Standard Template Library

Контейнеры. Алгоритмы.

2018

CQG

О чем эта лекция?

1. **Контейнеры (продолжение).**
   - Ассоциативные контейнеры

2. **Функциональные объекты**
   - функторы, функциональные объекты
   - binders

3. **Алгоритмы**
   - Классификация
   - Адаптеры итераторов
   - Удаление элементов. Идиома «erase-remove»

STL: Containers

Последовательные

ОСНОВНЫЕ:

vector (расширяемый массив)
deque (двусторонняя очередь)
list (двусвязный список)

STL: Containers

Последовательные

ОСНОВНЫЕ:

vector (расширяемый массив)
deque (двусторонняя очередь)
list (двусвязный список)

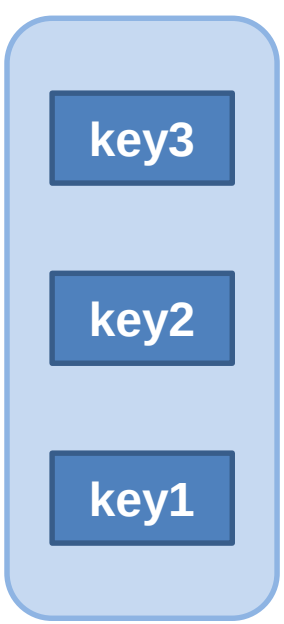
Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)

STL: Containers

Ассоциативные

set (отсортированное множество)



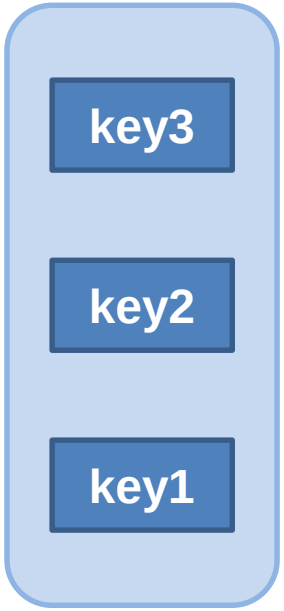

```
#include <set>

using Collection = std::set<int>;

Collection s;

s.insert( 8 );
s.insert( -1 );
s.insert( 3 );

Collection::const_iterator iter;
for( iter = s.begin(); iter != s.end(); ++iter )
{
    cout << *iter << " ";
}
```

```
-1  3  8
```

STL: Containers

Ассоциативные

set (отсортированное множество)

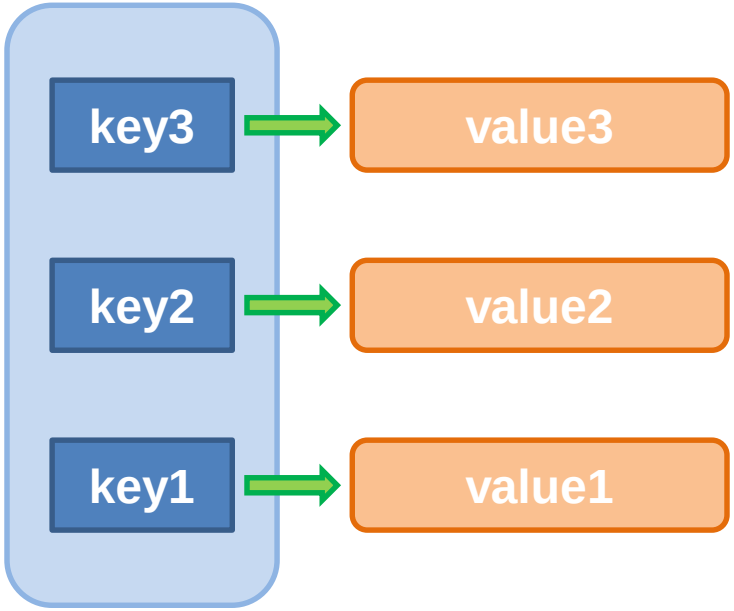



```
#include <set>

using Collection = std::set<int>;

Collection s;

s.insert( 8 );
s.insert( -1 );
s.insert( 3 );

Collection::iterator iter = s.find( 4 );
if( iter != s.end() )
{
	std::cout << *iter << std::endl;
}
```

STL: Containers

Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)




```
#include <map>

using Collection = std::map<int, double>;

Collection m;
```

STL: Containers

Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)


```
#include <map>

using Collection = std::map<int, double>;

Collection m;

m[7] = 1.2;
```

STL: Containers

Ассоциативные

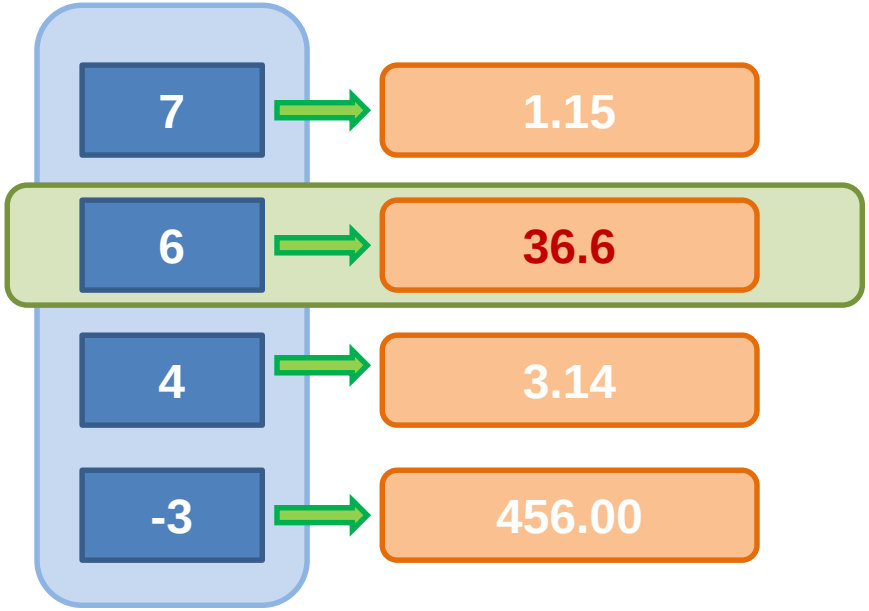
set (отсортированное множество)
map (словарь: ключ -> значение)


```
#include <map>

using Collection = std::map<int, double>;

Collection m;

m[7] = 1.2;

m[7] = 1.15;
```

STL: Containers

Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)

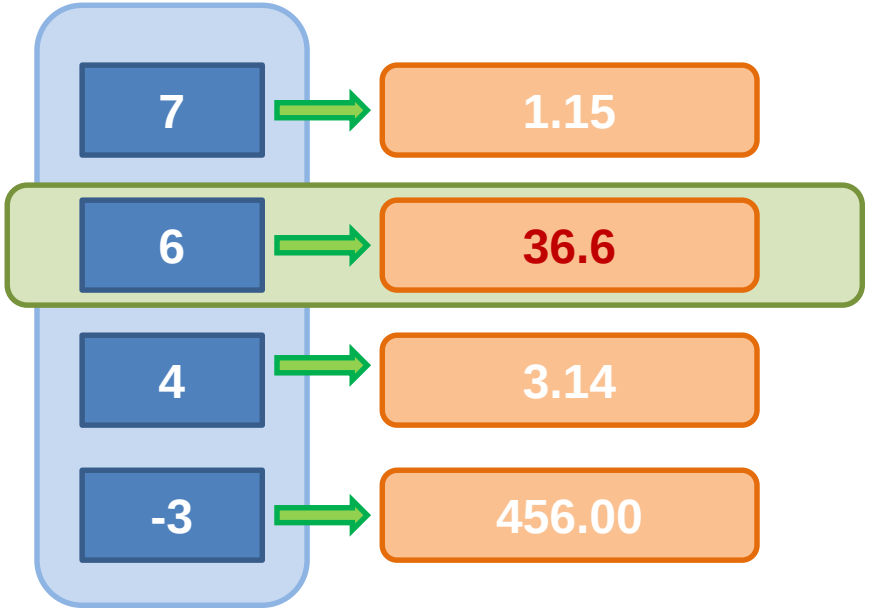



```
#include <map>

using Collection = std::map<int, double>;

Collection m;

m.insert( std::make_pair(6, 36.6) );

// or

m.emplace( 6, 36.6 );
```

STL: Containers

Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)

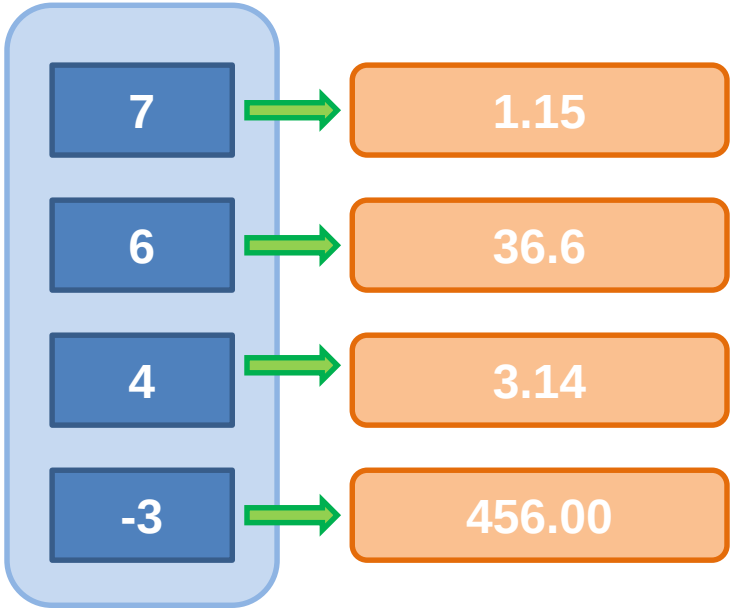



```
#include <map>

using Collection = std::map<int, double>;

Collection m;

m.insert( std::make_pair(6, 36.6) );

template< typename T1, typename T2 >
struct pair
{
   T1 first;
   T2 second;

   // other stuff
};
```

STL: Containers

Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)

