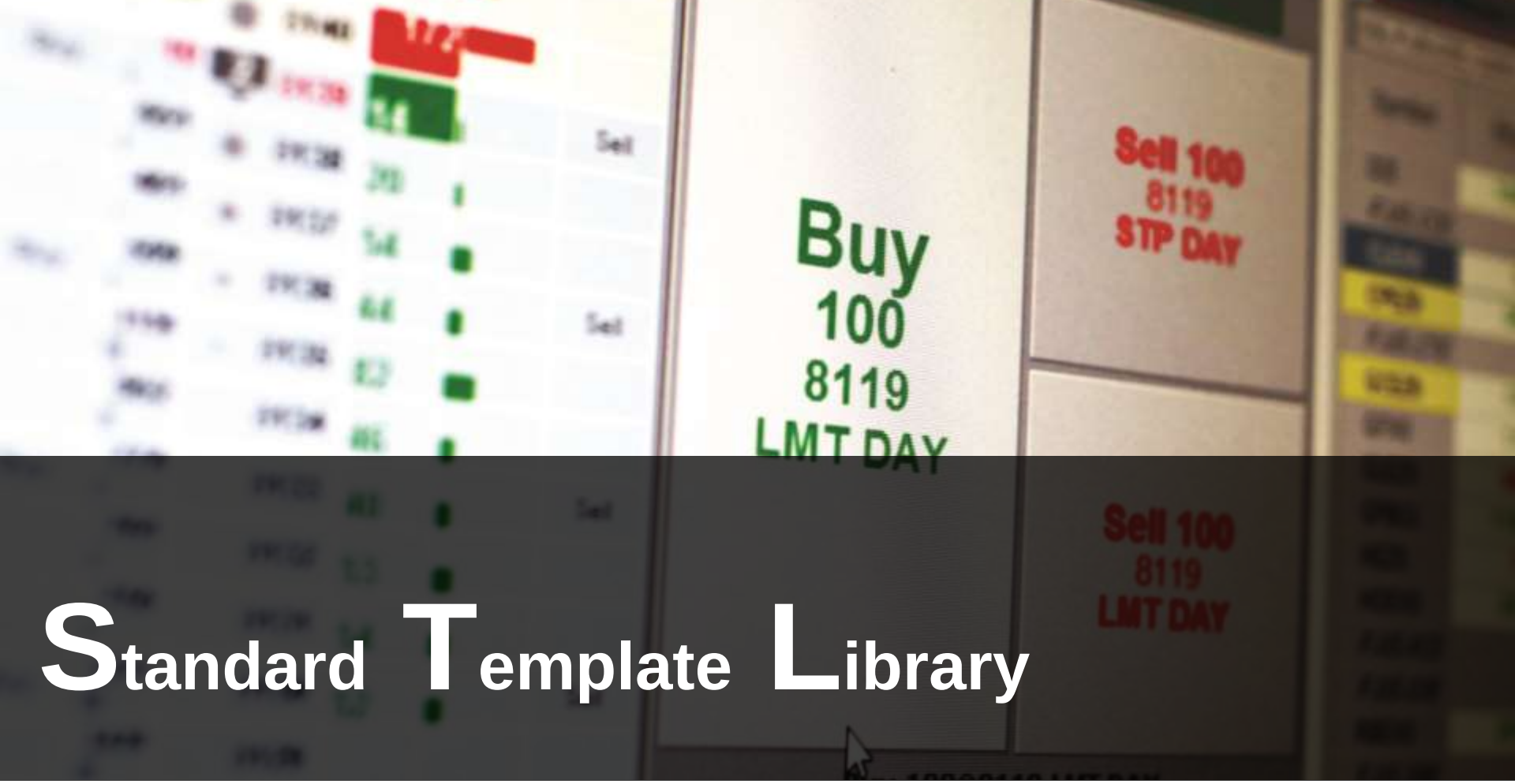


Standard Template Library

Контейнеры. Итераторы. Алгоритмы.

2022

CQG

О чем эта лекция?

«У STL не существует официального определения, и разные авторы вкладывают в этот термин разный смысл...
...Термин STL относится к **компонентам Стандартной библиотеки C++, работающим с итераторами»**

Скотт Мейерс, «Эффективное использование STL»

«Стандарт языка не называет её «STL», так как эта библиотека стала неотъемлемой частью языка, однако многие люди до сих пор используют это название, чтобы отличать её от остальных частей Стандартной Библиотеки (таких, как потоки ввода-вывода (*iostream*), подраздел *Си* и др.)»

Wikipedia

STL: Components

Стандартная библиотека шаблонов (*Standard Template Library*):

1. **Набор** согласованно работающих вместе обобщённых (шаблонных) ***компонентов*** C++.

2. **Набор *соглашений*** (требований, предъявляемых к интерфейсу компонентов), позволящих ***расширять*** этот набор.

STL: Components

Контейнеры
(структуры хранения данных)

Алгоритмы
(методы работы с данными)

Ортогональная архитектура:

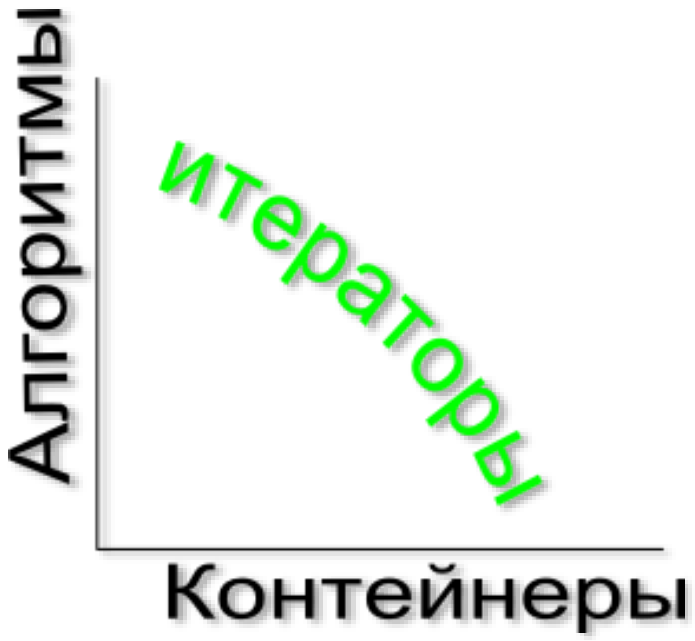
Контейнеры **не знают** об алгоритмах.
Алгоритмы **не знают** о контейнерах.

