Introduction to C++ STL

(Standard Template Library)

Giulio Ermanno Pibiri
giulio.pibiri@di.unipi.it

Rossano Venturini
rossano.venturini@unipi.it

22/09/2017

What is the C++ STL?

Alexander Stepanov

The Standard Template Library (STL) is a C++ framework consisting in **template-based** classes and algorithms that implement mostly used data structures and common tasks.

What is the C++ STL?

Alexander Stepanov

The Standard Template Library (STL) is a C++ framework consisting in **template-based** classes and algorithms that implement mostly used data structures and common tasks.

Roughly speaking:

16 containers (data structures)

~90 algorithms

+ utilities

Why we care about STL

- Because we will use C++ in this course.
- You do not have to re-invent the wheel.
- Learn more about C++.

Why we care about STL

- Because we will use C++ in this course.
- You do not have to re-invent the wheel.
- Learn more about C++.

My advice: **be skeptic**.
Measure first, then conclude.

Why we care about STL

- Because we will use C++ in this course.
- You do not have to re-invent the wheel.
- Learn more about C++.

Robert Sedgewick

My advice: **be skeptic**.
Measure first, then conclude.

"Use a data structure (or an algorithm) once you know its performance".

Why we care about STL

- Because we will use C++ in this course.
- You do not have to re-invent the wheel.
- Learn more about C++.

Robert Sedgewick

My advice: **be skeptic**.
Measure first, then conclude.

"Use a data structure (or an algorithm) once you know its performance".

You **must** consult sources like:
http://www.cplusplus.com/
http://en.cppreference.com/

STL goal

Generic programming: code once, re-use many times.

Increase correctness.

Wider range of uses.

STL goal

Generic programming: code once, re-use many times.

Increase correctness.
Wider range of uses.

C++ templates

STL goal

Generic programming: code once, re-use many times.

Increase correctness.
Wider range of uses.

function templates ← **C++ templates**

```
template<typename T>
T max(T x, T y) {
    return x > y ? x : y;
}
```

```
auto x = max<int>(3, 12);
auto y = max<float>(3.4, 0.03);
```

```
template<typename T1,
         typename T2>
bool are_equal(T1 x, T2 y) {
    return x == y;
}
```

```
if (are_equal<int, double>(5, 5.0)) {
    std::cout << "equal" << std::endl;
}
```

STL goal

Generic programming: code once, re-use many times.

Increase correctness.
Wider range of uses.

function templates ← **C++ templates** → **class** templates

```
template<typename T>
T max(T x, T y) {
    return x > y ? x : y;
}
```

```
auto x = max<int>(3, 12);
auto y = max<float>(3.4, 0.03);
```

```
template<typename T1,
         typename T2>
bool are_equal(T1 x, T2 y) {
    return x == y;
}
```

```
if (are_equal<int, double>(5, 5.0)) {
    std::cout << "equal" << std::endl;
}
```

```
template<typename T>
struct my_container {

    my_container(T val)
        : m_val(val)
    {}

    void increment() {
        ++m_val;
    }

    T get() {
        return m_val;
    }

private:
    T m_val;
};
```

STL goal

Generic programming: code once, re-use many times.

Increase correctness.
Wider range of uses.

function templates ← **C++ templates** → **class** templates

```
template<typename T>
T max(T x, T y) {
    return x > y ? x
}
```

```
auto x = max<int>(3,
auto y = max<float>(
```

```
template<typename T1
         typename T2
bool are_equal(T1 x,
    return x == y;
}
```

```
if (are_equal<int, d
    std::cout << "equal" << std::endl;
}
```

```
template<typename T>
t my_container {

y_container(T val)
    : m_val(val)
}

oid increment() {
    ++m_val;


 get() {
    return m_val;


te:
    T m_val;
};
```

Trade-off between reusability and performance.

Experiment by yourself.
Try to understand **why**.

STL goal

Generic programming: code once, re-use many times.

Increase correctness.
Wider range of uses.

function templates ← **C++ templates** → **class** templates

```
template<typename T>
T max(T x, T y) {
    return x > y ? x
}
```

```
auto x = max<int>(3,
auto y = max<float>(
```

```
template<typename T1
         typename T2
bool are_equal(T1 x,
    return x == y;
}
```

```
if (are_equal<int, d
    std::cout << "equal" << std::endl;
}
```

```
template<typename T>
t my_container {

y_container(T val)
    : m_val(val)
}

oid increment() {
    ++m_val;


 get() {
    return m_val;


te:
    T m_val;
};
```

Trade-off between reusability and performance.

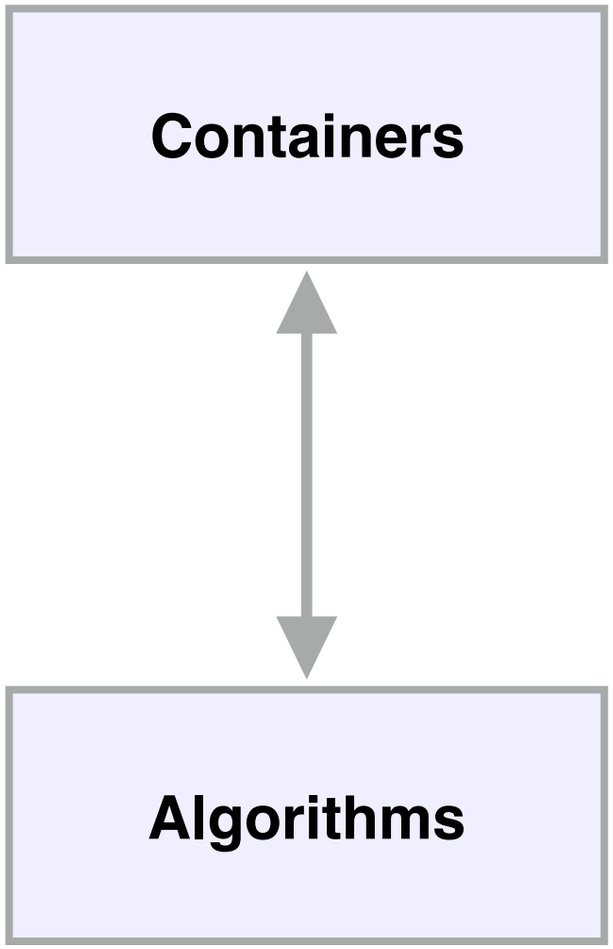
Experiment by yourself.
Try to understand **why**.

