# A quick overview of the Standard Template Library
## Advanced Programming and Algorithmic Design

Alberto Sartori

November 27, 2018

# Outline

# What is the standard library?

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

# The C++ Programming Language

**Part IV: The Standard Library**               **857**

# The header files

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

# The header files

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

# The header files

| Algorithms | | |
|---|---|---|
| **<algorithm>** | General algorithms | §32.2 |
| **<cstdlib>** | **bsearch()**, **qsort()** | §43.7 |

# The header files

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

# The header files

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

# The header files

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

# The header files

| Localization | | |
|---|---|---|
| **\<locale\>** | Represent cultural differences | Chapter 39 |
| **\<clocale\>** | Represent cultural differences C-style | |
| **\<codecvt\>** | Code conversion facets | §39.4.6 |

# The header files

| Language Support | | |
|---|---|---|
| **&lt;limits&gt;** | Numeric limits | §40.2 |
| **&lt;climits&gt;** | C-style numeric scalar-limit macros | §40.2 |
| **&lt;cfloat&gt;** | C-style numeric floating-point limit macros | §40.2 |
| **&lt;cstdint&gt;** | Standard integer type names | §43.7 |
| **&lt;new&gt;** | Dynamic memory management | §11.2.3 |
| **&lt;typeinfo&gt;** | Run-time type identification support | §22.5 |
| **&lt;exception&gt;** | Exception-handling support | §30.4.1.1 |
| **&lt;initializer_list&gt;** | **initializer_list** | §30.3.1 |
| **&lt;cstddef&gt;** | C library language support | §10.3.1 |
| **&lt;cstdarg&gt;** | Variable-length function argument lists | §12.2.4 |
| **&lt;csetjmp&gt;** | C-style stack unwinding | |
| **&lt;cstdlib&gt;** | Program termination | §15.4.3 |
| **&lt;ctime&gt;** | System clock | §43.6 |
| **&lt;csignal&gt;** | C-style signal handling | |

# The header files

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

# The header files

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

# The header files

| **C Compatibility** | | |
|---|---|---|
| **&lt;cinttypes&gt;** | Aliases for common integer types | §43.7 |
| **&lt;cstdbool&gt;** | C **bool** | |
| **&lt;ccomplex&gt;** | **&lt;complex&gt;** | |
| **&lt;cfenv&gt;** | Floating-point environment | |
| **&lt;cstdalign&gt;** | C alignment | |
| **&lt;ctgmath&gt;** | C "type generic math": **&lt;complex&gt;** and **&lt;cmath&gt;** | |

# The header files

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

# We will focus on the STL

# 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 alogorithms 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
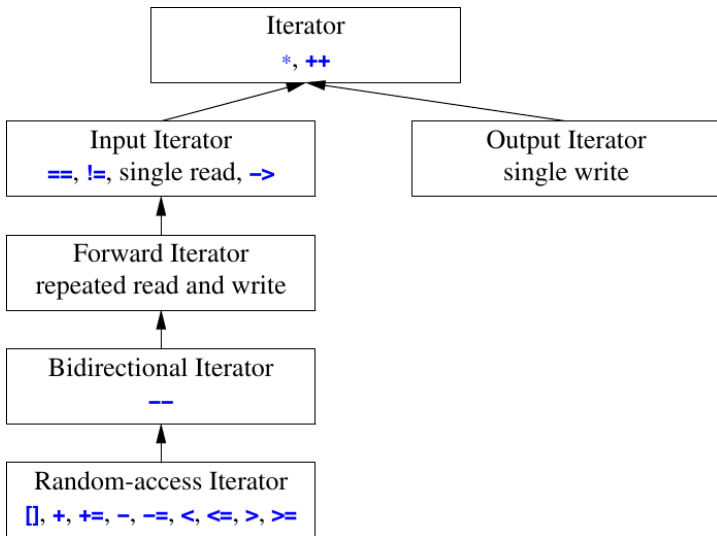
# Does our iterator work?

```cpp
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> {
  ...
};
```

```cpp
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;
};
```

# 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

| | |
|---|---|
| **map<K,V,C,A>** | An ordered map from **K** to **V**; a sequence of (**K**,**V**) pairs |
| **multimap<K,V,C,A>** | An ordered map from **K** to **V**; duplicate keys allowed |
| **set<K,C,A>** | An ordered set of **K** |
| **multiset<K,C,A>** | 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 | |
| unordered_map<K,V,H,E,A> | An unordered map from **K** to **V** |
| unordered_multimap<K,V,H,E,A> | An unordered map from **K** to **V**; duplicate keys allowed |
| unordered_set<K,H,E,A> | An unordered set of **K** |
| unordered_multiset<K,H,E,A> | An unordered set of **K**; duplicate keys allowed |

