

# A quick overview of the C++ Standard (Template) Library

Advanced Programming

Alberto Sartori

December 03, 2019

# Outline

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

# 1 The C++ standard library

## 2 Containers

## 3 Iterators

## 4 Algorithms

## 5 Function objects

# What is the standard library?

The standard library is the set of components specified by the ISO C++ standard ( $\sim 1600$  dense pages for C++17) and shipped with identical behavior (modulo performance) by every C++ implementation.

<https://github.com/cplusplus/draft>

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

# The header files

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

# The header files

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



# The header files

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

# The header files

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

# The header files

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

# The header files

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

# The header files

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

# The header files

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

# The header files

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

# The header files

C Compatibility		
<code>&lt;inttypes&gt;</code>	Aliases for common integer types	§43.7
<code>&lt;cstdbool&gt;</code>	C <code>bool</code>	
<code>&lt;ccomplex&gt;</code>	<code>&lt;complex&gt;</code>	
<code>&lt;cfenv&gt;</code>	Floating-point environment	
<code>&lt;cstdalign&gt;</code>	C alignment	
<code>&lt;ctgmath&gt;</code>	C “type generic math”: <code>&lt;complex&gt;</code> and <code>&lt;cmath&gt;</code>	



# The header files

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

# We will focus on the STL 😊



# We will not see the concurrency library ☹

```
int main(){  
    // f and g are independent  
    f();  
    g();  
}
```

# We will not see the concurrency library ☹

```
#include <thread>
```

```
int main(){  
    // f and g are independent  
    std::thread t{ f };  
    g();  
    t.join();  
}
```

# We will not see the concurrency library ☹

```
#include <future>

int main(){
    // f and g are independent
    auto from_f = std::async( f );
    auto from_g = g();
    ...
    complicated( from_g, from_f.get() );
}
```

# We will not see the concurrency library 😞

Link against `pthread`

```
$ c++ test.cpp -pthread
```

```
$ c++ test.cpp -c  
$ c++ test.o -pthread
```

- 1 The C++ standard library
- 2 Containers
- 3 Iterators
- 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

<b>vector&lt;T,A&gt;</b>	A contiguously allocated sequence of <b>T</b> s; the default choice of container
<b>list&lt;T,A&gt;</b>	A doubly-linked list of <b>T</b> ; use when you need to insert and delete elements without moving existing elements
<b>forward_list&lt;T,A&gt;</b>	A singly-linked list of <b>T</b> ; ideal for empty and very short sequences
<b>deque&lt;T,A&gt;</b>	A double-ended queue of <b>T</b> ; 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&lt;K,V,C,A&gt;</code>	An ordered map from <b>K</b> to <b>V</b> ; a sequence of ( <b>K</b> , <b>V</b> ) pairs
<code>multimap&lt;K,V,C,A&gt;</code>	An ordered map from <b>K</b> to <b>V</b> ; duplicate keys allowed
<code>set&lt;K,C,A&gt;</code>	An ordered set of <b>K</b>
<code>multiset&lt;K,C,A&gt;</code>	An ordered set of <b>K</b> ; 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&lt;K,V,H,E,A&gt;</code>	An unordered map from <b>K</b> to <b>V</b>
<code>unordered_multimap&lt;K,V,H,E,A&gt;</code>	An unordered map from <b>K</b> to <b>V</b> ; duplicate keys allowed
<code>unordered_set&lt;K,H,E,A&gt;</code>	An unordered set of <b>K</b>
<code>unordered_multiset&lt;K,H,E,A&gt;</code>	An unordered set of <b>K</b> ; duplicate keys allowed