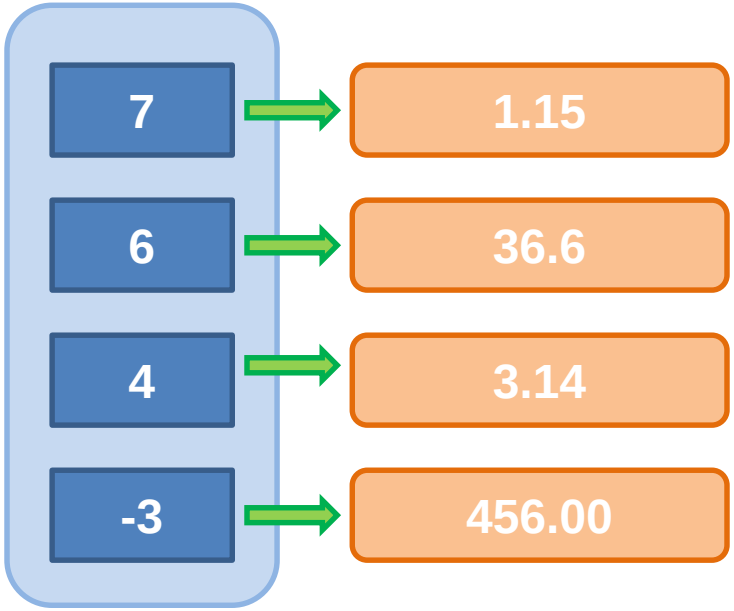



```
#include <map>

using Collection = std::map<int, double>;

Collection m;

m.insert( std::make_pair(6, 36.6) );

m.insert( std::make_pair(6, 37.1) );
```

STL: Containers

Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)

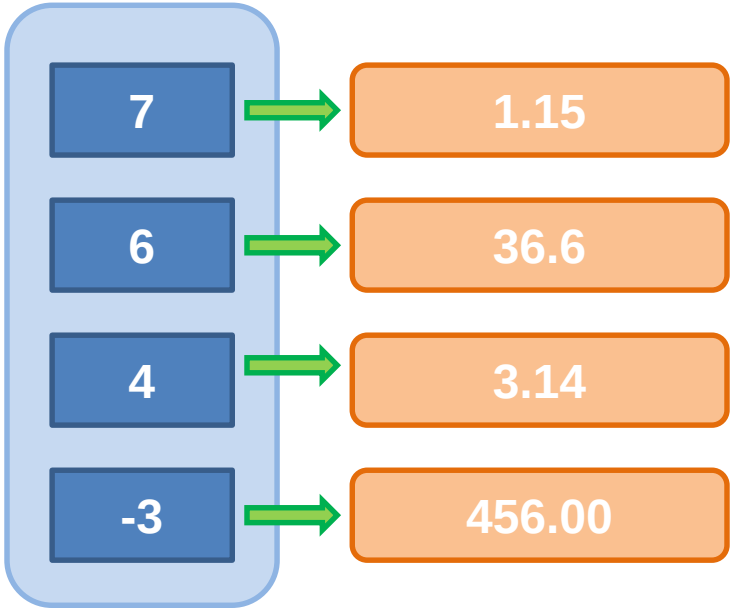



```
#include <map>

using Collection = std::map<int, double>;

Collection m;

m.insert( std::make_pair(6, 36.6) );

std::pair< Collection::iterator, bool > res =
        m.insert( std::make_pair(6, 37.1) );

bool bInsertedOk = res.second;
```

STL: Containers

Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)




```
#include <map>

using Collection = std::map<int, double>;

Collection m;

m.insert( std::make_pair(6, 36.6) );

Collection::iterator  iter =  m.find( 4 );

If( iter !=  m.end() )
{
      std::cout << (*iter).first << " " <<
                        (*iter).second << std::endl
}
```

STL: Containers

Последовательные

ОСНОВНЫЕ:

vector (расширяемый массив)
deque (двусторонняя очередь)
list (двусвязный список)

Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)

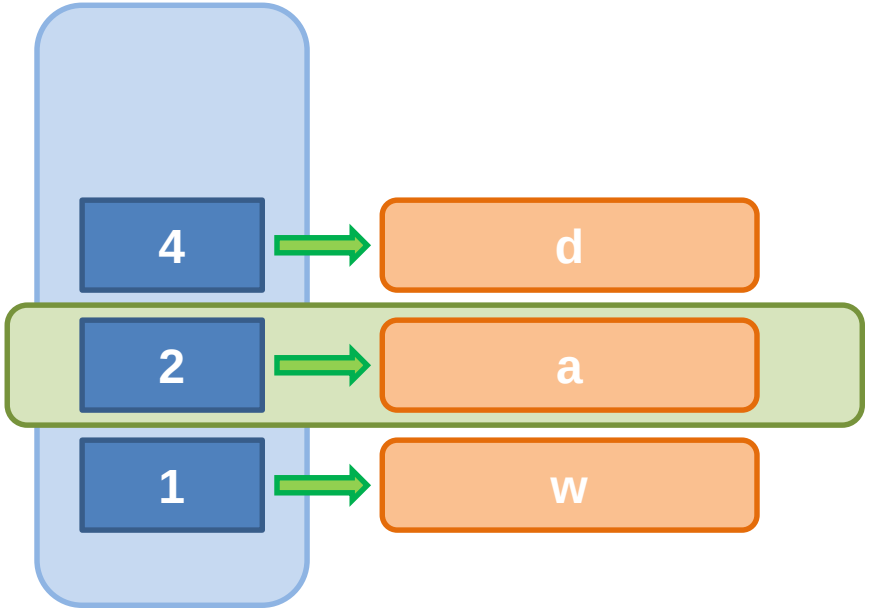
multiset (множество с дубликатами)
multimap (словарь с дубликатами)

CQG

STL: Containers

Ассоциативные

multimap (словарь: ключ -> значение; ключи могут повторяться)

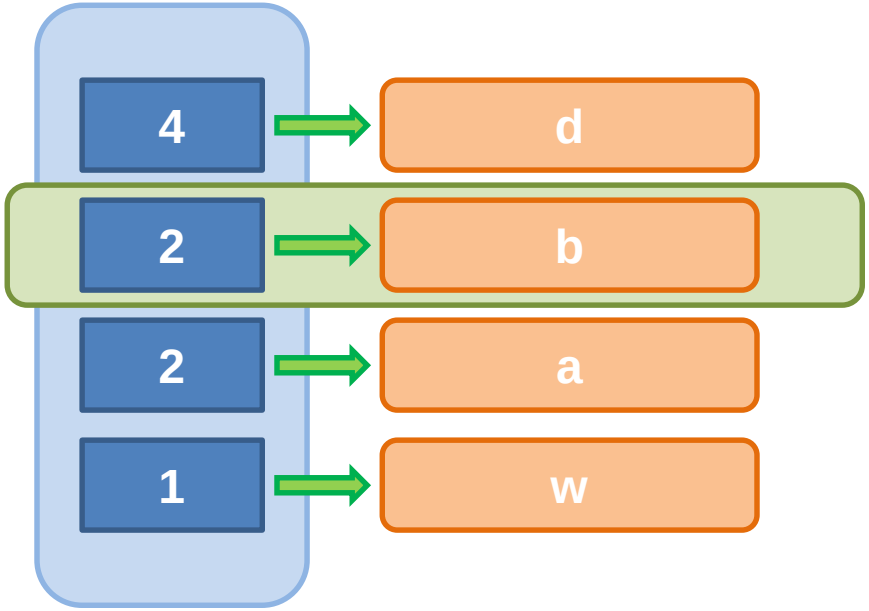



```
#include <map>

using Collection = std::multimap<int, double>;

Collection m;

m.insert( std::make_pair(2, ‘a’) );
```

STL: Containers

Ассоциативные

multimap (словарь: ключ -> значение; ключи могут повторяться)

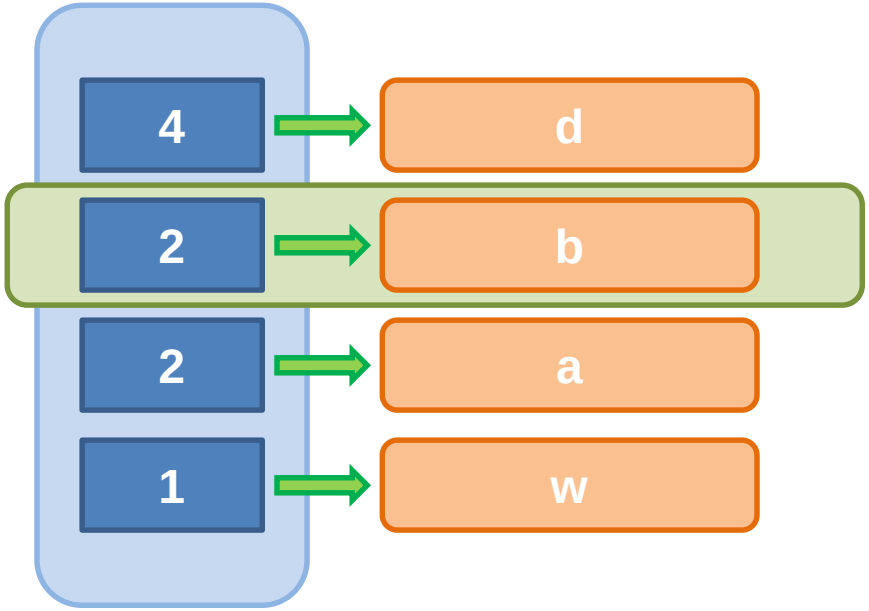



```
#include <map>

using Collection = std::multimap<int, double>;

Collection m;

m.insert( std::make_pair(2, ‘a’) );
m.insert( std::make_pair(2, ‘b’) );
```

STL: Containers

Ассоциативные

multimap (словарь: ключ -> значение; ключи могут повторяться)

