::cout << x << std::endl;
}
```

Word count

```
#include <map>

int main(){
    std::map<std::string, int> words;

    for (std::string s; std::cin>>s;)
        ++words[s];

    for (const auto& x: words)
        std::cout << x.first << ": "
                    << x.second << std::endl;
}
```


Word count

```
#include <unordered_map>

int main(){
    std::unordered_map<std::string, int> words;

    for (std::string s; std::cin>>s;)
        ++words[s];

    for (const auto& x: words)
        std::cout << x.first << ": "
                    << x.second << std::endl;
}
```

- 1 The C++ standard library
- 2 Iterators
- 3 Containers
- 4 Algorithms**
- 5 Function objects

STL algorithms

- about 80 algorithms in `<algorithm>` and `<numeric>`
- operate on *sequences*
 - ▶ pair of iterators for inputs $[b : e)$
 - ▶ single iterator for output $[b2 : b2 + (e - b))$
- can take functions or function objects
- report failure by returning the end of the sequence

Examples

Sequences

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

int main(){
    std::vector<double> v1;
    ...
    std::vector<double> v2(v1.size());
    std::sort(v1.begin(), v1.end());
    std::copy(v1.begin(), v1.end(), v2.begin());
}
```

Examples

Sequences

```
#include <numeric>
#include <vector>

int main(){
    std::vector<double> v1;
    ...
    double sum{0};
    sum = std::accumulate(v1.begin(), v1.end(), sum);
}
```

Examples

User-defined functions

```
#include <numeric>
#include <vector>

double my_f(const double& a, const double& b){
    if(std::abs(b - 2.2) < 1e-12)
        return a;
    return a+b;
}

int main(){
    std::vector<double> v1;
    ...
    double sum{0};
    sum = std::accumulate(first, last, sum, my_f);
}
```

Examples

User-defined functions

```
#include <numeric>
#include <vector>

int main(){
    std::vector<double> v1;
    ...
    auto my_f = [](const double& a, const double &b)
        -> double {
        double res = 0;
        ((std::abs(b-2.2) < 1e-12) ? res = a : res= a+
            b);
        return res;
    };
    double sum{0};
    sum = std::accumulate(first, last, sum, my_f);
```

Examples

Failure check

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

int main(){
    std::vector<double> v1;
    ...
    auto it = std::find(v1.begin(), v1.end(), 2.2);

    if(it != v1.end())
        std::cout << "found " << *it << std::endl;
    else
        std::cout << "not found\n";
}
```


- 1 The C++ standard library
- 2 Iterators
- 3 Containers
- 4 Algorithms
- 5 **Function objects**

Function objects

- defined in `<functional>`
- comparison criteria
- predicates (functions returning `bool`)
- arithmetic operations

Predicates

Predicates (§iso.20.8.5, §iso.20.8.6, §iso.20.8.7)	
<code>p=equal_to<T>(x,y)</code>	<code>p(x,y)</code> means $x==y$ when x and y are of type T
<code>p=not_equal_to<T>(x,y)</code>	<code>p(x,y)</code> means $x!=y$ when x and y are of type T
<code>p=greater<T>(x,y)</code>	<code>p(x,y)</code> means $x>y$ when x and y are of type T
<code>p=less<T>(x,y)</code>	<code>p(x,y)</code> means $x<y$ when x and y are of type T
<code>p=greater_equal<T>(x,y)</code>	<code>p(x,y)</code> means $x>=y$ when x and y are of type T
<code>p=less_equal<T>(x,y)</code>	<code>p(x,y)</code> means $x<=y$ when x and y are of type T
<code>p=logical_and<T>(x,y)</code>	<code>p(x,y)</code> means $x\&\&y$ when x and y are of type T
<code>p=logical_or<T>(x,y)</code>	<code>p(x,y)</code> means $x y$ when x and y are of type T
<code>p=logical_not<T>(x)</code>	<code>p(x)</code> means $!x$ when x is of type T
<code>p=bit_and<T>(x,y)</code>	<code>p(x,y)</code> means $x\&y$ when x and y are of type T
<code>p=bit_or<T>(x,y)</code>	<code>p(x,y)</code> means $x y$ when x and y are of type T
<code>p=bit_xor<T>(x,y)</code>	<code>p(x,y)</code> means $x\hat{y}$ when x and y are of type T

Arithmetic operations

Arithmetic Operations (§iso.20.8.4)

f=plus<T>(x,y)	f(x,y) means x+y when x and y are of type T
f=minus<T>(x,y)	f(x,y) means x-y when x and y are of type T
f=multiplies<T>(x,y)	f(x,y) means x*y when x and y are of type T
f=divides<T>(x,y)	f(x,y) means x/y when x and y are of type T
f=modulus<T>(x,y)	f(x,y) means x%y when x and y are of type T
f=negate<T>(x)	f(x) means -x when x is of type T

Decreasing sort

```
#include <algorithm>
#include <vector>
#include <functional>

int main(){
    std::vector<double> v1;
    ...
    std::sort(v1.begin(), v1.end(),
              std::greater<double>{});
}
```

My comparison

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

template <typename num>
struct my_comparison{
    bool operator()(const num& a, const num& b) {
        return a > b;}
};

int main(){
    std::vector<double> v1;
    ...
    std::sort(v1.begin(), v1.end(),
              my_comparison<double>{});
}
```



C makes it easy to shoot yourself in
the foot; C++ makes it harder, but
when you do, it blows away your
whole leg.

— Bjarne Stroustrup —

AZ QUOTES