# Array

**array:**

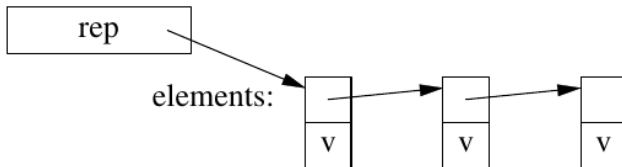
elements

# Vector

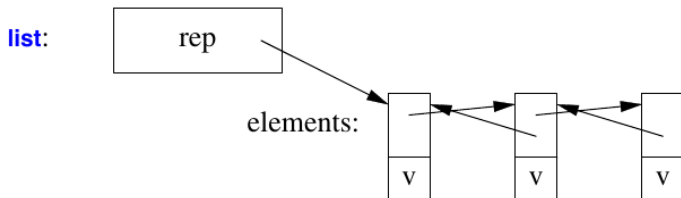


# Forward list

**forward\_list:**

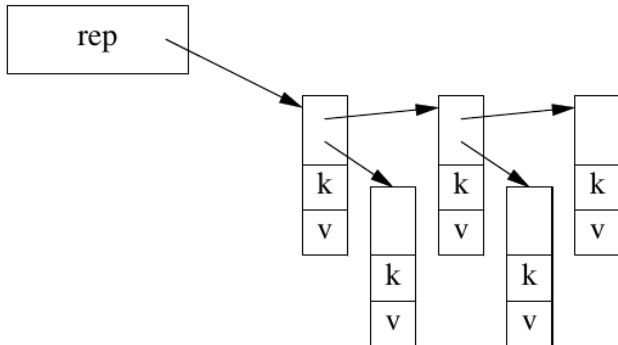


# List



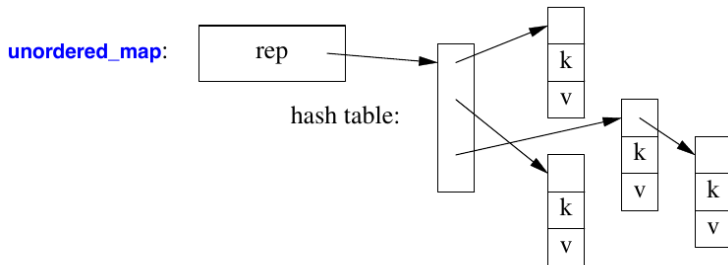
# Map

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\_map

multiset

unordered\_multiset

unordered\_set

unordered\_multimap

# 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
<b>vector</b>	const	O(n)+		const+	Ran
<b>list</b>		const	const	const	Bi
<b>forward_list</b>		const	const		For
<b>deque</b>	const	O(n)	const	const	Ran
<b>stack</b>				const	
<b>queue</b>			const	const	
<b>priority_queue</b>			O(log(n))	O(log(n))	
<b>map</b>	O(log(n))	O(log(n))+			Bi
<b>multimap</b>		O(log(n))+			Bi
<b>set</b>		O(log(n))+			Bi
<b>multiset</b>		O(log(n))+			Bi
<b>unordered_map</b>	const+	const+			For
<b>unordered_multimap</b>		const+			For
<b>unordered_set</b>		const+			For
<b>unordered_multiset</b>		const+			For
<b>string</b>	const	O(n)+	O(n)+	const+	Ran
<b>array</b>	const				Ran
<b>built-in array</b>	const				Ran
<b>valarray</b>	const				Ran
<b>bitset</b>	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 Containers
- 3 Iterators**
- 4 Algorithms
- 5 Function objects

# What is an Iterator?

## Design pattern [GoF]

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

## Stepanov

Iterator is a coordinate.

## 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

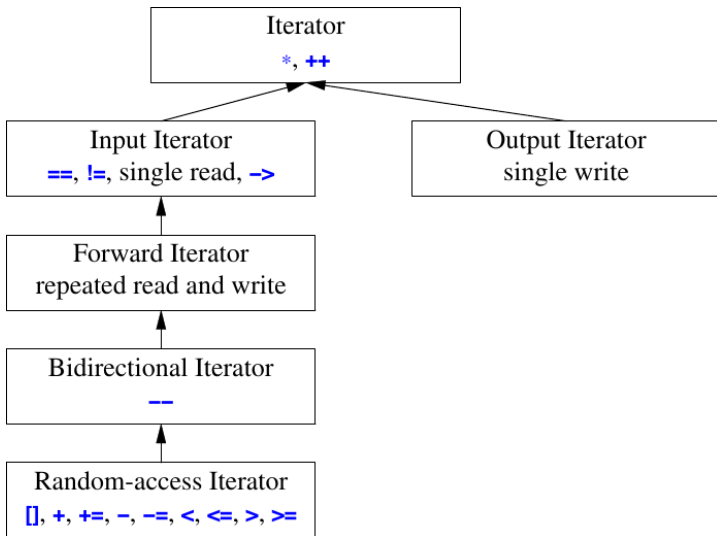
- Iterators are the glue that ties the standard-library algorithms to their data
- Iterators are the mechanism used to minimize an algorithm's dependence on the data structures on which it operates.