STL: Components

Контейнеры
(структуры хранения данных)

Алгоритмы
(методы работы с данными)

Итераторы
(с их помощью контейнеры предоставляют алгоритмам доступ к своим данным)



STL: Iterators

* Итератор – это ***тип.***

* Любой объект может играть роль итератора, если он удовлетворяет ***определенному набору условий*** .

* Итераторы можно рассматривать как ***обобщение указателей*** .

* Итераторы являются ***интерфейсом взаимодействия между контейнером и алгоритмом***, который им манипулирует .

CQG

STL: Intro to Concept of Iterator

Пример: Напишем функцию, которая ищет первый элемент массива, равный заданному числу:

```
int* find0(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
}
```

Каковы ограничения, которые накладывает наша реализация?

STL: Intro to Concept of Iterator

```
int* find0(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
}
```

1 3 4 2

Ограничения алгоритма:

1. Он ищет int
2. Он сканирует массив int-ов
3. Мы должны передать указатель на первый элемент
4. Мы должны передать число элементов в массиве

Как ослабить ограничения?

CQG

STL: Intro to Concept of Iterator

Шаг 1: Обобщим ТИП элементов массива:

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

Функция стала более общей, однако мы неявно требуем, чтобы тип T имел **`operator==()`**

STL: Intro to Concept of Iterator

Основной недостаток нашего алгоритма (с точки зрения общности) – мы завязаны на специфическую структуру хранения данных: С-массив.

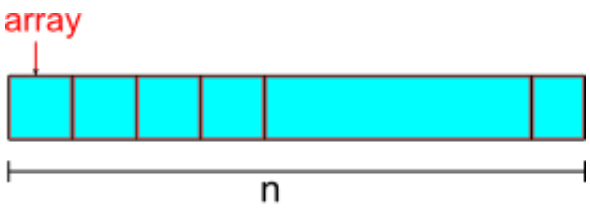
1. Мы знаем **адрес первого элемента** массива.
2. Мы используем **информацию о количестве элементов** массива для выхода из цикла.

Попытаемся избавиться от этих ограничений.

STL: Intro to Concept of Iterator

Шаг 2: Избавимся от размера массива:

```
template< typename T >
T* find2(T* array, T* beyond, const T& x)
{
    T* p = array;
    while (p != beyond)
    {
        if (*p == x) return p; // success
        ++p;
    }
    return beyond; // fail
}
```



STL: Intro to Concept of Iterator

Шаг 3: Упростим и немного оптимизируем алгоритм:

```
template< typename T >
T* find3(T* first, T* beyond, const T& x)
{
    T* p = first;
    while (p != beyond && *p != x)
    {
        ++p;
    }
    return p; // результат
}
```

Изменения:

1. Мы объединили две проверки вместе, инвертировав условие.
2. Мы убрали один **return**, функция стала короче.
3. Мы переименовали **array** в **first**

STL: Intro to Concept of Iterator

Шаг 4, последний: Откажемся от указателей совсем!

```
template< typename T, typename P >
P find4(P first, P beyond, const T& x)
{
    P p = first;
    while (p != beyond && *p != x)
    {
        ++p;
    }
    return p; // результат
}
```


CQG

STL: Intro to Concept of Iterator

```
template< typename T, typename P >
P find4(P first, P beyond, const T& x)
{
    P p = first;
    while (p != beyond && *p != x)
    {
        ++p;
    }
    return p; // результат
}
```

- Алгоритм **find4** осуществляет поиск в структуре с данными типа **T**
- Доступ к данным осуществляется с помощью объектов типа **P**

Требования/Предположения:

1. Тип **T** должен поддерживать **operator!=()**
2. Тип **P** должен поддерживать **operator*()**, возвращающий значение типа **T**.
3. Тип **P** должен поддерживать **operator++()**, возвращающий значение типа **P**, ассоциированное со *следующим элементом в структуре*.
4. Тип **P** должен поддерживать **operator!=()**

STL: Intro to Concept of Iterator

```
template< typename T, typename P >
P find4(P first, P beyond, const T& x)
{
   P p = first;
   while (p != beyond && *p != x)
   {
      ++p;
   }
   return p; // result
}
```

Использование с С-массивами:

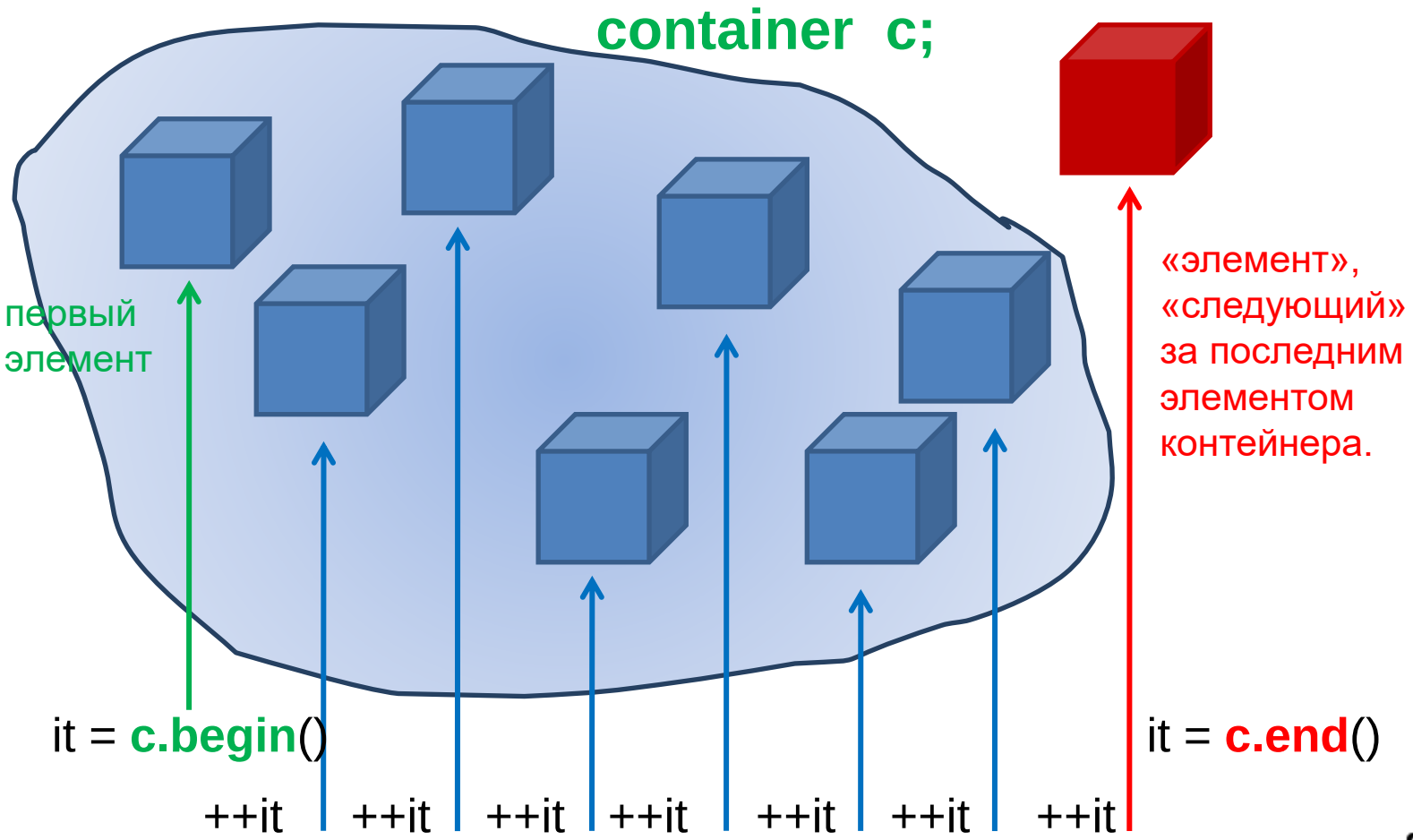
```
int A[100];
// initialization...

int* pResult = find4(A, &A[100], 5);
```

Явное инстанцирование:

T <=> int

P <=> int*

STL: Iterators



STL: Iterators

```
using Collection = std::vector<int>;

Collection data;

// filling the vector somewhere here…

Collection::iterator it =  data.begin();              // auto it =  data.begin();
const Collection::iterator itEnd =  data.end();  // const auto itEnd =  data.end();

for (; it != itEnd;  ++it)
{
      std::cout << *it << std::endl;
}
```


CQG

STL: Iterators

```
using Collection = std::vector<int>;

Collection data;

// filling the vector somewhere here…

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

STL: Iterators

```
using Collection = std::vector<int>;

Collection data;

// filling the vector somewhere here…

for (const auto& element : data)
{
	std::cout << element << std::endl;
}
```

STL: Iterators

```
using Collection = std::vector<int>;

Collection data;

// filling the vector somewhere here…

const auto itFound = std::find(data.begin(), data.end(), 4 );
// note: we pass iterators, NOT the container itself!

if (itFound != data.end())
{
	std::cout << *itFound << std::endl;
}
```

STL: Iterators

```
using Collection = std::vector<int>;

Collection data;

// filling the vector somewhere here…

const auto sum = std::accumulate(data.begin(), data.end(), 0);
```

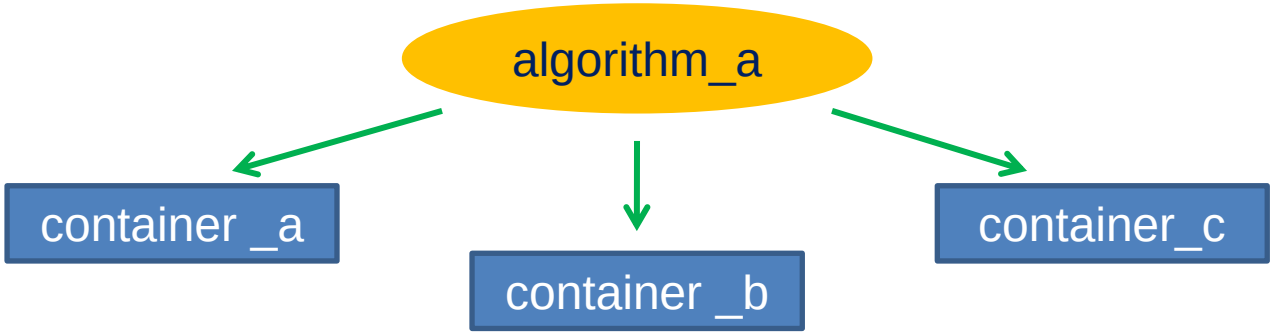
STL: Iterators

```
using Collection = std::vector<int>;

Collection data;

// filling the vector somewhere here…

auto sum = std::accumulate(data.begin(), data.end(), 0);