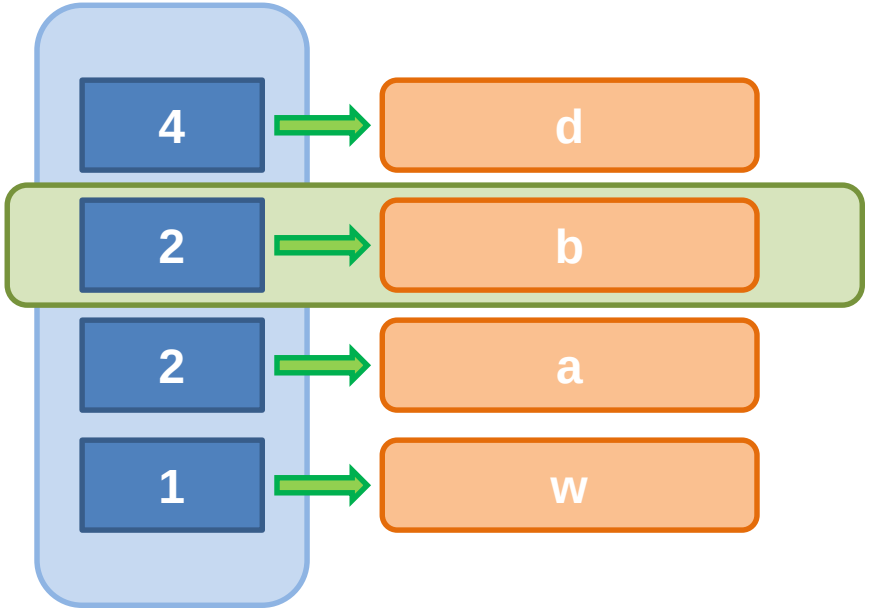



```
#include <map>

using Collection = std::multimap<int, double>;

Collection m;

m.insert( std::make_pair(2, ‘a’) );
m.insert( std::make_pair(2, ‘b’) );

// возвращается не пара iterator-bool, а просто
iterator, поскольку вставка с тем же ключом
всегда возможна
```

STL: Containers

Ассоциативные

multimap (словарь: ключ -> значение; ключи могут повторяться)

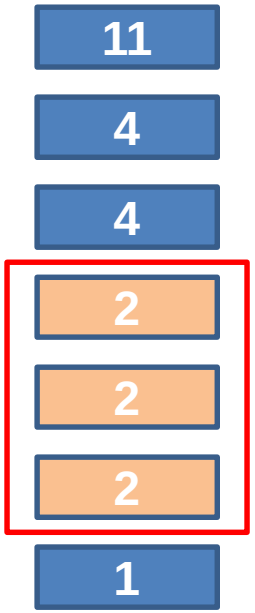



```
#include <map>

using Collection = std::multimap<int, double>;

Collection m;

m.insert( std::make_pair(2, ‘a’) );
m.insert( std::make_pair(2, ‘b’) );
// возвращается не пара iterator-bool, а просто
iterator, поскольку вставка с тем же ключом
всегда возможна

// увы, больше нет m[2] = ‘b’;
```

STL: Containers

Ассоциативные

multimap (словарь: ключ -> значение; ключи могут повторяться)



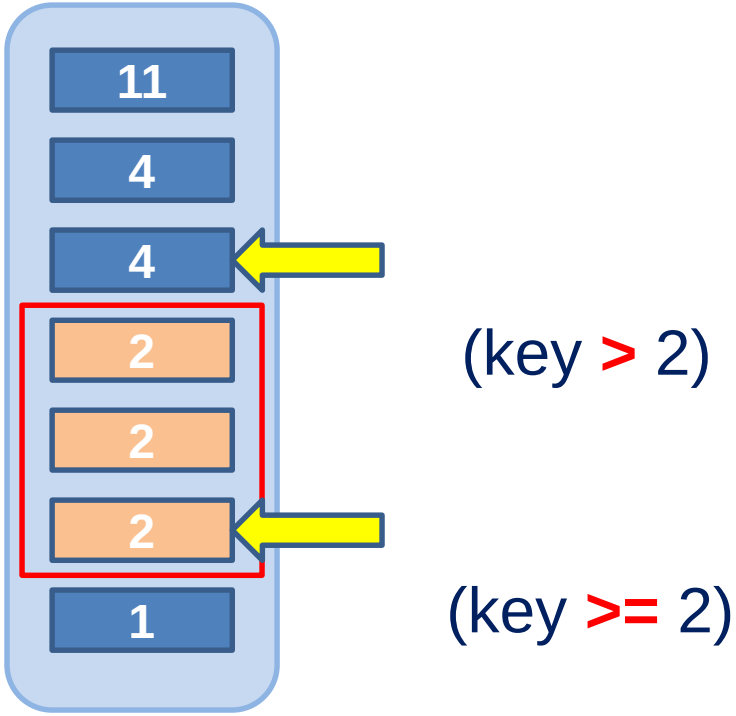
key == 2

```
#include <map>

using Collection = std::multimap<int, double>;
using Iterator = Collection::iterator;

Collection m;

Iterator  it1 =   m.lower_bound(2);
Iterator  it2 =   m.upper_bound(2);
```

STL: Containers

Ассоциативные

multimap (словарь: ключ -> значение; ключи могут повторяться)

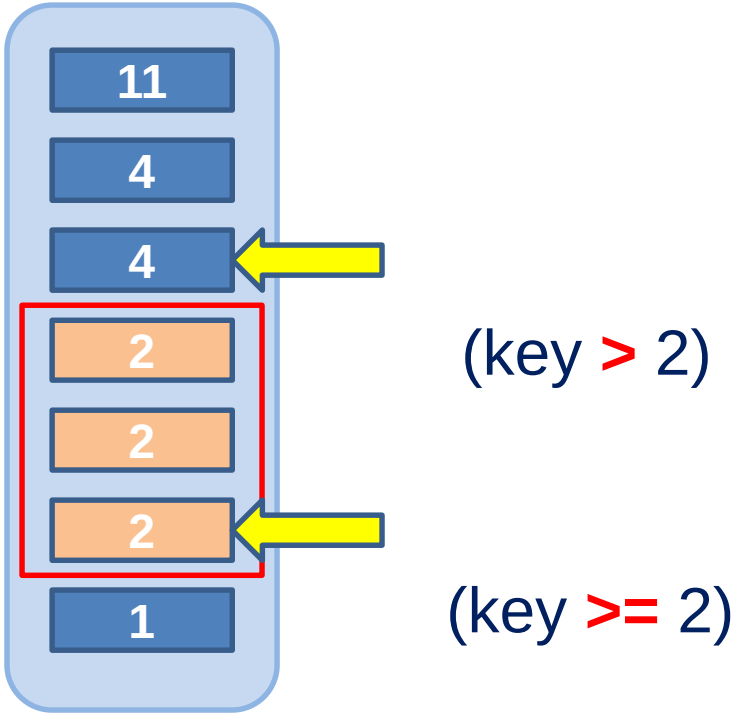



```
#include <map>

using Collection = std::multimap<int, double>;
using Iterator = Collection::iterator;

Collection m;

Iterator  it1 =   m.lower_bound(2);
Iterator  it2 =   m.upper_bound(2);
```

STL: Containers

Ассоциативные

multimap (словарь: ключ -> значение; ключи могут повторяться)

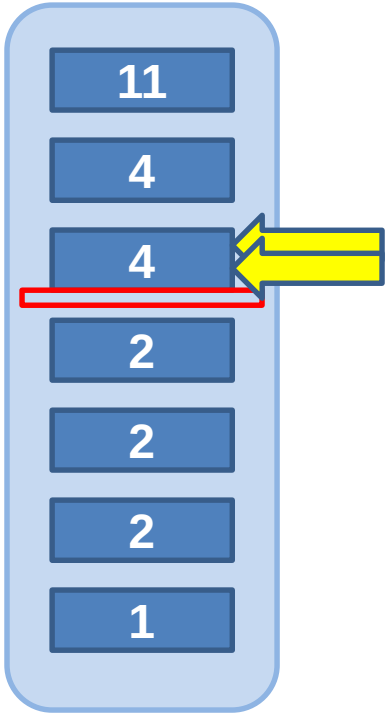



```
#include <map>

using Collection = std::multimap<int, double>;
using Iterator = Collection::iterator;

Collection m;

std::pair<Iterator, Iterator > range
                    =   m.equal_range(2);
```

STL: Containers

Ассоциативные

multimap (словарь: ключ -> значение; ключи могут повторяться)



(key **>** 3)

(key **>=** 3)

```
#include <map>

using Collection = std::multimap<int, double>;
using Iterator = Collection::iterator;

Collection m;

std::pair<Iterator, Iterator > range
                    =   m.equal_range(3);

If( range.first  ==  range.second  )
{
     // no elements
}
```

STL: Containers

Последовательные

vector (расширяемый массив)
deque (двусторонняя очередь)
list (двусвязный список)

Ассоциативные

set (отсортированное множество)
map (словарь: ключ -> значение)

multiset (множество с дубликатами)
multimap (словарь с дубликатами)

unordered_set (hashed)
unordered_map (hashed)

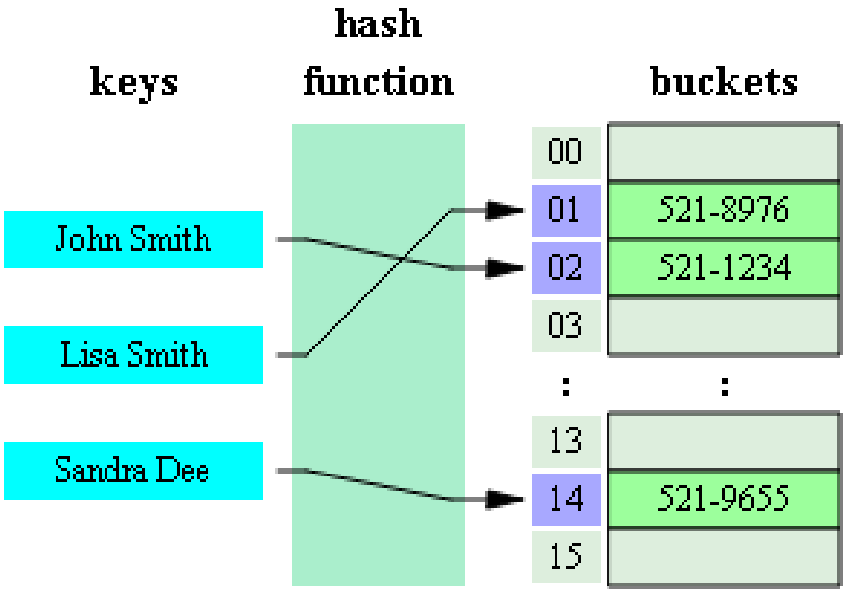
unordered_multiset (hashed)
unordered_multimap (hashed)

CQG

STL: Containers

Хэш-таблица

Хэш-функция: для каждого значения ключа вычисляем число, которое будет использовано как индекс в таблице:



Скорость доступа/поиска – **O(1)** (плюс время вычисления hash функции).