Indeed one of the goals of ours for this course!

STL basic model

Separation of concerns

16



Store data;
do not know
about algorithms.

~90

Manipulate data;
do not know
about containers.

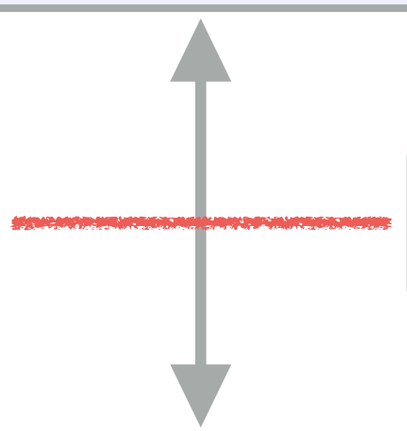
STL basic model

Separation of concerns

16

Containers

Store data;
do not know
about algorithms.



Iterators

~90

Algorithms

Manipulate data;
do not know
about containers.

An example

[source: http://www.cplusplus.com]

function template

std::find <algorithm>

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

Find value in range

Returns an iterator to the first element in the range `[first,last)` that compares equal to *val*. If no such element is found, the function returns *last*.

The function uses `operator==` to compare the individual elements to *val*.

An example

[source: http://www.cplusplus.com]

function template

std::find <algorithm>

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

Find value in range

Returns an iterator to the first element in the range `[first,last)` that compares equal to *val*. If no such element is found, the function returns *last*.

The function uses `operator==` to compare the individual elements to *val*.

```
// find example
#include <iostream>     // std::cout
#include <algorithm>    // std::find
#include <vector>       // std::vector

int main () {
  // using std::find with array and pointer:
  int myints[] = { 10, 20, 30, 40 };
  int * p;

  p = std::find (myints, myints+4, 30);
  if (p != myints+4)
    std::cout << "Element found in myints: " << *p << '\n';
  else
    std::cout << "Element not found in myints\n";

  // using std::find with vector and iterator:
  std::vector<int> myvector (myints,myints+4);
  std::vector<int>::iterator it;

  it = find (myvector.begin(), myvector.end(), 30);
  if (it != myvector.end())
    std::cout << "Element found in myvector: " << *it << '\n';
  else
    std::cout << "Element not found in myvector\n";

  return 0;
}
```

Containers

A container is a holder object that stores a **collection** of other objects (its elements). These are implemented as **class templates**.
The container manages the **storage space** for its elements and provides member functions to access them, either directly or through iterators.

Containers

A container is a holder object that stores a **collection** of other objects (its elements). These are implemented as **class templates**.
The container manages the **storage space** for its elements and provides member functions to access them, either directly or through iterators.

```
vector
deque
list
```

```
set
multiset
map
multimap
```

```
stack
queue
priority_queue
```

Containers

A container is a holder object that stores a **collection** of other objects (its elements). These are implemented as **class templates**.
The container manages the **storage space** for its elements and provides member functions to access them, either directly or through iterators.

sequential

```
vector
deque
list
```

```
stack
queue
priority_queue
```

```
set
multiset
map
multimap
```

Containers

A container is a holder object that stores a **collection** of other objects (its elements). These are implemented as **class templates**.
The container manages the **storage space** for its elements and provides member functions to access them, either directly or through iterators.

sequential

```
vector
deque
list
```

```
set
multiset
map
multimap
```

adaptors

```
stack
queue
priority_queue
```

Containers

A container is a holder object that stores a **collection** of other objects (its elements). These are implemented as **class templates**.
The container manages the **storage space** for its elements and provides member functions to access them, either directly or through iterators.

sequential

```
vector
deque
list
```

adaptors

```
stack
queue
priority_queue
```

associative

```
set
multiset
map
multimap
```

Containers

A container is a holder object that stores a **collection** of other objects (its elements). These are implemented as **class templates**.
The container manages the **storage space** for its elements and provides member functions to access them, either directly or through iterators.

+ 6 containers with C++11

sequential

```
vector
deque
list
array
forward_list
```

adaptors

```
stack
queue
priority_queue
```

associative

```
set
multiset
map
multimap
unordered_set
unordered_multiset
unordered_map
unordered_multimap
```

Sequential containers

random access: O(1)
other operations: O(N)

class template

std::array

```
template < class T, size_t N > class array;
```

Arrays are **fixed-size** sequence containers: they hold a specific number of elements ordered in a strict linear sequence. A wrap class around the an ordinary array declared with the language's bracket syntax ([ ]).

```cpp
#include <iostream>
#include <array>

int main(int argc, char** argv) {

    // size_t N = std::stoull(argv[1]); --> compile-time error:
    //                                  N must be known in advance
    const uint32_t N = 100;
    std::array<uint32_t, N> a;

    for (uint32_t i = 0; i < N; ++i) {
        a[i] = i;
    }

    a[4] = 13;
    a[0] = 23;
    a.front() = 1;
    a.back() = 1000;
    // a[N + 1] = 9; --> runtime error: access out of bounds

    std::cout << "array size is: " << a.size() << "\n";
    for (auto it = a.begin(); it != a.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

class template

std::vector

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

Just like arrays, vectors use contiguous storage locations for their elements but unlike arrays, **their size can change dynamically**, with their storage being handled automatically by the container.

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++i) {
        v.push_back(i);
    }

    v[4] = 13;
    v[0] = 23;
    v.front() = 1;
    v.back() = 1000;
    // v[N + 1] = 9; --> runtime error: access out of bounds

    std::cout << "vector size is: " << v.size() << "\n";
    for (auto item: v) {
        std::cout << item << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Sequential containers

class template

std::deque

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

Deque is a **d**ouble-**e**nded **que**ue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on **both ends** (either its front or its back). Behaviour similar to that of vectors.

random access: O(1)
other operations: O(N)

class template

std::forward_list