# Array

array:    | elements |

# Vector



**vector:**    rep

elements | free space

# Forward list

**forward_list**:

rep

elements:
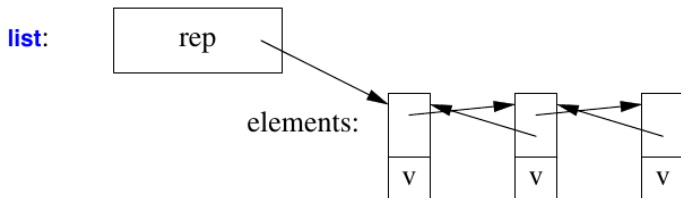
V          V          V

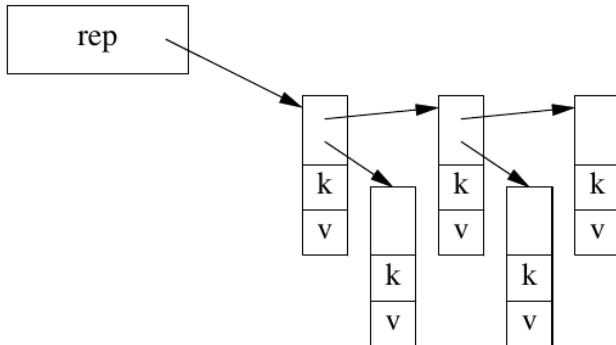# List



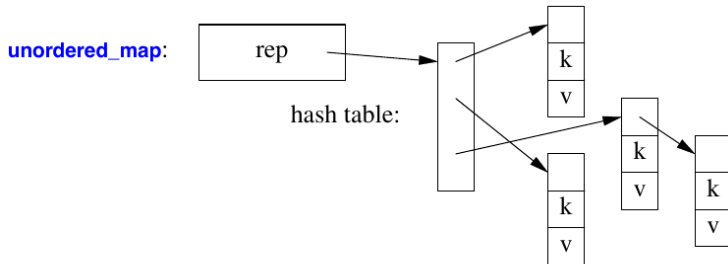**list:**     rep

elements:

V     V     V
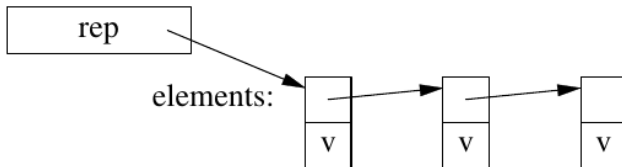
# Map

# 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()**

Assocative 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()**

**map**

**set**

**unordered_map**

**multimap**

**unordered_set**

**vector**

**multiset**

**unordered_multimap**

**splice()**

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

**list**

**forward_list**

**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 | 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 | | | | Ran |

# Prime numbers

```cpp
#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

```cpp
#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

```cpp
#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;
}
```

# 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 of function objects

- report failure by returning the end of the sequence

# Examples

Sequences

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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";
}
```

# 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) | |
|---|---|
| **p=equal_to<T>(x,y)** | **p(x,y)** means **x==y** when **x** and **y** are of type **T** |
| **p=not_equal_to<T>(x,y)** | **p(x,y)** means **x!=y** when **x** and **y** are of type **T** |
| **p=greater<T>(x,y)** | **p(x,y)** means **x>y** when **x** and **y** are of type **T** |
| **p=less<T>(x,y)** | **p(x,y)** means **x<y** when **x** and **y** are of type **T** |
| **p=greater_equal<T>(x,y)** | **p(x,y)** means **x>=y** when **x** and **y** are of type **T** |
| **p=less_equal<T>(x,y)** | **p(x,y)** means **x<=y** when **x** and **y** are of type **T** |
| **p=logical_and<T>(x,y)** | **p(x,y)** means **x&&y** when **x** and **y** are of type **T** |
| **p=logical_or<T>(x,y)** | **p(x,y)** means **x||y** when **x** and **y** are of type **T** |
| **p=logical_not<T>(x)** | **p(x)** means **!x** when **x** is of type **T** |
| **p=bit_and<T>(x,y)** | **p(x,y)** means **x&y** when **x** and **y** are of type **T** |
| **p=bit_or<T>(x,y)** | **p(x,y)** means **x|y** when **x** and **y** are of type **T** |
| **p=bit_xor<T>(x,y)** | **p(x,y)** means **x^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

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

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

# My comparison

```cpp
#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>{});
}
```