STL: Containers

Хэш-таблица

Хэш-функция: как гарантировать, что для каждого значения ключа индекс в таблице будет уникальным?

Ответ – никак. Могут быть совпадения (т.н. «коллизии»).

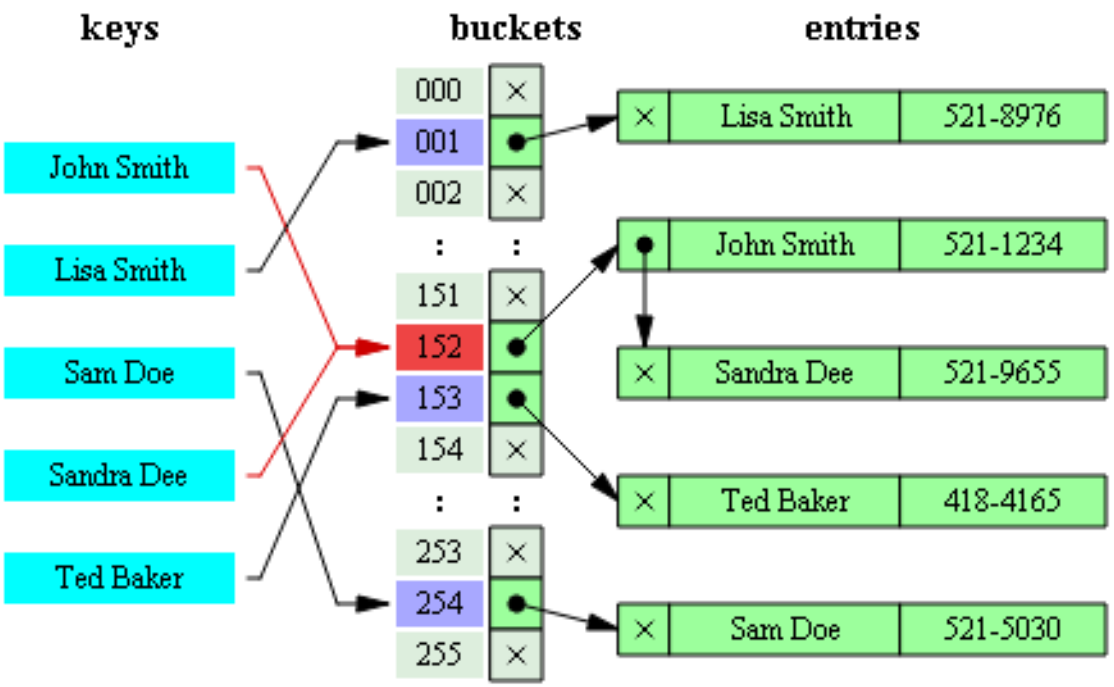
<уныние>И что теперь? Выкидывать хэш-таблицу на свалку? </уныние>

STL: Containers

Хэш-таблица: коллизии

Два метода разрешения коллизий:

“Chaining”

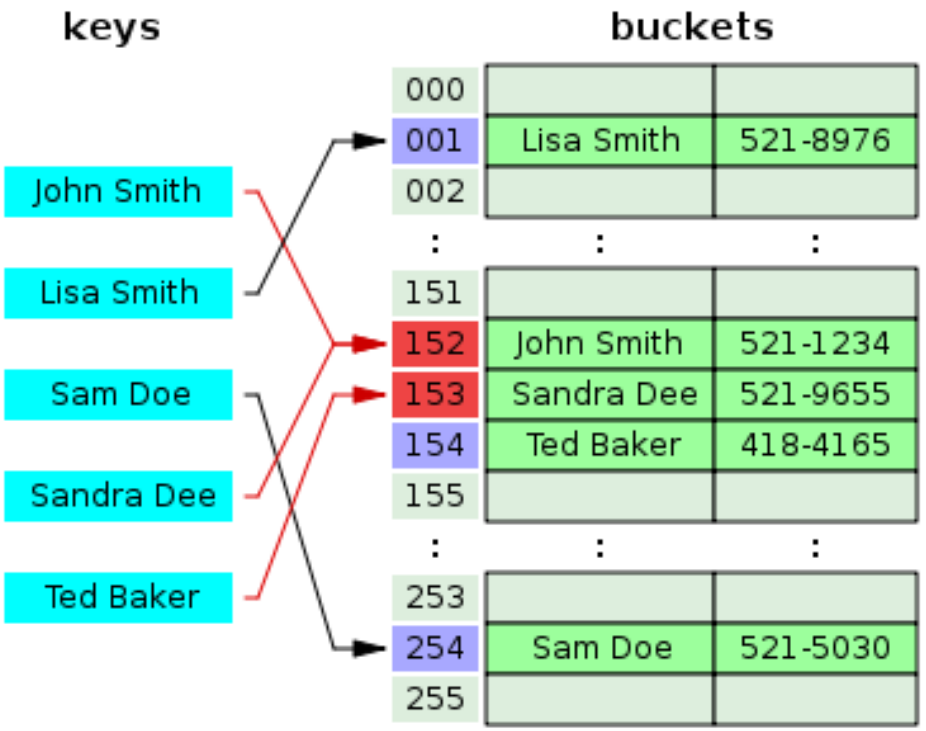


STL: Containers

Хэш-таблица: коллизии

Два метода разрешения коллизий:

“Open addressing”



STL: Containers

unordered_set

```
template<
    class Key,
    class Hasher = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

STL: Containers

unordered_set

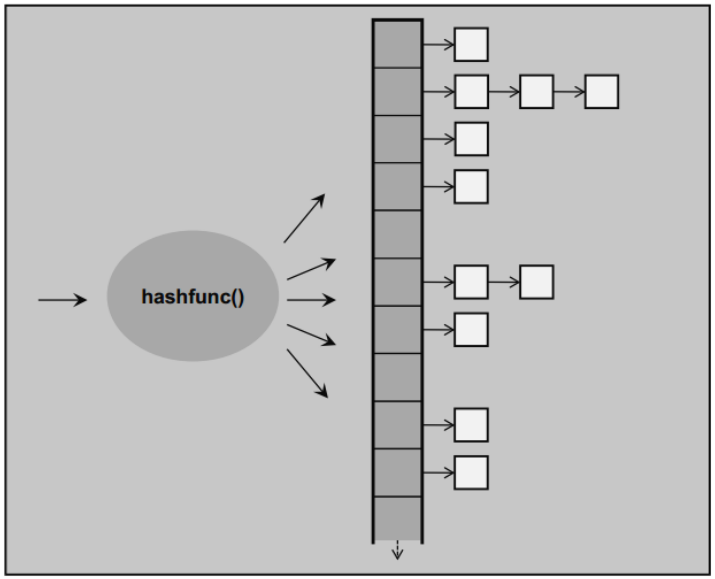
```
template<
    class Key,
    class Hasher = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

```
namespace std
{
template<>
class hash<Foo>
{
public:
    size_t operator()(const Foo& obj) const
    {
        size_t h1 = std::hash<string>()(obj.first_name);
        size_t h2 = std::hash<string>()(obj.last_name);
        return h1 ^ ( h2 << 1 );
    }
};
}
```

STL: Containers

unordered_set

```
template<
    class Key,
    class Hasher = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```



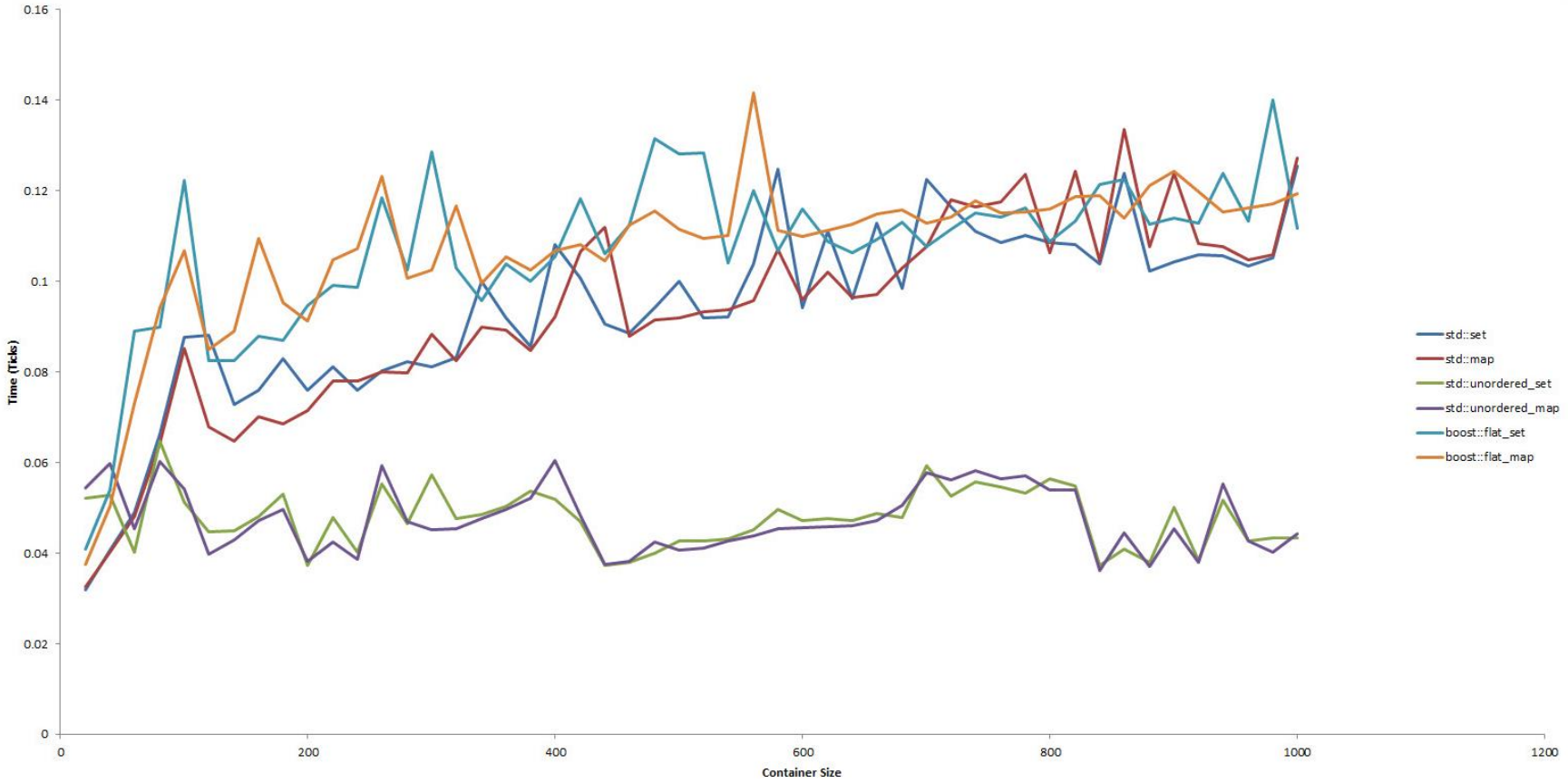
STL: Containers

unordered_map