```
template < class T, class Alloc = allocator<T> > class forward_list;
```

Forward lists are implemented as **singly-linked lists**. Singly 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 next element in the sequence.

random access: O(N)
insert/delete: O(1)

class template

std::list

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

List containers are implemented as **doubly-linked lists**.

Sequential containers

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++i) {
        v.push_back(i);
    }

    uint64_t sum = 0;
    for (auto item: v) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

Sequential containers

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++i) {
        v.push_back(i);
    }

    uint64_t sum = 0;
    for (auto item: v) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
→  Desktop g++ -std=c++11 sum_vector.cpp -o sum_vector
→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  2.39s user 0.19s system 98% cpu 2.628 total
→  Desktop
```

Sequential containers

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++i) {
        v.push_back(i);
    }

    uint64_t sum = 0;
    for (auto item: v) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[→  Desktop g++ -std=c++11 sum_vector.cpp -o sum_vector
[→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  2.39s user 0.19s system 98% cpu 2.628 total
→  Desktop
```

```cpp
#include <iostream>
#include <forward_list>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::forward_list<uint32_t> l;

    for (uint32_t i = 0; i < N; ++i) {
        l.push_front(i);
    }

    uint64_t sum = 0;
    for (auto item: l) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

Sequential containers

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++i) {
        v.push_back(i);
    }

    uint64_t sum = 0;
    for (auto item: v) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[➜  Desktop g++ -std=c++11 sum_vector.cpp -o sum_vector
[➜  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  2.39s user 0.19s system 98% cpu 2.628 total
➜  Desktop
```

```cpp
#include <iostream>
#include <forward_list>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::forward_list<uint32_t> l;

    for (uint32_t i = 0; i < N; ++i) {
        l.push_front(i);
    }

    uint64_t sum = 0;
    for (auto item: l) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[➜  Desktop g++ -std=c++11 sum_list.cpp -o sum_list
[➜  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  15.29s user 0.95s system 95% cpu 16.978 total
➜  Desktop
```

Sequential containers

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++i) {
        v.push_back(i);
    }

    uint64_t sum = 0;
    for (auto item: v) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[➜  Desktop g++ -std=c++11 sum_vector.cpp -o sum_vector
[➜  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  2.39s user 0.19s system 98% cpu 2.628 total
➜  Desktop
```

```cpp
#include <iostream>
#include <forward_list>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::forward_list<uint32_t> l;

    for (uint32_t i = 0; i < N; ++i) {
        l.push_front(i);
    }

    uint64_t sum = 0;
    for (auto item: l) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[➜  Desktop g++ -std=c++11 sum_list.cpp -o sum_list
[➜  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  15.29s user 0.95s system 95% cpu 16.978 total
➜  Desktop
```

X6

Sequential containers

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++i) {
        v.push_back(i);
    }

    uint64_t sum = 0;
    for (auto item: v) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[→  Desktop g++ -std=c++11 sum_vector.cpp -o sum_vector
[→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  2.39s user 0.19s system 98% cpu 2.628 total
→  Desktop
```

```cpp
#include <iostream>
#include <forward_list>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::forward_list<uint32_t> l;

    for (uint32_t i = 0; i < N; ++i) {
        l.push_front(i);
    }

    uint64_t sum = 0;
    for (auto item: l) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[→  Desktop g++ -std=c++11 sum_list.cpp -o sum_list
[→  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  15.29s user 0.95s system 95% cpu 16.978 total
→  Desktop
```

X6

```
[→  Desktop g++ -std=c++11 -O3 sum_list.cpp -o sum_list
[→  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  10.04s user 0.91s system 95% cpu 11.468 total
```

Sequential containers

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++i) {
        v.push_back(i);
    }

    uint64_t sum = 0;
    for (auto item: v) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[→  Desktop g++ -std=c++11 sum_vector.cpp -o sum_vector
[→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  2.39s user 0.19s system 98% cpu 2.628 total
→  Desktop
```

```
[→  Desktop g++ -std=c++11 -O3 sum_vector.cpp -o sum_vector
[→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  0.21s user 0.19s system 92% cpu 0.434 total
→  Desktop
```

```cpp
#include <iostream>
#include <forward_list>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::forward_list<uint32_t> l;

    for (uint32_t i = 0; i < N; ++i) {
        l.push_front(i);
    }

    uint64_t sum = 0;
    for (auto item: l) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[→  Desktop g++ -std=c++11 sum_list.cpp -o sum_list
[→  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  15.29s user 0.95s system 95% cpu 16.978 total
→  Desktop
```

X6

```
[→  Desktop g++ -std=c++11 -O3 sum_list.cpp -o sum_list
[→  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  10.04s user 0.91s system 95% cpu 11.468 total
```

Sequential containers

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++i) {
        v.push_back(i);
    }

    uint64_t sum = 0;
    for (auto item: v) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
→  Desktop g++ -std=c++11 sum_vector.cpp -o sum_vector
→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  2.39s user 0.19s system 98% cpu 2.628 total
→  Desktop
```

```
→  Desktop g++ -std=c++11 -O3 sum_vector.cpp -o sum_vector
→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  0.21s user 0.19s system 92% cpu 0.434 total
→  Desktop
```

```cpp
#include <iostream>
#include <forward_list>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::forward_list<uint32_t> l;

    for (uint32_t i = 0; i < N; ++i) {
        l.push_front(i);
    }

    uint64_t sum = 0;
    for (auto item: l) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
→  Desktop g++ -std=c++11 sum_list.cpp -o sum_list
→  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  15.29s user 0.95s system 95% cpu 16.978 total
→  Desktop
```

X6

```
→  Desktop g++ -std=c++11 -O3 sum_list.cpp -o sum_list
→  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  10.04s user 0.91s system 95% cpu 11.468 total
```

X47

Sequential containers

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++
        v.push_back(i);
    }

    uint64_t sum = 0;
    for (auto item: v) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```cpp
#include <iostream>
#include <forward_list>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::forward_list<uint32_t> l;

    for (uint32_t i = 0; i < N; ++i) {
        l.push_front(i);
    }

    uint64_t sum = 0;
    for (auto item: l) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

Do not use lists. Always prefer contiguous (cache-friendly) data structures, like `vectors`.