## Alex Stepanov

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

```
while( first != last )
```

# Iterator categories



# How to implement our own iterator?

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

# How to implement our own iterator?

```
#include <iterator>

...

template <typename T>
class List<T>::Iterator{
    typename List<T>::node* current;
public:
    using value_type = T;
    using difference_type = std::ptrdiff_t;
    using iterator_category =
        std::forward_iterator_tag;
    using reference = value_type&;
    using pointer = value_type*;
    ...
}
```

## How to implement our own iterator?

```
...  
reference operator*() {  
    return current->value; }  
pointer operator->() { return &**this; }  
Iterator& operator++() {  
    current = current->next;  
    return *this;  
}  
  
friend  
bool operator==(const Iterator&, const  
    Iterator&);  
  
friend  
bool operator!=(const Iterator&, const  
    Iterator&);  
  
};
```

- 1 The C++ standard library
- 2 Containers
- 3 Iterators
- 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 (e.g. not found) 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

## Lambda functions

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

# Examples

Generic lambdas (since C++14)

```
#include <numeric>
#include <vector>
int main(){
    std::vector<double> v1;
    ...
    auto my_f = [](const auto& a, const auto& b) {
        return ( (std::abs(b-2.2) < 1e-12) ? a : a+b);
    };
    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 Containers
- 3 Iterators
- 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&lt;T&gt;(x,y)</code>	<code>p(x,y)</code> means $x==y$ when $x$ and $y$ are of type $T$
<code>p=not_equal_to&lt;T&gt;(x,y)</code>	<code>p(x,y)</code> means $x!=y$ when $x$ and $y$ are of type $T$
<code>p=greater&lt;T&gt;(x,y)</code>	<code>p(x,y)</code> means $x>y$ when $x$ and $y$ are of type $T$
<code>p=less&lt;T&gt;(x,y)</code>	<code>p(x,y)</code> means $x<y$ when $x$ and $y$ are of type $T$
<code>p=greater_equal&lt;T&gt;(x,y)</code>	<code>p(x,y)</code> means $x>=y$ when $x$ and $y$ are of type $T$
<code>p=less_equal&lt;T&gt;(x,y)</code>	<code>p(x,y)</code> means $x<=y$ when $x$ and $y$ are of type $T$
<code>p=logical_and&lt;T&gt;(x,y)</code>	<code>p(x,y)</code> means $x\&\&y$ when $x$ and $y$ are of type $T$
<code>p=logical_or&lt;T&gt;(x,y)</code>	<code>p(x,y)</code> means $x  y$ when $x$ and $y$ are of type $T$
<code>p=logical_not&lt;T&gt;(x)</code>	<code>p(x)</code> means $!x$ when $x$ is of type $T$
<code>p=bit_and&lt;T&gt;(x,y)</code>	<code>p(x,y)</code> means $x\&y$ when $x$ and $y$ are of type $T$
<code>p=bit_or&lt;T&gt;(x,y)</code>	<code>p(x,y)</code> means $x y$ when $x$ and $y$ are of type $T$
<code>p=bit_xor&lt;T&gt;(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)

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

# 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>{});
}
```

# Lambda

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

int main(){
    std::vector<double> v1;
    ...
    std::sort(v1.begin(), v1.end(),
              [](const auto& a, const auto& b)
                { return a>b; } );
}
```