```
template<
    class Key,
    class Value,
    class Hasher = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<…>
> class unordered_map;
```

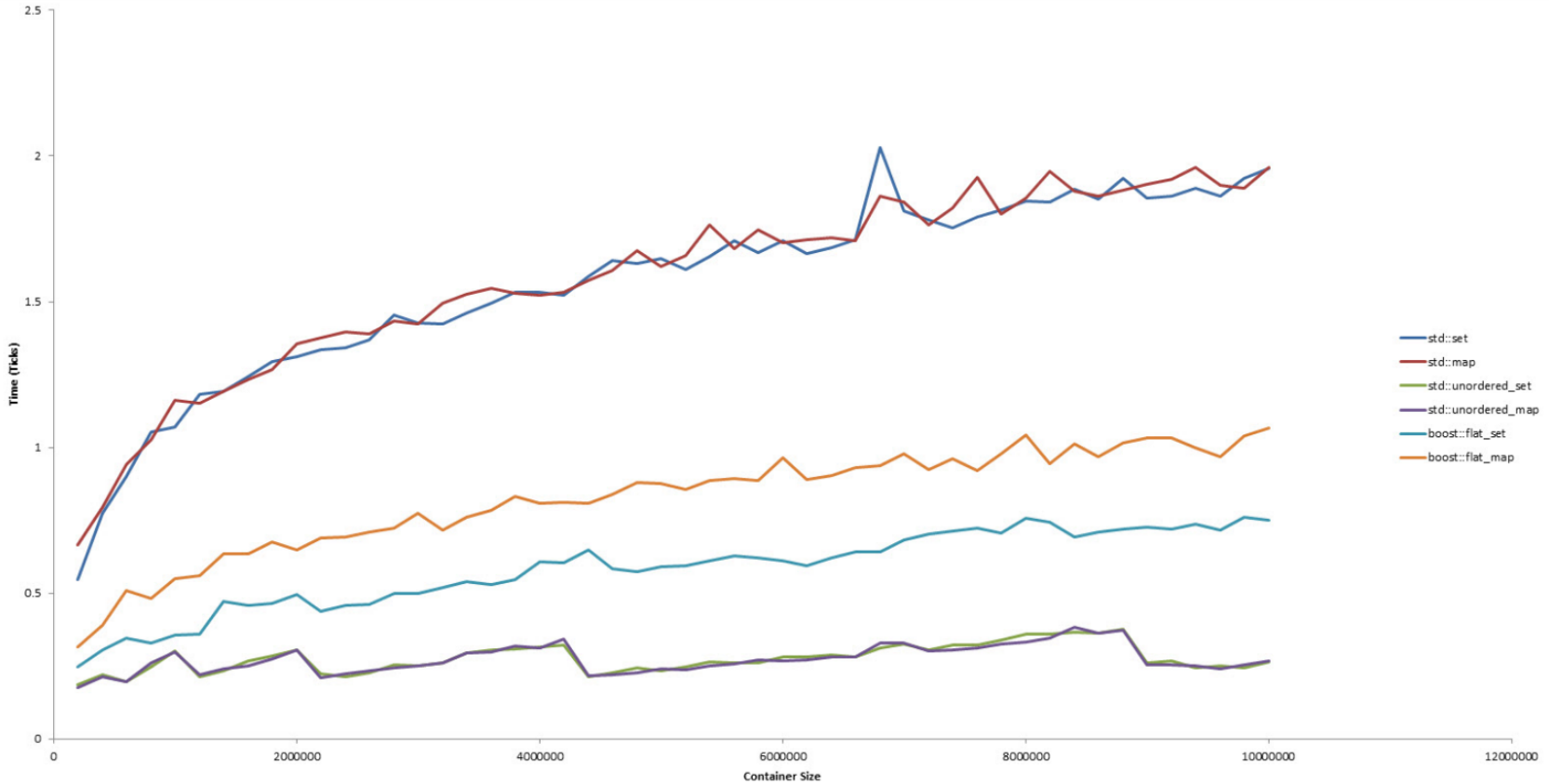
STL: Containers

unordered_set/unordered_map: performance



STL: Containers

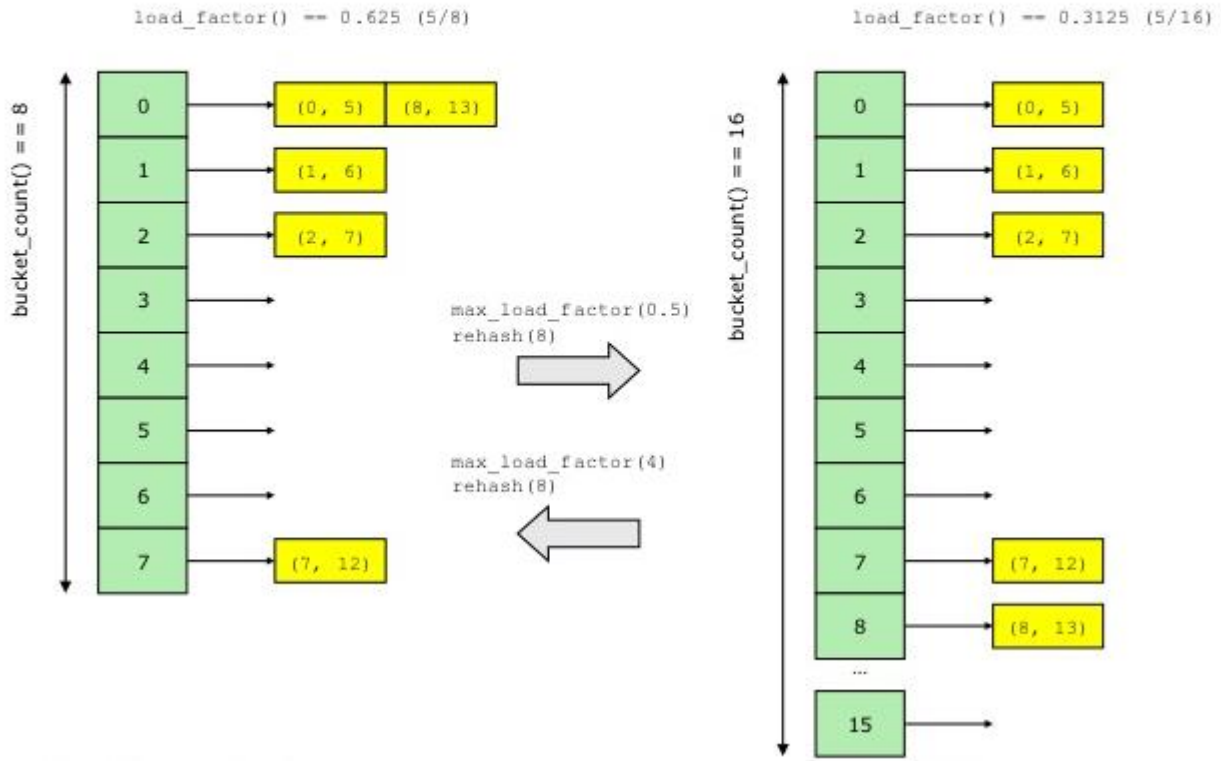
unordered_set/unordered_map: performance



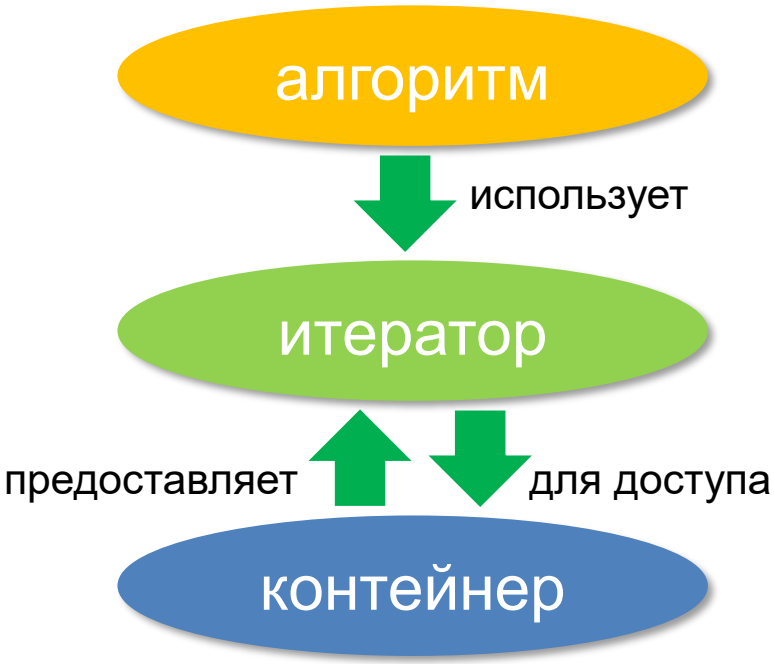
STL: Containers

Дополнительные возможности: перехэширование