```
[→  Desktop g++ -std=c++11 sum_vector.cpp -o sum_vector
[→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  2.39s user 0.19s system 98% cpu 2.628 total
→  Desktop
```

```
[→  Desktop g++ -std=c++11 sum_list.cpp -o sum_list
[→  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  15.29s user 0.95s system 95% cpu 16.978 total
→  Desktop
```

X6

```
[→  Desktop g++ -std=c++11 -O3 sum_vector.cpp -o sum_vector
[→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  0.21s user 0.19s system 92% cpu 0.434 total
→  Desktop
```

```
[→  Desktop g++ -std=c++11 -O3 sum_list.cpp -o sum_list
[→  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  10.04s user 0.91s system 95% cpu 11.468 total
```

X47

Sequential containers

```cpp
#include <iostream>
#include <vector>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::vector<uint32_t> v;
    v.reserve(N);

    for (uint32_t i = 0; i < N; ++i) {
        v.push_back(i);
    }

    uint64_t sum = 0;
    for (auto item: v) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[→  Desktop g++ -std=c++11 sum_vector.cpp -o sum_vector
[→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  2.39s user 0.19s system 98% cpu 2.628 total
→  Desktop
```

```
[→  Desktop g++ -std=c++11 -O3 sum_vector.cpp -o sum_vector
[→  Desktop time ./sum_vector 50000000
sum: 1249999975000000
./sum_vector 50000000  0.21s user 0.19s system 92% cpu 0.434 total
→  Desktop
```

Do not use lists. Always prefer contiguous (cache-friendly) data structures, like `vectors`.

```cpp
#include <iostream>
#include <forward_list>

int main(int argc, char** argv) {

    size_t N = std::stoull(argv[1]);
    std::forward_list<uint32_t> l;

    for (uint32_t i = 0; i < N; ++i) {
        l.push_front(i);
    }

    uint64_t sum = 0;
    for (auto item: l) {
        sum += item;
    }

    std::cout << "sum: " << sum << std::endl;

    return 0;
}
```

```
[→  Desktop g++ -std=c++11 sum_list.cpp -o sum_list
[→  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  15.29s user 0.95s system 95% cpu 16.978 total
→  Desktop
```

X6

```
[→  Desktop g++ -std=c++11 -O3 sum_list.cpp -o sum_list
[→  Desktop time ./sum_list 50000000
sum: 1249999975000000
./sum_list 50000000  10.04s user 0.91s system 95% cpu 11.468 total
```

X47

Container adaptors

Containers adaptors are classes that use an encapsulated object of a specific container class as its **underlying container**, providing a specific set of member functions to access its elements.

class template

std::stack

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

class template

std::queue

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

class template

std::priority_queue

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

Container adaptors

Containers adaptors are classes that use an encapsulated object of a specific container class as its **underlying container**, providing a specific set of member functions to access its elements.

class template

std::stack

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

LIFO policy
push/pop: O(1)

class template

std::queue

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

class template

std::priority_queue

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

Container adaptors

Containers adaptors are classes that use an encapsulated object of a specific container class as its **underlying container**, providing a specific set of member functions to access its elements.

class template

std::stack

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

LIFO policy
push/pop: O(1)

class template

std::queue

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

FIFO policy
push/pop: O(1)

class template

std::priority_queue

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

Container adaptors

Containers adaptors are classes that use an encapsulated object of a specific container class as its **underlying container**, providing a specific set of member functions to access its elements.

class template

std::stack

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

LIFO policy
push/pop: O(1)

class template

std::queue

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

FIFO policy
push/pop: O(1)

class template

std::priority_queue

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

CUSTOM policy
push/pop: O(log N)

Container adaptors

Containers adaptors are classes that use an encapsulated object of a specific container class as its **underlying container**, providing a specific set of member functions to access its elements.

class template

std::stack

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

LIFO policy
push/pop: O(1)

class template

std::queue

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

FIFO policy
push/pop: O(1)

class template

std::priority_queue

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

CUSTOM policy
push/pop: O(log N)

`vector`, `deque` and `list` can be used here

Container adaptors

```cpp
#include <iostream>
#include <vector>
#include <stack>

int main() {

    // std::stack<int> st; --> uses a std::deque<int> internally
    std::stack<int, std::vector<int>> st;

    if (st.empty()) {
        std::cout << st.size() << std::endl;
    }

    for (int i = 0; i < 10; ++i) {
        st.push(i);
    }

    for (int i = 0; i < 10; ++i) {
        std::cout << st.top() << "\n";
        st.pop();
    }

    return 0;
}
```

Container adaptors

```cpp
#include <iostream>
#include <vector>
#include <stack>

int main() {

    // std::stack<int> st; --> uses a std::deque<int> internally
    std::stack<int, std::vector<int>> st;

    if (st.empty()) {
        std::cout << st.size() << std::endl;
    }

    for (int i = 0; i < 10; ++i) {
        st.push(i);
    }

    for (int i = 0; i < 10; ++i) {
        std::cout << st.top() << "\n";
        st.pop();
    }

    return 0;
}
```

```cpp
#include <iostream>
#include <list>
#include <queue>

int main() {

    std::queue<int, std::list<int>> q;

    for (int i = 0; i < 10; ++i) {
        q.push(i);
    }

    for (int i = 0; i < 10; ++i) {
        std::cout << q.front() << "\n";
        q.pop();
    }