sum = std::accumulate(data.begin(), data.find(5), 0);
```

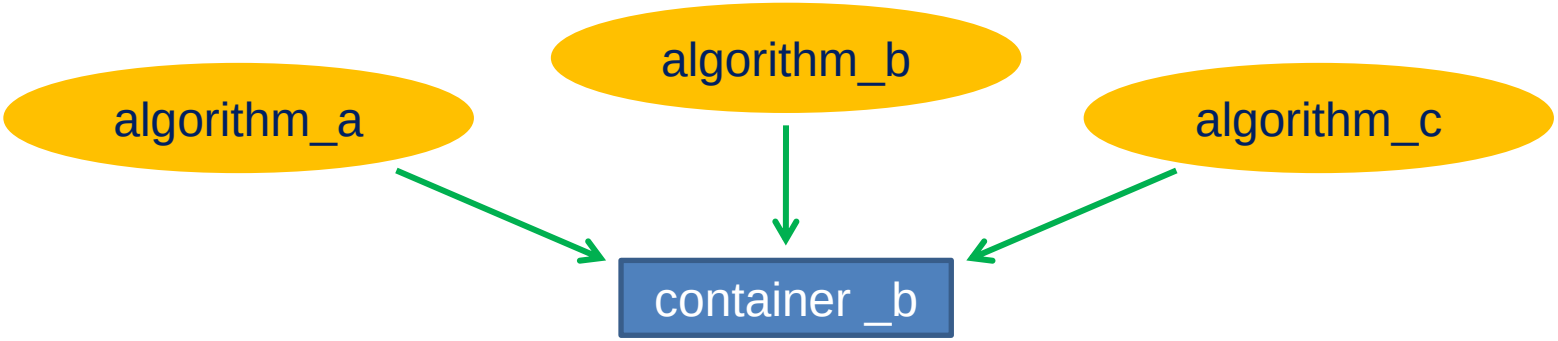
STL: Iterators

Категория	Поддерживаемые операции
Выходные	operators: **++, *,** copy ctor
Входные	operators: **++, *, ->, =, ==, !=** copy ctor
Однонаправленные	operators: **++, *, ->, =, ==, !=** copy ctor, default ctor
Двунаправленные	operators: **++, --, *, ->, =, ==, !=** copy ctor, default ctor
Произвольного доступа	operators: **++, --, *, ->, =, ==,!=,** **+, -, +=, -=, <, >, <=, >=, []** copy ctor, default ctor

STL: Гибкость



мы можем применить один и тот же **алгоритм** почти ко всем **контейнерам**



мы можем применить разные **алгоритмы** к одному и тому же **контейнеру**

STL: Гибкость

- **Существующие компоненты STL ортогональны**:
  - ✓ мы можем применить **один и тот же алгоритм** к **разным контейнерам**
  - ✓ мы можем применить **разные алгоритмы** к **одному и тому же контейнеру**

- **STL расширяема «по обеим осям»** :
  - ✓ Программист может определить **свои контейнеры**. Библиотечные алгоритмы будут работать и с ними (при соблюдении соглашений)
  - ✓ Программист может определить **свои алгоритмы**. Они будут работать с библиотечными контейнерами (при соблюдении соглашений)

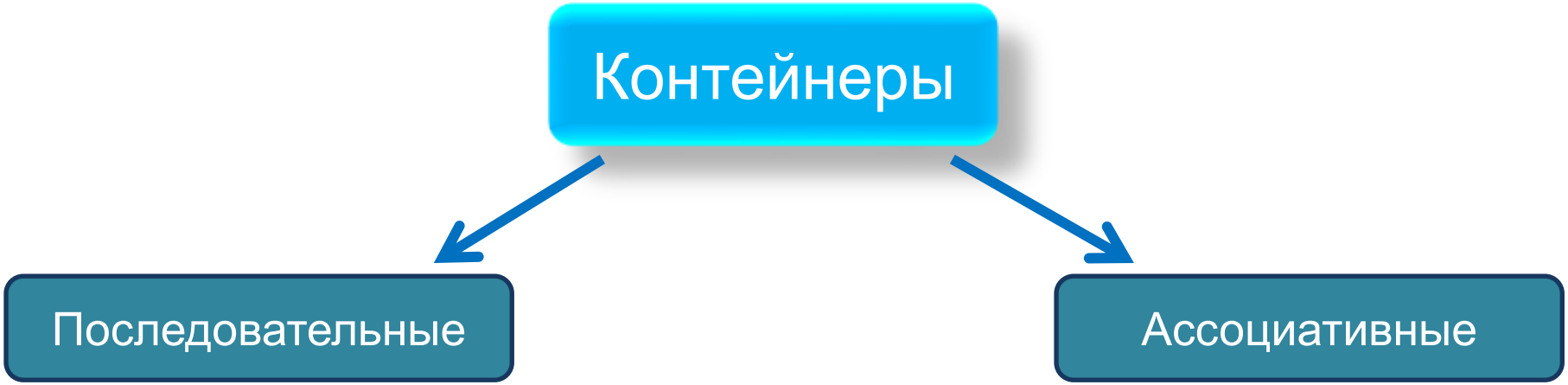
STL: История

Александр Степанов

- **Александр Степанов** родился в 1950 в Москве. Учился в Московском государственном университете.
- В 1977 Александр уехал в США и начал работать в *General Electric Research Center*. Получил грант для работы над реализацией своих идей обобщённого программирования в виде библиотеки алгоритмов на языке Ada.
- В 1987 получил предложение поработать в *Bell Laboratories*, чтобы реализовать свой подход в виде библиотеки на языке C++. Однако стандарт языка в это время ещё не позволял в полном объёме осуществить задуманное.
- В 1992 он вернулся к работе над алгоритмами. В конце 1993 он рассказал о своих идеях Эндрю Кёнигу, который, высоко оценив их, организовал ему встречу с членами Комитета ANSI/ISO по стандарту C++.
- Весной **1994** библиотека STL, разработанная Александром Степановым при помощи Менг Ли стала частью официального стандарта языка C++

CQG

STL: Containers



CQG

STL: Containers

Последовательные

ОСНОВНЫЕ:

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

ИХ АДАПТЕРЫ:

stack
queue
priority_queue

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

Последовательные

vector
(расширяемый массив)

vector:

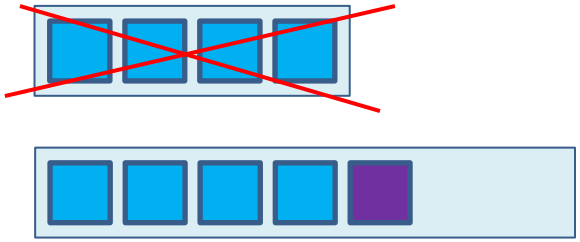
- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- возможность управления выделением памяти
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные

vector
(расширяемый массив)



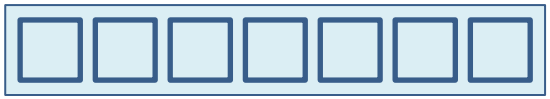
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
```