```
cout << map2.load_factor();  // average load factor for a bucket; prints .625  (size()/bucket_count())
cout << map2.max_load_factor();  // prints 4
map2.max_load_factor(0.5); // sets  new target load factor
map2.rehash(8); // rehash such that the load factor does not exceed target load factor, add new buckets
map2.max_load_factor(4); // sets target load factor back to 4
map2.rehash(8); // rehash, get back to original state, get at least 8 buckets
```



STL: Algorithms



STL: Functional objects

Вернемся к нашему старому примеру: Функция, которая ищет первый элемент массива, равный заданному числу:

```
int* find1(int* array, int n, int x)
{
    int* p = array;
    for( int i = 0; i < n; ++i )
    {
        if ( *p == x ) return p; // success
        ++p;
    }
    return 0; // fail
}
```

STL: Functional objects

Вернемся к нашему старому примеру: Функция, которая ищет первый элемент массива, равный заданному числу:

```
int* find1(int* array, int n, int x)
{
    int* p = array;
    for( int i = 0; i < n; ++i )
    {
        if ( *p == x ) return p; // success
        ++p;
    }
    return 0; // fail
}
```

```
int A[100];
...
int* p = find1(A, 100, 5);
```

STL: Functional objects

Более общая проблема: Функция, которая ищет первый элемент массива, удовлетворяющий заданному условию:

```
int* find2(int* array, int n, bool (cond*)(int) )
{
    int* p = array;
    for( int i = 0; i < n; ++i )
    {
        if ( cond(*p ) ) return p; // success
        ++p;
    }
    return 0; // fail
}
```

Указатель на функцию

Вызов функции по указателю

STL: Functional objects

Другие примеры:

```
int A[100];

bool cond_less5( int x )
{
    return x < 5;
}
int* p = find2(A, 100, cond_less5);
```

STL: Functional objects

Другие примеры:

```
int A[100];

bool cond_less5( int x )
{
   return x < 5;
}
int* p = find2(A, 100, cond_less5);

bool cond_range_0_100( int x )
{
   return x >= 0 && x <= 100;
}
int* p = find2(A, 100, cond_range_0_100);
```

STL: Functional objects

Недостатки:

недостаточная гибкость

невысокое быстродействие (вызов функции не inline)

STL: Functional objects

Недостатки:

недостаточная гибкость

невысокое быстродействие (вызов функции не inline)

Решение:

функциональный объект

CQG

STL: Functional objects

Недостатки:

недостаточная гибкость

невысокое быстродействие (вызов функции не inline)

Решение:

функциональный объект

Определения:

Функциональный объект – объект функционального типа
Функциональный тип – тип, который предоставляет **operator()**(…)

STL: Functional objects

Функциональный вызов:

```
F( <argument list> )
```

Чем может быть **F** ?

STL: Functional objects

Функциональный вызов:

```
F( <argument list> )
```

Чем может быть **F** ?

Функция

```
int F( int x ) { return x*x; }

int a = F(1);
```

STL: Functional objects

Функциональный вызов:

```
F( <argument list> )
```

Чем может быть **F** ?

Функция

```
int F( int x ) { return x*x; }

int a = F(1);
```

Указатель на функцию

```
int (*pF)(int x) = F;

int b = pF(1);
```

STL: Functional objects

Чем может быть **F** ?

- Функция
- Указатель на функцию
- **Функциональный объект**

(объект типа, предоставляющего оператор вызова)

```
struct C
{
    int operator()(int x) { return x*x; }
};
```

STL: Functional objects

Чем может быть **F** ?

- Функция
- Указатель на функцию
- **Функциональный объект**

(объект типа, предоставляющего оператор вызова)

```
struct C
{
    int operator()(int x) { return x*x; }
};


C obj;

int a = obj(1);

int b = obj.operator()(1);
```

CQG

STL: Functional objects

```
struct greater_than_5
{
   bool operator()(int x) const { return x > 5; }
};
```

```
int* find2(int* array, int n, greater_than_5 obj )
{
   int* p = array;
   for( int i = 0; i < n; ++i )
   {
      if( obj(*p ) ) return p; // success
      ++p;
   }
   return 0; // fail
}
```

Передаем объект типа, для которого есть **operator()**

obj.operator()(*p)

CQG

STL: Functional objects

Обобщим наш «компаратор»:

```
template< typename T, T N >
struct greater
{
   bool operator()(T x) const { return x > N; }
};
```

```
int* find2(int* array, int n, greater<int, 5> obj )
{
   int* p = array;
   for( int i = 0; i < n; ++i )
   {
      if( obj(*p ) ) return p; // success
      ++p;
   }
   return 0; // fail
}
```

STL: Functional objects

Недостатки «алгоритма» :

```
int* find2(int* array, int n, greater<int, 5> obj )
{
   int* p = array;
   for( int i = 0; i < n; ++i )
   {
      if( obj(*p ) ) return p; // success
      ++p;
   }
   return 0; // fail
}
```

1. Ищет в массиве **int**
2. Ищет только значение, превышающее 5.

STL: Functional objects

Обобщим наш «алгоритм» :

```
template< typename T, typename Comparator >
T* find3( T* array, int n, Comparator obj )
{
   T* p = array;
   for( int i = 0; i < n; ++i )
   {
      if( obj(*p ) ) return p; // success
      ++p;
   }
   return 0; // fail
}
```

STL: Functional objects

```
template< typename T, T N >
struct greater
{
    b
};
```

```
template< typename T, T N >
struct greater_equal
{
    b
};
```

```
template< typename T, T N >
struct less_equal
{
    bool operator()(T x) const { return x <= N; }
};
```

```
int* p = find3( A, 100, greater<int,5>() );

int* q = find3( A, 100, greater_equal<int,10>() );

int* r = find3( A, 100, less<int,0>() );
```

CQG

STL: Functional objects

```
template<typename T> struct equal_to;
template<typename T> struct not_equal_to;
template<typename T> struct greater;
template<typename T> struct less;
template<typename T> struct greater_equal;
template<typename T> struct less_equal;
template<typename T> struct plus;
template<typename T> struct minus;
template<typename T> struct multiplies;
template<typename T> struct divides;
template<typename T> struct modulus;
template<typename T> struct negate;
template<typename T> struct logical_and;
template<typename T> struct logical_or;
template<typename T> struct logical_not;
```

STL: Functional objects

```
template < typename T >
struct greater
{
   bool operator()( T x, T y ) const // inline
   {
      return x > y;
   }
};
```

Упрощенно!

Пример использования:

```
std::greater<int> comp;
if( comp(x,5) )
{
   ...
}
```

STL: Functional objects

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

STL: Functional objects

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

```
template< class Arg1, class Arg2, class Result >
struct binary_function
{
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};
```

CQG

STL: Functional objects

```
std::greater<int> comp;
if( comp(a, 5) )
{
    ...
}

std::binary_negate< std::greater<int> > neg_comp(comp);
if( neg_comp(a, 5) )
{
    ...
}
```

```
template< class Fn >
class binary_negate :
   public binary_function< typename Fn::first_argument_type,
                           typename Fn::second_argument_type,
                           bool>
{
   ...
};
```

CQG

STL: Algorithms

Алгоритмы (почти все) являются **внешними по отношению к классам контейнеров .**

Алгоритмы объявляются как **шаблонные функции** (шаблонными аргументами которых являются типы передаваемых итераторов)

В качестве **аргументов** алгоритм принимает не контейнер, а **набор итераторов.**

Алгоритм может быть применен к любой структуре данных, **если итераторы**, предоставляемые ей, **удовлетворяют требованиям, которые данный алгоритм предъявляет** к итераторам**.**

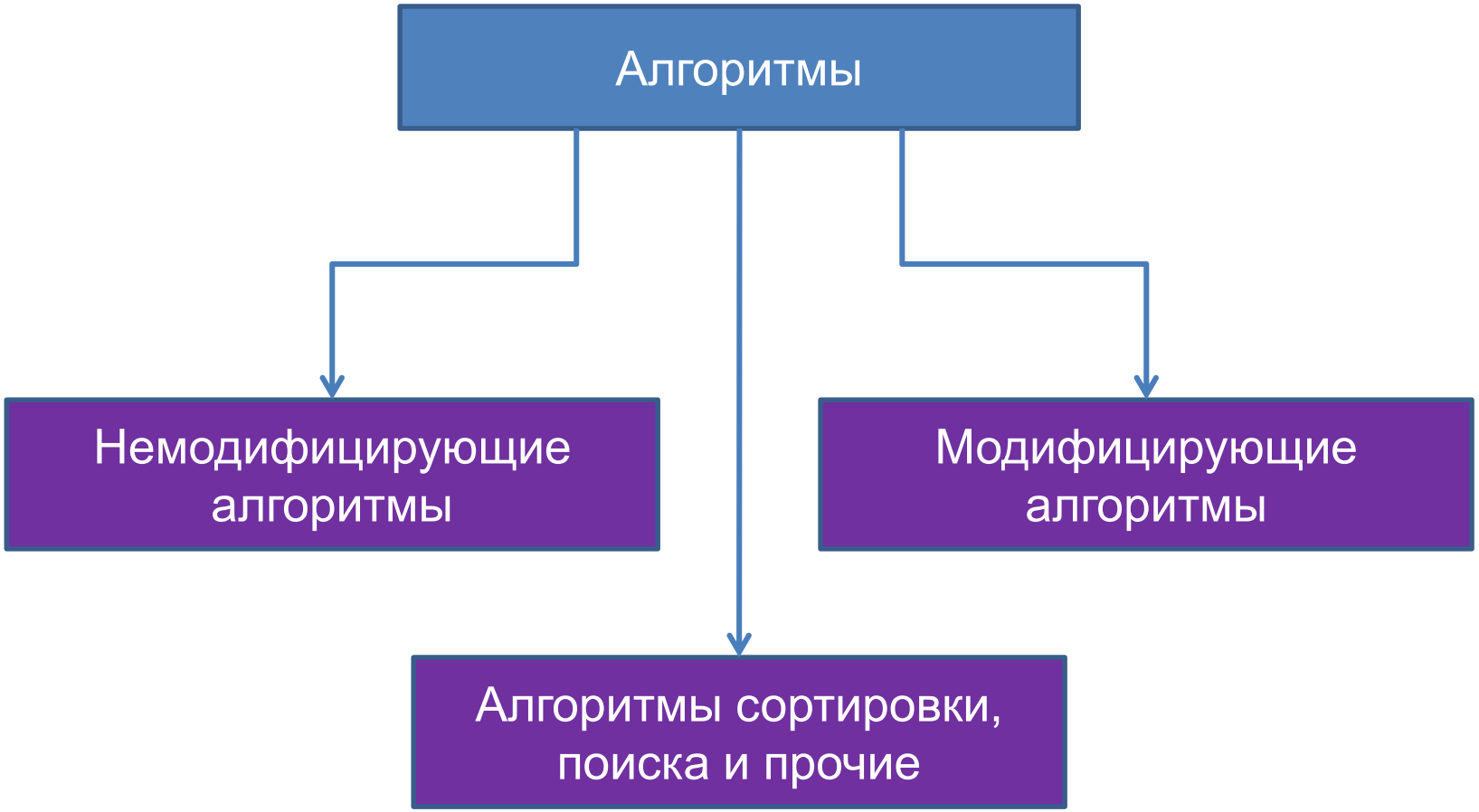
STL: Algorithms

Пример :

```
namespace std
{
   template< class Iter, class Func >
   Func for_each( Iter begin, Iter end, Func f );
}
```

```
#include <algorithm>

...
A_CONTAINER::iterator it1, it2;
A_FUNCTIONAL_TYPE f;
. . .
f = std::for_each( it1, it2, f );
```

STL: Algorithms



CQG

STL: Algorithms

Классификация: «in-place» и «copy» версии алгоритмов.