    return 0;
}
```

Container adaptors

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <functional> // for std::greater

template<typename T>
struct even_comparator {
    bool operator()(T const& x, T const& y) {
        if (x % 2 == 0) return true;
        if (y % 2 == 0) return false;
        return false;
    }
};

template<typename PriorityQueue>
void print(PriorityQueue& pq, int N) {
    for (int i = 0; i < N; ++i) {
        std::cout << pq.top() << "\n";
        pq.pop();
    }
}
```

```cpp
int main() {

    int vec[] = {0, 23, 1, 4, 12, 5, 8, 11};
    int N = sizeof(vec) / sizeof(int);
    std::cout << "N: " << N << std::endl;

    std::cout << "=======\n"; {
        std::priority_queue<int> pq(std::begin(vec),
                                    std::end(vec));
        print<std::priority_queue<int>>(pq, N);
    }

    std::cout << "=======\n"; {
        typedef std::priority_queue<int,
                                    std::vector<int>,
                                    std::greater<int>
                                   > custom_pq1;
        custom_pq1 pq(std::begin(vec),
                      std::end(vec));
        print<custom_pq1>(pq, N);
    }

    std::cout << "=======\n"; {
        typedef std::priority_queue<int,
                                    std::vector<int>,
                                    even_comparator<int>
                                   > custom_pq2;
        custom_pq2 pq(std::begin(vec),
                      std::end(vec));
        print<custom_pq2>(pq, N);
    }

    std::cout << std::flush;
    return 0;
}
```

Associative containers

class template

std::set

```
template < class T,                          // set::key_type/value_type
           class Compare = less<T>,          // set::key_compare/value_compare
           class Alloc = allocator<T>        // set::allocator_type
           > class set;
```

class template

std::map

```
template < class Key,                                    // map::key_type
           class T,                                      // map::mapped_type
           class Compare = less<Key>,                    // map::key_compare
           class Alloc = allocator<pair<const Key,T> >   // map::allocator_type
           > class map;
```

class template

std::unordered_set

```
template < class Key,                          // unordered_set::key_type/value_type
           class Hash = hash<Key>,             // unordered_set::hasher
           class Pred = equal_to<Key>,         // unordered_set::key_equal
           class Alloc = allocator<Key>        // unordered_set::allocator_type
           > class unordered_set;
```

class template

std::unordered_map

```
template < class Key,                                    // unordered_map::key_type
           class T,                                      // unordered_map::mapped_type
           class Hash = hash<Key>,                       // unordered_map::hasher
           class Pred = equal_to<Key>,                   // unordered_map::key_equal
           class Alloc = allocator< pair<const Key,T> >  // unordered_map::allocator_type
           > class unordered_map;
```

Associative containers

class template

std::set

```
template < class T,                          // set::key_type/value_type
           class Compare = less<T>,          // set::key_compare/value_compare
           class Alloc = allocator<T>        // set::allocator_type
           > class set;
```

class template

std::map

```
template < class Key,                                    // map::key_type
           class T,                                      // map::mapped_type
           class Compare = less<Key>,                    // map::key_compare
           class Alloc = allocator<pair<const Key,T> >   // map::allocator_type
           > class map;
```

class template

std::unordered_set

```
template < class Key,                          // unordered_set::key_type/value_type
           class Hash = hash<Key>,             // unordered_set::hasher
           class Pred = equal_to<Key>,         // unordered_set::key_equal
           class Alloc = allocator<Key>        // unordered_set::allocator_type
           > class unordered_set;
```

class template

std::unordered_map

```
template < class Key,                                  // unordered_map::key_type
           class T,                                    // unordered_map::mapped_type
           class Hash = hash<Key>,                     // unordered_map::hasher
           class Pred = equal_to<Key>,                 // unordered_map::key_equal
           class Alloc = allocator< pair<const Key,T> >  // unordered_map::allocator_type
           > class unordered_map;
```

based on (balanced)
binary search trees

insert/delete: O(log N)
range queries: O(|range|)

Associative containers

class template

std::set

```
template < class T,                          // set::key_type/value_type
           class Compare = less<T>,          // set::key_compare/value_compare
           class Alloc = allocator<T>        // set::allocator_type
           > class set;
```

class template

std::map

```
template < class Key,                                    // map::key_type
           class T,                                      // map::mapped_type
           class Compare = less<Key>,                    // map::key_compare
           class Alloc = allocator<pair<const Key,T> >   // map::allocator_type
           > class map;
```

based on (balanced)
binary search trees

insert/delete: O(log N)
range queries: O(|range|)

class template

std::unordered_set

```
template < class Key,                          // unordered_set::key_type/value_type
           class Hash = hash<Key>,             // unordered_set::hasher
           class Pred = equal_to<Key>,         // unordered_set::key_equal
           class Alloc = allocator<Key>        // unordered_set::allocator_type
           > class unordered_set;
```

class template

std::unordered_map

```
template < class Key,                                  // unordered_map::key_type
           class T,                                    // unordered_map::mapped_type
           class Hash = hash<Key>,                     // unordered_map::hasher
           class Pred = equal_to<Key>,                 // unordered_map::key_equal
           class Alloc = allocator< pair<const Key,T> >  // unordered_map::allocator_type
           > class unordered_map;
```

based on **hashing**

insert/delete: O(1) exp.
range queries: ——

Associative containers

```cpp
#include <iostream>
#include <chrono>
#include <set>

#define MILLION 1000000

int main(int argc, char** argv) {

    if (argc < 2) {
        return 1;
    }

    size_t N = std::stoull(argv[1]);
    std::set<uint64_t> s;
    for (uint64_t i = 0; i < N; ++i) {
        s.insert(i);
    }

    typedef std::chrono::high_resolution_clock clock;

    auto start = clock::now();
    for (int run = 0; run < 5; ++run) {
        for (uint64_t i = 0; i < N; ++i) {
            s.find(i);
        }
    }
    auto end = clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "avg. time x find: "
              << elapsed.count() / (5 * N) * MILLION
              << " [musec]" << std::endl;

    return 0;
}
```

```cpp
#include <iostream>
#include <chrono>
#include <unordered_set>

#define MILLION 1000000

int main(int argc, char** argv) {

    if (argc < 2) {
        return 1;
    }

    size_t N = std::stoull(argv[1]);
    std::unordered_set<uint64_t> s;
    for (uint64_t i = 0; i < N; ++i) {
        s.insert(i);
    }

    typedef std::chrono::high_resolution_clock clock;

    auto start = clock::now();
    for (int run = 0; run < 5; ++run) {
        for (uint64_t i = 0; i < N; ++i) {
            s.find(i);
        }
    }
    auto end = clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "avg. time x find: "
              << elapsed.count() / (5 * N) * MILLION
              << " [musec]" << std::endl;

    return 0;
}
```