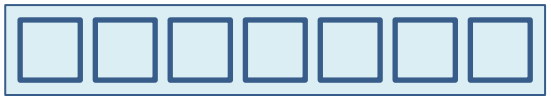
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



capacity() == 7
size() == 0

vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
```

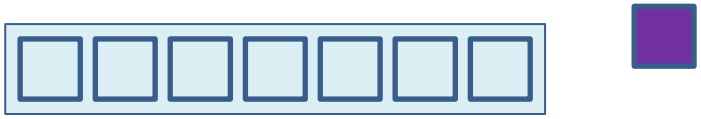
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
data.push_back(item1);
```

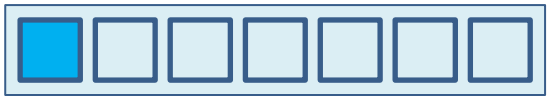
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



capacity() == 7
size() == **1**

vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
data.push_back(item1);
```

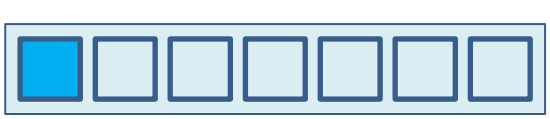
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
data.push_back(item1);
data.push_back(item2);
```

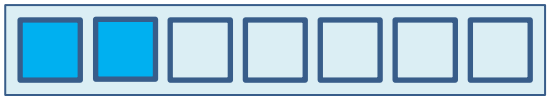
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные



capacity() == 7
size() == **2**

vector
(расширяемый массив)

```
std::vector< int > data;
data.reserve(7);
data.push_back(item1);
data.push_back(item2);
```

vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- **возможность управления выделением памяти**
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные

vector
(расширяемый массив)

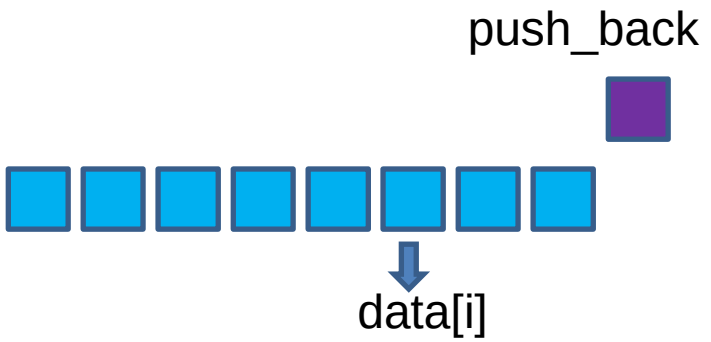
```
#include <vector>

using Collection = std::vector<int>;

Collection vec;

vec.push_back( 8 );      /// [8]

vec.reserve( 10 );       /// [8xxxxxxxxx]

vec.push_back( 2 );      /// [82xxxxxxxx]

vec.resize( 4, 7 );      /// [8277xxxxxx]

int a = vec[5];
```


CQG

STL: Containers

Последовательные

vector (расширяемый массив)



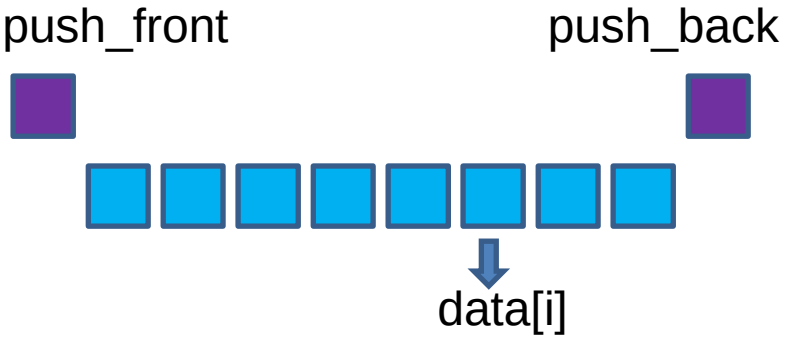
vector:

- быстрое добавление в **конец** (push_back)
- быстрый доступ к **произвольному элементу [i]**
- возможность управления выделением памяти
- гарантии последовательного размещения в памяти: совместимость.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

STL: Containers

Последовательные

vector (расширяемый массив)
deque (двусторонняя «очередь»)



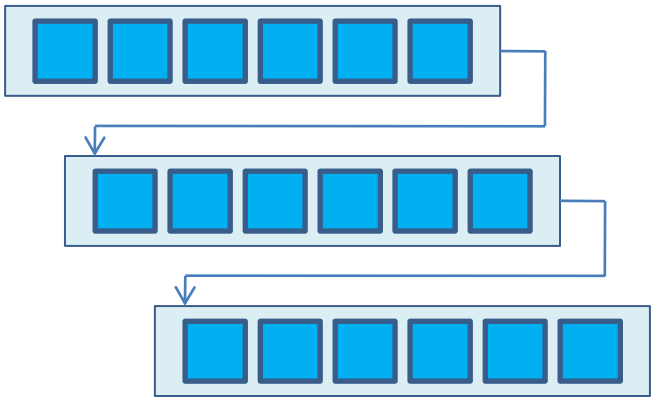
deque:

- быстрое добавление **в конец** **и в начало** (push_front , pop_front)
- быстрый доступ к произвольному элементу **[i]** (но медленнее вектора)
- **нет возможности управлять выделением памяти**
- **несовместимость** с С-массивами.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

STL: Containers

Последовательные

vector (расширяемый массив)
deque (двусторонняя «очередь»)



deque:

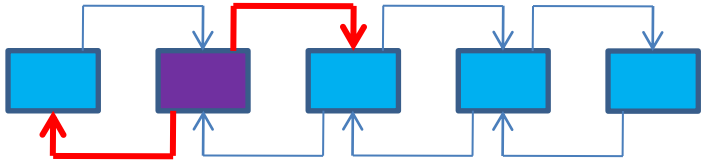
- быстрое добавление **в конец и в начало** (push_front , pop_front)
- быстрый доступ к произвольному элементу **[i]** (но медленнее вектора)
- **нет возможности управлять выделением памяти**
- **несовместимость** с С-массивами.
- медленная вставка в произвольной позиции
- итераторы могут стать невалидны при любом добавлении-удалении

CQG

STL: Containers

Последовательные

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



list:

- **быстрое добавление в любую позицию** (insert, push_back)
- **нет доступа (!)** к произвольному элементу **[i]**
- нет возможности управлять выделением памяти
- несовместимость с C-массивами.
- **итераторы остаются валидны** при любом добавлении-удалении (кроме итератора на сам удаляемый элемент)
- поддерживает ряд оптимизированных методов (аналогов алгоритмов)

CQG

STL: Containers

Последовательные

ОСНОВНЫЕ:

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

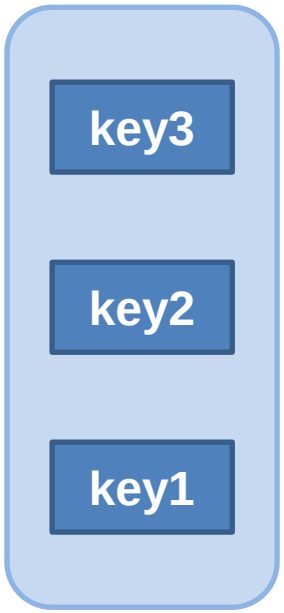
Ассоциативные

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

STL: Containers

Ассоциативные

set (отсортированное множество)



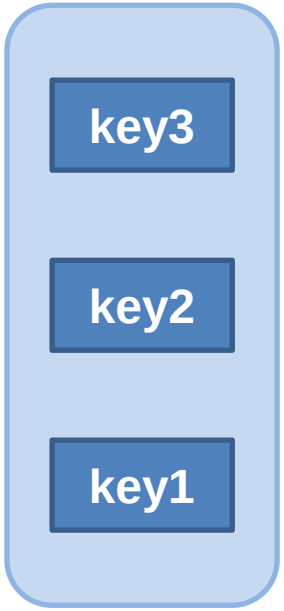

```
#include <set>

using Collection = std::set<int>;

Collection s;

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

for (const auto& element : s)
{
    std::cout << element << " ";
}
```

```
-1 3 8
```

STL: Containers

Ассоциативные

set (отсортированное множество)



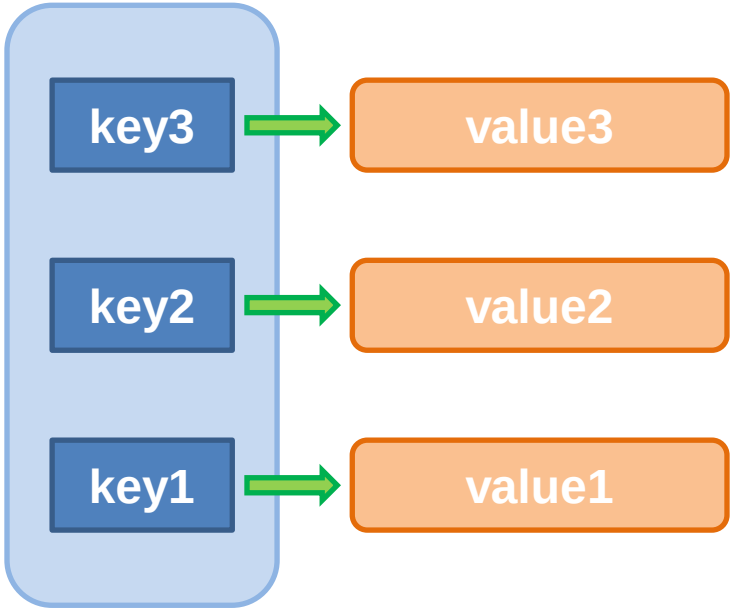

```
#include <set>

using Collection = std::set<int>;

Collection s;

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

const auto it = s.find(4);
if (it != s.end())
{
	std::cout << *it << std::endl;
}
```


CQG

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;
```


CQG

STL: Containers

Ассоциативные

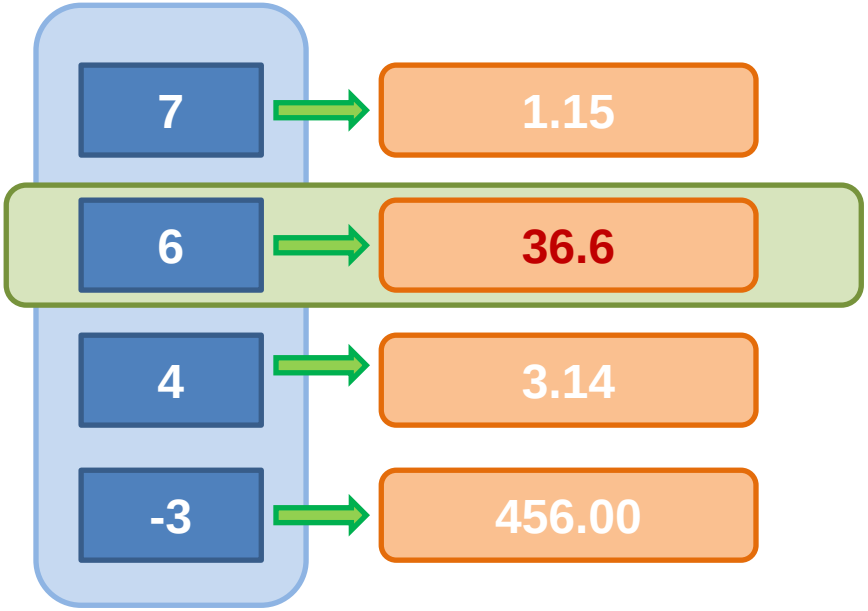
set (отсортированное множество)
map (словарь: ключ -> значение)


```
#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)