```
template < class ForwardIt, class T >
void replace( ForwardIt first, ForwardIt last,
              const T& v_old, const T& v_new );
```

Что делает: **`if ( *p == v_old ) *p = v_new;`**
последовательно для каждого **p** из **[first, last)**.

```
template<class InputIt, class OutputIt, class T>
OutputIt replace_copy( InputIt first, InputIt last,
                       OutputIt x,
                       const T& v_old, const T& v_new );
```

Что делает: **`if (*(first+i)==v_old) *(x+i) = v_new;`**
`else *(x+i) = *(first+i);`
последовательно для каждого **i** из **[0, last - first)**.

CQG

STL: Algorithms

Классификация: безусловные и «_if» версии алгоритмов.

```
template < class ForwardIt, class T >
void replace( ForwardIt first, ForwardIt last,
              const T& v_old, const T& v_new );
```

Что делает: **`if ( *p == v_old ) *p = v_new;`**
последовательно для каждого **p** из **[first, last)**.

```
template < class ForwardIt, class Pred, class T >
void replace_if( ForwardIt first, ForwardIt last,
                 Pred pred,
                 const T& val);
```

Что делает: **`if ( pred(*p) ) *p = val;`**
последовательно для каждого **p** из **[first, last).**

STL: Algorithms

Классификация: версии алгоритмов для всего контейнера или для его части.

```
template < class ForwardIt, class Gen >
void generate( ForwardIt first, ForwardIt last, Gen g );
```

Что делает: **`*p = g();`**
последовательно для каждого **p** из **[first, last)**.

```
template < class OutputIt, class Size, class Gen >
void generate_n( OutputIt first, Size n, Gen g );
```

Что делает: **`*(first + i) = g();`**
последовательно для каждого **i** из **[0, n).**

STL: Algorithms

Немодифицирующие алгоритмы:

```
for_each                (МОЖЕТ модифицировать!)

find
find_if
find_end                    (две версии)
find_first_of
adjacent_find   (две версии)

count
count_if

mismatch                    (две версии)
equal                       (две версии)

search          (две версии)
search_n                    (две версии)
```

STL: Algorithms

Модифицирующие алгоритмы:

copy
copy_backward
swap
swap_ranges
iter_swap
transform (two versions)
replace
replace_if
replace_copy
replace_copy_if
fill
fill_n
random_shuffle (two versions)
partition
stable_partition

generate
generate_n
remove
remove_if
remove_copy
remove_copy_if
unique (two versions)
unique_copy (two versions)

reverse
reverse_copy
rotate
rotate_copy

STL: Algorithms

Алгоритмы сортировки и прочие:

sort
stable_sort
partial_sort
partial_sort_copy
nth_element
lower_bound
upper_bound
equal_range
binary_search
merge
inplace_merge
includes
set_union
set_intersection
set_difference
set_simmetric_difference

push_heap
pop_heap
make_heap
sort_heap
min
max
min_element
max_element
lexicographical_compare
next_permutation
prev_permutation

CQG

STL: Algorithms

Пример: **for_each** с функциональным объектом «с состоянием»:

```
class Average
{
private:
   long m_count;
   long m_sum;

public:
   Average() : m_count(0), m_sum(0) {}

   void operator() (int elem)
   {
      m_sum += elem;
      ++m_count;
   }

   double Get()
   {
      return static_cast <double>(m_sum) /
             static_cast <double>(m_count);
   }
};
```

STL: Algorithms

Пример: for_each:

```
vector< int > v;

// здесь заполняем вектор...

Average res = for_each( v.begin(), v.end(), Average() );

cout << "The average of the elements of v is " << res.Get() << endl;
```

Реализация for_each:

```
template< class InIt, class Fn1 > inline
Fn1 for_each( InIt first, InIt last, Fn1 func)
{
   for( ; first != last; ++first )
   {
      func( *first );
   }
   return func;
}
```

STL: Algorithms

Адаптеры итераторов **(«итераторы вставки»)**:

```
vector<int> vec1;
vec1.push_back(10);
vec1.push_back(20);

vector<int> vec2;
```

Нужно скопировать вектор **vec1** в **vec2**…

```
template< class InputIterator, class OutputIterator >
OutputIterator copy(
   InputIterator src_begin, InputIterator src_end,
   OutputIterator dest_begin );
```

STL: Algorithms

Адаптеры итераторов **(«итераторы вставки»)**:

```
template< class InputIterator, class OutputIterator >
OutputIterator copy (
   InputIterator src_begin, InputIterator src_end,
   OutputIterator dest_begin )
{
   while( src_begin != src_end )
   {
      *dest_begin = *src_begin; //copy values
      ++src_begin; //increment iterators
      ++dest_begin;
   }
   return dest_begin;
}
```

STL: Algorithms

Адаптеры итераторов **(«итераторы вставки»)**:

```
vector<int> vec1;
vec1.push_back(10);
vec1.push_back(20);

vector<int> vec2;

copy( vec1.begin(), vec1.end(), vec2.begin() );  // ☹
```

STL: Algorithms

Адаптеры итераторов **(«итераторы вставки»)**:

```
vector<int> vec1;
vec1.push_back(10);
vec1.push_back(20);

vector<int> vec2;

// copy( vec1.begin(), vec1.end(), vec2.begin() );  // ☹

copy( vec1.begin(), vec1.end(), back_inserter(vec2) ); // ☺
```

STL: Algorithms

Адаптеры итераторов **(«итераторы вставки»)**:

Виды итераторов вставки			
Имя	Класс	Что вызывает	Как создать
Back inserter	back_insert_iterator	**push_back** (value)	**back_inserter** (cntr)
Front inserter	front_insert_iterator	**push_front** (value)	**front_inserter** (cntr)
General inserter	insert_iterator	**insert** (pos, value)	**inserter** (cntr, pos)

STL: Algorithms

Примеры:

```
vector<int> vec;

// заполнение...

// умножить все элементы в контейнере на -1

transform( vec.begin(), vec.end(), //source range
           vec.begin(),            //destination range
           negate<int>());         //operation
```

STL: Algorithms

Примеры:

```
vector<int> coll1;

// заполнение...

list<int> coll2;

// перенести элементы в list, умножив каждый на 10

transform( coll1.begin(), coll1.end(),      //source range
           back_inserter(coll2),            //destination
           bind2nd(multiplies<int>(),10)); //operation
```

STL: Algorithms

Примеры:

```
vector<int> vec;

// заполнение...

// вывести элементы на консоль

copy( vec.begin(), vec.end(),           //source range
   ostream_iterator<int>(cout," "));    //destination
```

STL: Algorithms

Примеры:

```
vector<int> coll1;

// заполнение...

// вывести на консоль в обратном порядке и с противоположным знаком

transform( coll1.rbegin(), coll1.rend(),   //source range
   ostream_iterator<int>(cout," "),        //destination
   negate<int>());                         //operation
```

STL: Removing elements

```
template < class ForwardIt, class T >
ForwardIt remove( ForwardIt first, ForwardIt last,
                  const T& value );

template < class ForwardIt, class UnaryPredicate >
ForwardIt remove_if( ForwardIt first, ForwardIt last,
 UnaryPredicate pred );
```

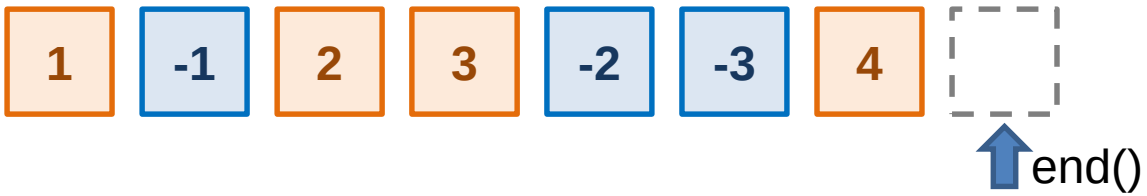
1. Обе версии алгоритма **ничего из контейнера не удаляют**.
   Причина проста: алгоритм ***НЕ УМЕЕТ*** удалять элементы из контейнера.

2. Алгоритм **перегруппировывает** элементы так, что элементы, не подлежащие удалению, перемещаются в начало. Относительный порядок этих, оставшихся, элементов не нарушается. Сохранность остальных («удаляемых») элементов вообще **не гарантируется**! Другими словами, не используйте этот алгоритм как способ разделить элементы на удовлетворяющие условию и не удовлетворяющие ему.

CQG

STL: Removing elements

```
vector<int> vec;

// fill it here

// удалить из вектора все элемены <= 0.
```

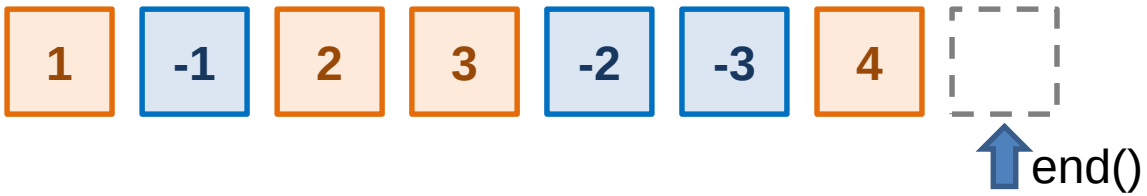


STL: Removing elements