Associative containers

```cpp
#include <iostream>
#include <chrono>
#include <set>

#define MILLION 1000000

int main(int argc, char** argv) {

    if (argc < 2) {
        return 1;
    }

    size_t N = std::stoull(argv[1]);
    std::set<uint64_t> s;
    for (uint64_t i = 0; i < N; ++i) {
        s.insert(i);
    }

    typedef std::chrono::high_resolution_clock clock;

    auto start = clock::now();
    for (int run = 0; run < 5; ++run) {
        for (uint64_t i = 0; i < N; ++i) {
            s.find(i);
        }
    }
    auto end = clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "avg. time x find: "
              << elapsed.count() / (5 * N) * MILLION
              << " [musec]" << std::endl;

    return 0;
}
```

```
→  STL git:(master) ✗ ./set 5000000
avg. time x find: 0.338512 [musec]
```

```cpp
#include <iostream>
#include <chrono>
#include <unordered_set>

#define MILLION 1000000

int main(int argc, char** argv) {

    if (argc < 2) {
        return 1;
    }

    size_t N = std::stoull(argv[1]);
    std::unordered_set<uint64_t> s;
    for (uint64_t i = 0; i < N; ++i) {
        s.insert(i);
    }

    typedef std::chrono::high_resolution_clock clock;

    auto start = clock::now();
    for (int run = 0; run < 5; ++run) {
        for (uint64_t i = 0; i < N; ++i) {
            s.find(i);
        }
    }
    auto end = clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "avg. time x find: "
              << elapsed.count() / (5 * N) * MILLION
              << " [musec]" << std::endl;

    return 0;
}
```

Associative containers

```cpp
#include <iostream>
#include <chrono>
#include <set>

#define MILLION 1000000

int main(int argc, char** argv) {

    if (argc < 2) {
        return 1;
    }

    size_t N = std::stoull(argv[1]);
    std::set<uint64_t> s;
    for (uint64_t i = 0; i < N; ++i) {
        s.insert(i);
    }

    typedef std::chrono::high_resolution_clock clock;

    auto start = clock::now();
    for (int run = 0; run < 5; ++run) {
        for (uint64_t i = 0; i < N; ++i) {
            s.find(i);
        }
    }
    auto end = clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "avg. time x find: "
              << elapsed.count() / (5 * N) * MILLION
              << " [musec]" << std::endl;

    return 0;
}
```

```
→  STL git:(master) ✗ ./set 5000000
avg. time x find: 0.338512 [musec]
```

```cpp
#include <iostream>
#include <chrono>
#include <unordered_set>

#define MILLION 1000000

int main(int argc, char** argv) {

    if (argc < 2) {
        return 1;
    }

    size_t N = std::stoull(argv[1]);
    std::unordered_set<uint64_t> s;
    for (uint64_t i = 0; i < N; ++i) {
        s.insert(i);
    }

    typedef std::chrono::high_resolution_clock clock;

    auto start = clock::now();
    for (int run = 0; run < 5; ++run) {
        for (uint64_t i = 0; i < N; ++i) {
            s.find(i);
        }
    }
    auto end = clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "avg. time x find: "
              << elapsed.count() / (5 * N) * MILLION
              << " [musec]" << std::endl;

    return 0;
}
```

```
→  STL git:(master) ✗ ./unordered_set 5000000
avg. time x find: 0.082745 [musec]
```

Associative containers

```cpp
#include <iostream>
#include <chrono>
#include <set>

#define MILLION 1000000

int main(int argc, char** argv) {

    if (argc < 2) {
        return 1;
    }

    size_t N = std::stoull(argv[1]);
    std::set<uint64_t> s;
    for (uint64_t i = 0; i < N; ++i) {
        s.insert(i);
    }

    typedef std::chrono::high_resolution_clock clock;

    auto start = clock::now();
    for (int run = 0; run < 5; ++run) {
        for (uint64_t i = 0; i < N; ++i) {
            s.find(i);
        }
    }
    auto end = clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "avg. time x find: "
              << elapsed.count() / (5 * N) * MILLION
              << " [musec]" << std::endl;

    return 0;
}
```

```
→  STL git:(master) ✗ ./set 5000000
avg. time x find: 0.338512 [musec]
```

```cpp
#include <iostream>
#include <chrono>
#include <unordered_set>

#define MILLION 1000000

int main(int argc, char** argv) {

    if (argc < 2) {
        return 1;
    }

    size_t N = std::stoull(argv[1]);
    std::unordered_set<uint64_t> s;
    for (uint64_t i = 0; i < N; ++i) {
        s.insert(i);
    }

    typedef std::chrono::high_resolution_clock clock;

    auto start = clock::now();
    for (int run = 0; run < 5; ++run) {
        for (uint64_t i = 0; i < N; ++i) {
            s.find(i);
        }
    }
    auto end = clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "avg. time x find: "
              << elapsed.count() / (5 * N) * MILLION
              << " [musec]" << std::endl;

    return 0;
}
```

X4

```
→  STL git:(master) ✗ ./unordered_set 5000000
avg. time x find: 0.082745 [musec]
```

Iterators

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.