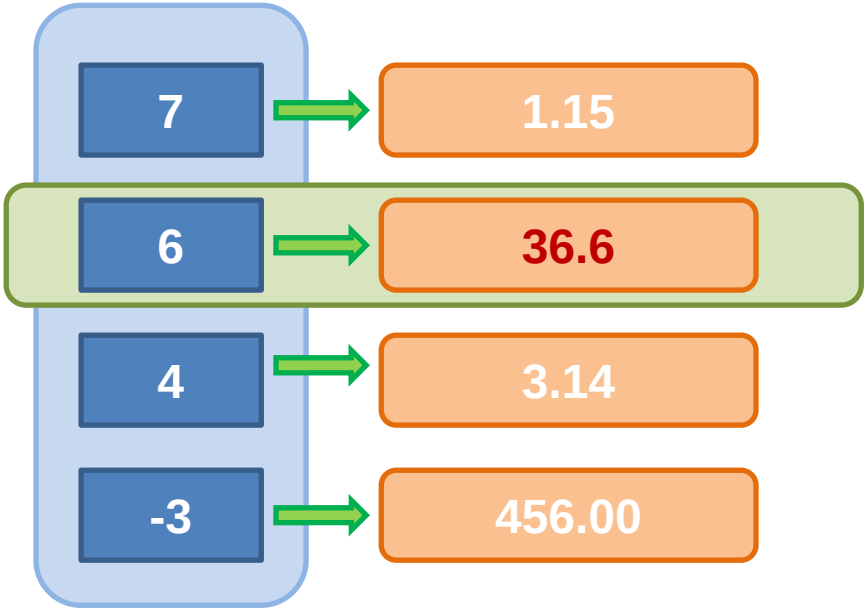

```
#include <map>

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

Collection m;

m.insert( std::pair<int, double>(6, 36.6) );

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

// or
m.emplace( 6, 36.6 );

// or (since C++17)
m.try_emplace( 6, 36.6 );
```


CQG

STL: Containers

Ассоциативные

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



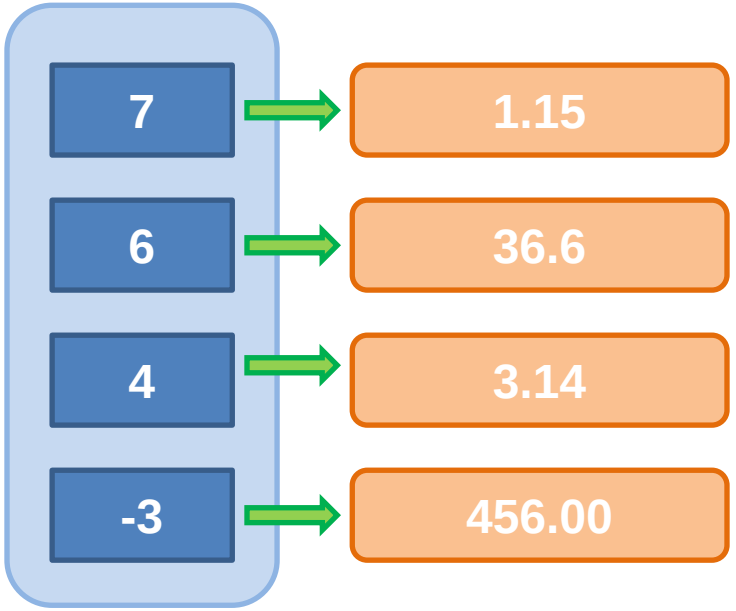

```
#include <map>

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

Collection m;

m.insert( std::pair<int, double>(6, 36.6) );

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

  // other stuff
};
```


CQG

STL: Containers

Ассоциативные

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

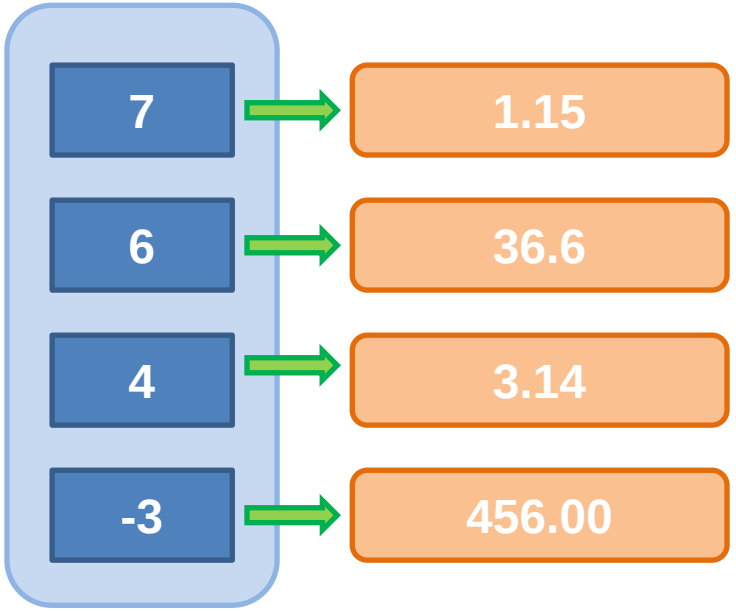



```
#include <map>

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

Collection m;

m.emplace(6, 36.6);

m.emplace(6, 37.1);
```

STL: Containers

Ассоциативные

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

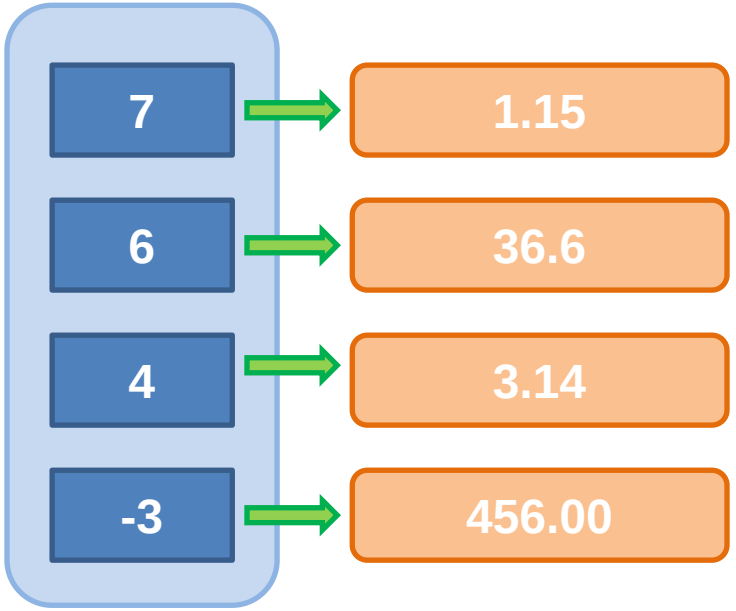



```
#include <map>

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

Collection m;

m.emplace(6, 36.6);

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

bool isInsertedOk = res.second;
```