```
vector<int> vec;

// fill it here

less_equal<int> comp;
binder2nd< less_equal<int> > non_positive( comp, 0);
```

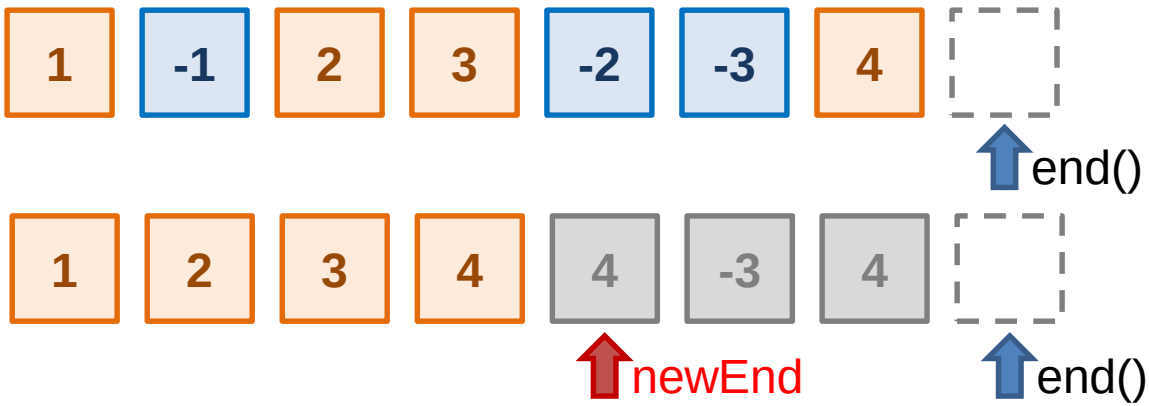
STL: Removing elements

```
vector<int> vec;

// fill it here

less_equal<int> comp;
binder2nd< less_equal<int> > non_positive( comp, 0);

vector<int>::iterator newEnd =
     remove_if( vec.begin(), vec.end(), non_positive );
```



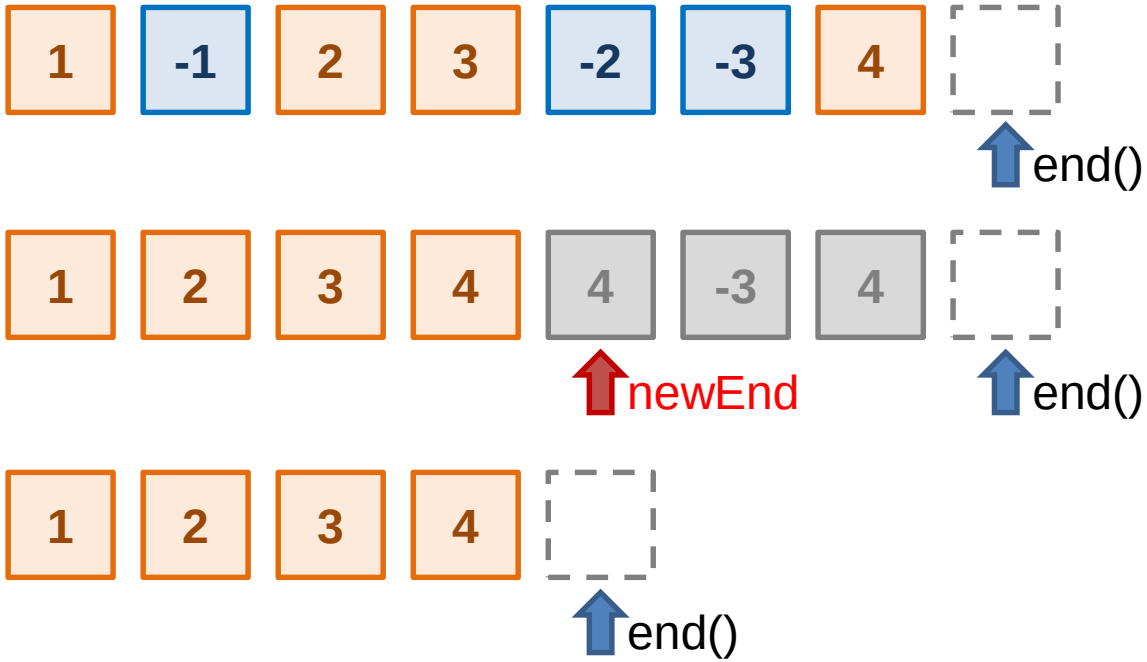
STL: Removing elements

```
vector<int> vec;

// fill it here

less_equal<int> comp;
binder2nd< less_equal<int> > non_positive( comp, 0);

vector<int>::iterator newEnd =
     remove_if( vec.begin(), vec.end(), non_positive );
```

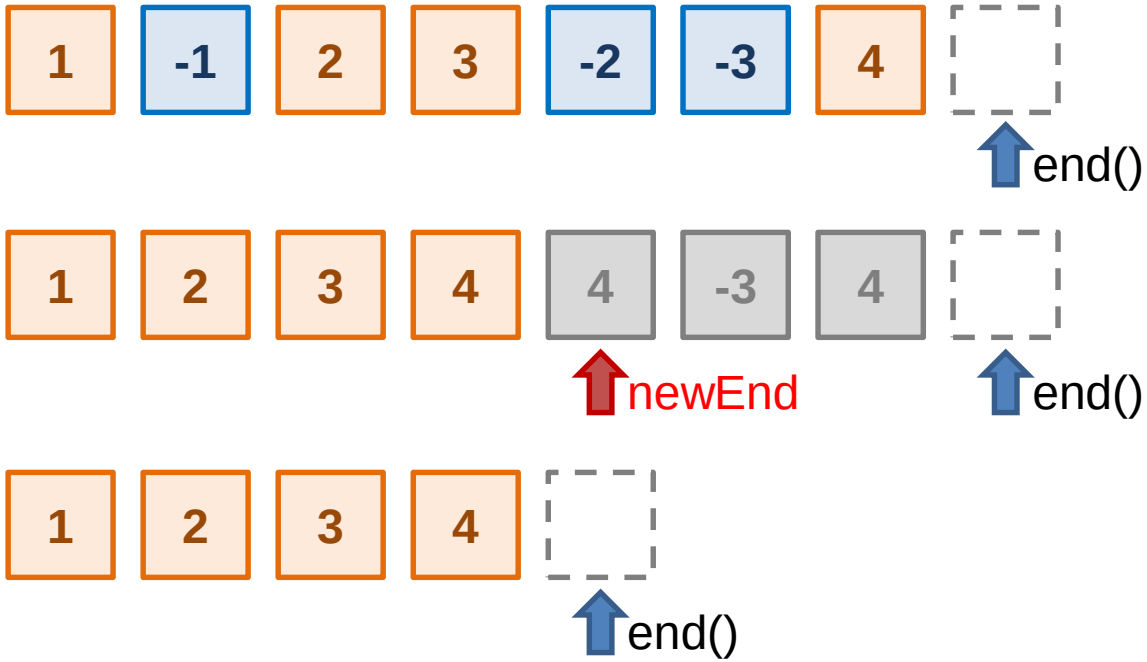


CQG

STL: Removing elements



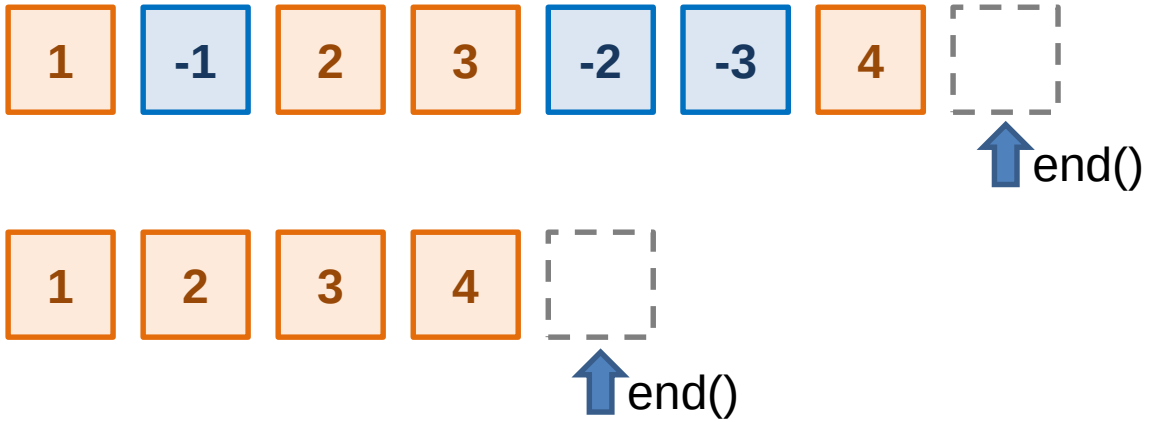
STL: Removing elements




```
vector<int>::iterator newEnd =
     remove_if( vec.begin(), vec.end(), non_positive );

vec.erase( newEnd, vec.end() );
```

CQG

STL: Removing elements



Идиома «Remove - Erase» :

```
cn.erase(
    remove_if( cn.begin(), cn.end(), predicate),
    cn.end() );
```

STL: Removing elements

Примечания:

С *ассоциативными контейнерами* так не получится – мешает копирование, используемое внутри алгоритма. Пользуемся их версиями функции **erase().**

У std::**list** есть свои функции **remove()** и **remove_if()**, которые оптимизированы по быстродействию и физически удаляют элементы:

```
template<class Predicate>
void remove_if( Predicate pred );
```

Приложение

Помимо собственноручно нарисованных, использованы картинки со следующих интернет-ресурсов:

http://scottmeyers.blogspot.ru/2015/09/should-you-be-using-something-instead.html

http://codeforces.com/blog/entry/4710

https://www.slideshare.net/gvivek1/c-advanced

http://conglang.github.io/2015/01/01/stl-unordered-container/