Operations:

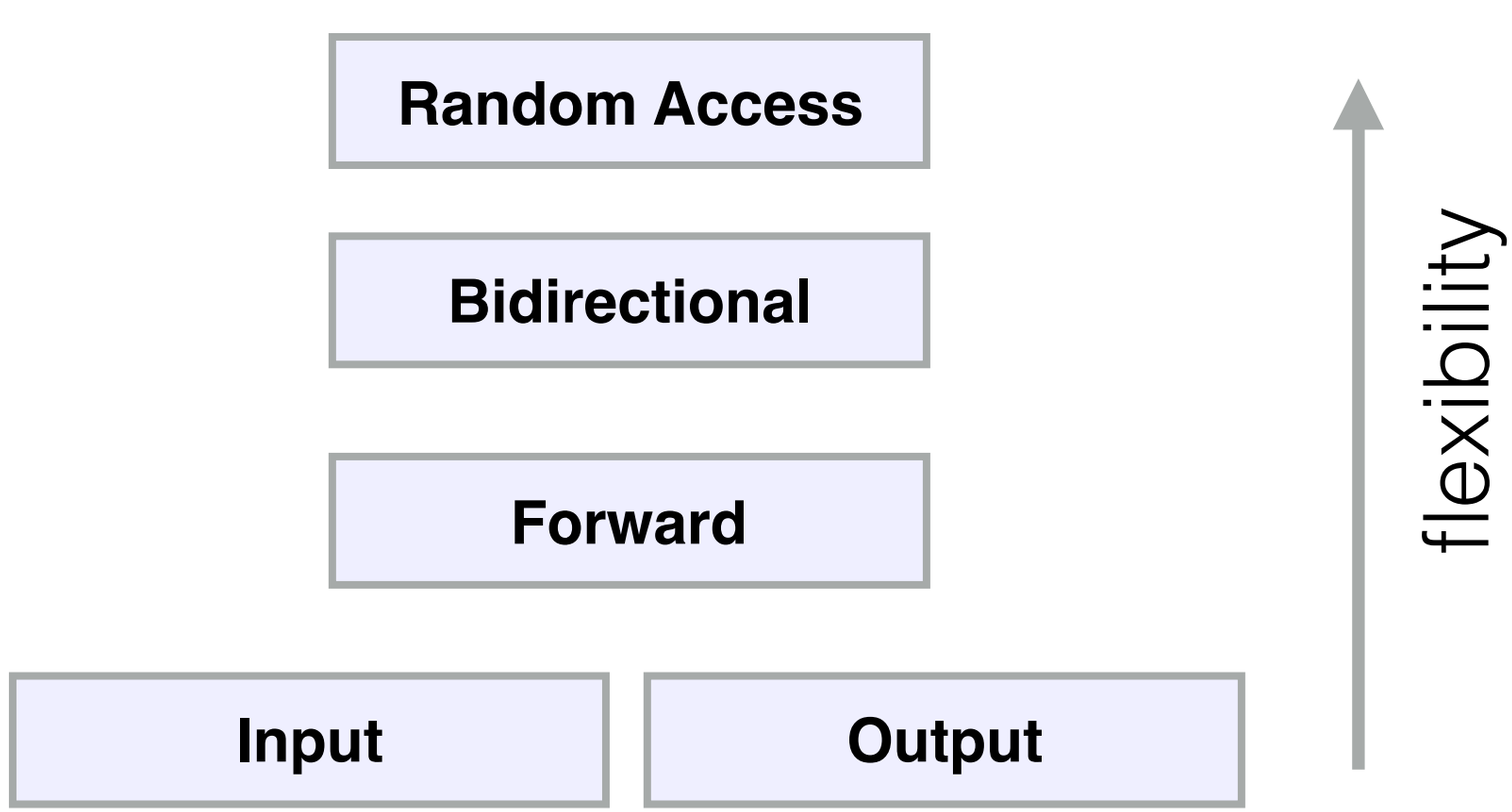
```
advance
distance
begin
end
prev
next
```

Iterators

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.

Operations:

```
advance
distance
begin
end
prev
next
```



<algorithm>

Algorithms

+ 20 with C++11

- all_of C++11
- any_of C++11
- none_of C++11
- for_each
- find
- find_if
- find_if_not C++11
- find_end
- find_first_of
- adjacent_find
- count
- count_if
- mismatch
- equal
- is_permutation C++11
- search
- search_n

- min
- max
- minmax C++11
- min_element
- max_element
- minmax_element C++11

- lower_bound
- upper_bound
- equal_range
- binary_search
- lexicographical_compare
- next_permutation
- prev_permutation

- push_heap
- pop_heap
- make_heap
- sort_heap
- is_heap C++11
- is_heap_until C++11

- merge
- inplace_merge
- includes
- set_union
- set_intersection
- set_difference
- set_symmetric_difference

- generate_n
- remove
- remove_if
- remove_copy
- remove_copy_if
- unique
- unique_copy
- reverse
- reverse_copy
- rotate
- rotate_copy
- random_shuffle
- shuffle C++11

- sort
- stable_sort
- partial_sort
- partial_sort_copy
- is_sorted C++11
- is_sorted_until C++11
- nth_element

- copy
- copy_n C++11
- copy_if C++11
- copy_backward
- move C++11
- move_backward C++11
- swap
- swap_ranges
- iter_swap
- transform
- replace
- replace_if
- replace_copy
- replace_copy_if
- fill
- fill_n
- generate

- is_partitioned C++11
- partition
- stable_partition
- partition_copy C++11
- partition_point C++11

std::sort

function template

std::sort

<algorithm>

default (1)	`template <class RandomAccessIterator>` `  void sort (RandomAccessIterator first, RandomAccessIterator last);`
custom (2)	`template <class RandomAccessIterator, class Compare>` `  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);`

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

struct pow2_comparator {
    bool operator()(int const x, int const y) {
        bool a = is_pow2(x);
        bool b = is_pow2(y);
        if (a != b) {
            return a < b;
        }
        return x > y;
    }

private:
    bool is_pow2(int x) {
        return (x & (x - 1)) == 0;
    }
};
```

```cpp
int main() {

    int a[] = {0, 3, 12, 8, 9, 23, 34, 1, 7, 16, 12, 2, 10, 112, 22};
    // int N = sizeof(a) / sizeof(a[0]);
    int N = std::distance(std::begin(a), std::end(a));

    // std::vector<int> vec(std::begin(a), std::end(a));
    // std::vector<int> vec(a, a + N);
    std::vector<int> vec;
    vec.reserve(N);
    std::for_each(std::begin(a), std::end(a),
        [&vec](const int x) {
            vec.push_back(x);
        }
    );

    std::sort(vec.begin(), vec.end(),
        [](int const x, int const y) {
            int mod1 = x % 2;
            int mod2 = y % 2;
            if (mod1 != mod2) {
                return mod1 < mod2;
            } else {
                return x < y;
            }
        }
    );

    std::for_each(vec.begin(), vec.end(),
        [](int x) {
            std::cout << x << " ";
        }
    );
    std::cout << "\n";

    pow2_comparator comp;
    std::sort(vec.begin(), vec.end(), comp);

    std::for_each(vec.begin(), vec.end(),
        [](int x) {
            std::cout << x << " ";
        }
    );
    std::cout << std::endl;
}
```

std::sort

function template

std::sort

<algorithm>

default (1)	`template <class RandomAccessIterator>` `  void sort (RandomAccessIterator first, RandomAccessIterator last);`
custom (2)	`template <class RandomAccessIterator, class Compare>` `  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);`

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

struct pow2_comparator {
    bool operator()(int const x, int const y) {
        bool a = is_pow2(x);
        bool b = is_pow2(y);
        if (a != b) {
            return a < b;
        }
        return x > y;
    }

private:
    bool is_pow2(int x) {
        return (x & (x - 1)) == 0;
    }
};
```

```cpp
int main() {

    int a[] = {0, 3, 12, 8, 9, 23, 34, 1, 7, 16, 12, 2, 10, 112, 22};
    // int N = sizeof(a) / sizeof(a[0]);
    int N = std::distance(std::begin(a), std::end(a));

    // std::vector<int> vec(std::begin(a), std::end(a));
    // std::vector<int> vec(a, a + N);
    std::vector<int> vec;
    vec.reserve(N);
    std::for_each(std::begin(a), std::end(a),
        [&vec](const int x) {
            vec.push_back(x);
        }
    );

    std::sort(vec.begin(), vec.end(),
        [](int const x, int const y) {
            int mod1 = x % 2;
            int mod2 = y % 2;
            if (mod1 != mod2) {
                return mod1 < mod2;
            } else {
                return x < y;
            }
        }
    );

    std::for_each(vec.begin(), vec.end(),
        [](int x) {
            std::cout << x << " ";
        }
    );          // 0 2 8 10 12 12 16 22 34 112 1 3 7 9 23
    std::cout << "\n";

    pow2_comparator comp;
    std::sort(vec.begin(), vec.end(), comp);

    std::for_each(vec.begin(), vec.end(),
        [](int x) {
            std::cout << x << " ";
        }
    );
    std::cout << std::endl;
}
```

std::sort

function template

std::sort

<algorithm>

default (1)	`template <class RandomAccessIterator>` `  void sort (RandomAccessIterator first, RandomAccessIterator last);`
custom (2)	`template <class RandomAccessIterator, class Compare>` `  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);`

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

struct pow2_comparator {
    bool operator()(int const x, int const y) {
        bool a = is_pow2(x);
        bool b = is_pow2(y);
        if (a != b) {
            return a < b;
        }
        return x > y;
    }

private:
    bool is_pow2(int x) {
        return (x & (x - 1)) == 0;
    }
};
```

```cpp
int main() {

    int a[] = {0, 3, 12, 8, 9, 23, 34, 1, 7, 16, 12, 2, 10, 112, 22};
    // int N = sizeof(a) / sizeof(a[0]);
    int N = std::distance(std::begin(a), std::end(a));

    // std::vector<int> vec(std::begin(a), std::end(a));
    // std::vector<int> vec(a, a + N);
    std::vector<int> vec;
    vec.reserve(N);
    std::for_each(std::begin(a), std::end(a),
        [&vec](const int x) {
            vec.push_back(x);
        }
    );

    std::sort(vec.begin(), vec.end(),
        [](int const x, int const y) {
            int mod1 = x % 2;
            int mod2 = y % 2;
            if (mod1 != mod2) {
                return mod1 < mod2;
            } else {
                return x < y;
            }
        }
    );

    std::for_each(vec.begin(), vec.end(),
        [](int x) {
            std::cout << x << " ";
        }
    );          // 0 2 8 10 12 12 16 22 34 112 1 3 7 9 23
    std::cout << "\n";

    pow2_comparator comp;
    std::sort(vec.begin(), vec.end(), comp);

    std::for_each(vec.begin(), vec.end(),
        [](int x) {
            std::cout << x << " ";
        }
    );          // 112 34 23 22 12 12 10 9 7 3 16 8 2 1 0
    std::cout << std::endl;
}
```

Another example

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

int main() {

    int a[] = {39, 43, 3, 1, 7, 36, 10, 58, 15, 23, 61, 46, 24};

    std::vector<int> vec(std::begin(a), std::end(a));
    std::sort(vec.begin(), vec.end());

    auto it = std::upper_bound(vec.begin(), vec.end(), vec.back() / 2);
    int val = *it;

    std::for_each(it + 1, vec.end(),
        [val](int& x) {
            x = x % val;
        }
    );

    std::sort(vec.begin(), vec.end());
    auto end = std::unique(vec.begin(), vec.end());

    for (auto it = vec.begin(); it != end; ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Another example

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

int main() {

    int a[] = {39, 43, 3, 1, 7, 36, 10, 58, 15, 23, 61, 46, 24};

    std::vector<int> vec(std::begin(a), std::end(a));
    std::sort(vec.begin(), vec.end());

    auto it = std::upper_bound(vec.begin(), vec.end(), vec.back() / 2);
    int val = *it;

    std::for_each(it + 1, vec.end(),
        [val](int& x) {
            x = x % val;
        }
    );

    std::sort(vec.begin(), vec.end());
    auto end = std::unique(vec.begin(), vec.end());

    for (auto it = vec.begin(); it != end; ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

```
[→  STL git:(master) ✗ g++ -std=c++11 algs_example.cpp -o algs_example
[→  STL git:(master) ✗ ./algs_example
1 3 7 10 15 22 23 24 25 36
→  STL git:(master) ✗
```

References

http://www.cplusplus.com/