STL: Containers

Ассоциативные

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




```
#include <map>

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

Collection m;

m.emplace(6, 36.6);

const auto it =  m.find(4);

if (it != m.end())
{
    std::cout << it->first << " " <<
                 it->second << std::endl;
}
```

STL: Containers

Последовательные

ОСНОВНЫЕ:

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

Ассоциативные

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

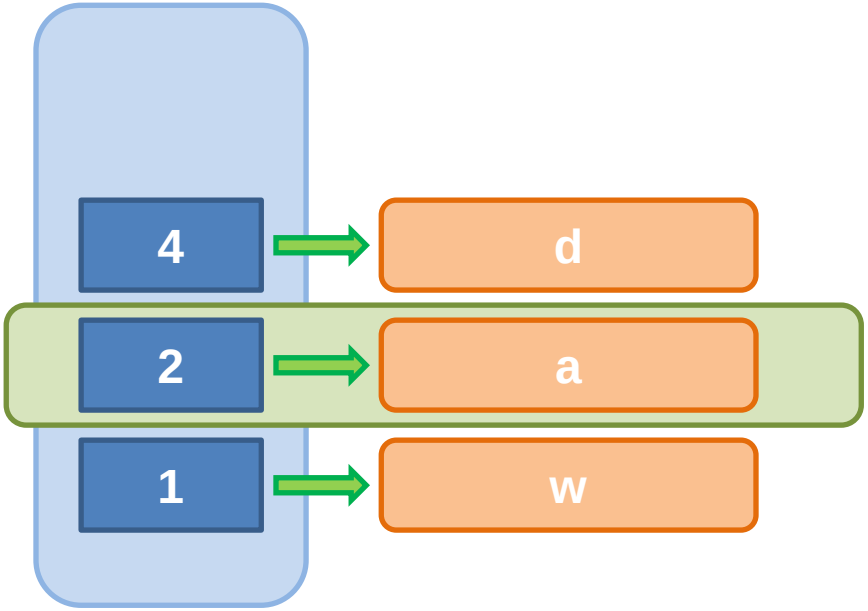
multiset (множество с дубликатами)
multimap (словарь с дубликатами)

CQG

STL: Containers

Ассоциативные

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

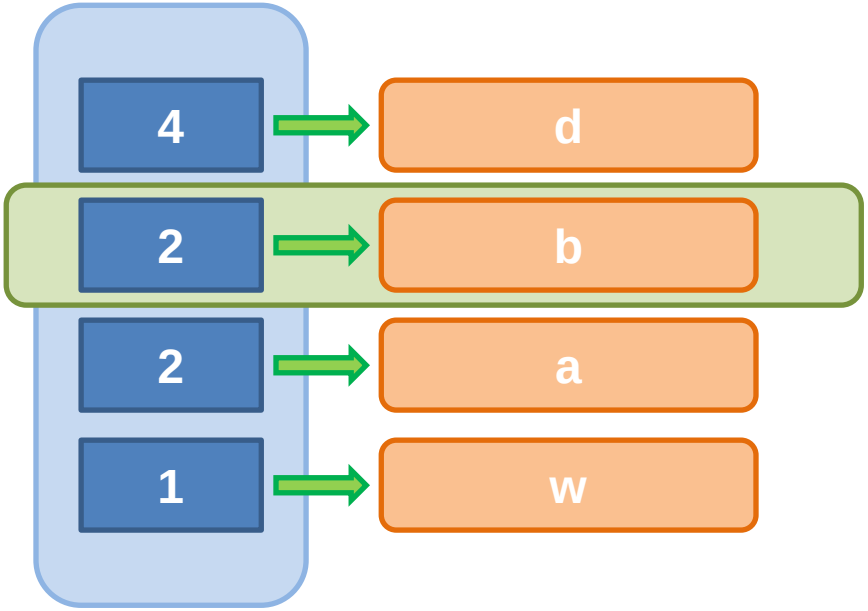



```
#include <map>

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

Collection m;

m.emplace(2, ‘a’);
```

STL: Containers

Ассоциативные

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



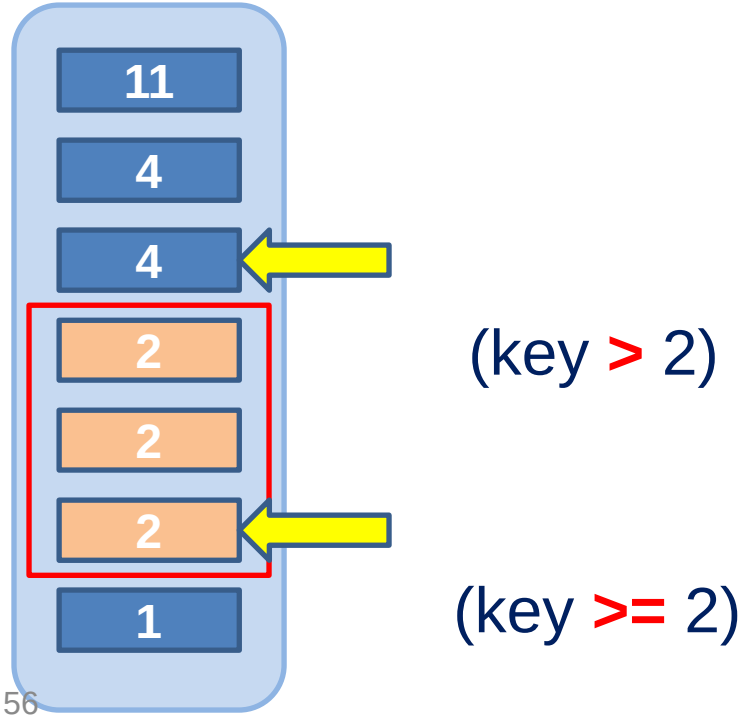

```
#include <map>

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

Collection m;

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

// увы, больше нет m[2] = ‘b’;
```


CQG

STL: Containers

Ассоциативные

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

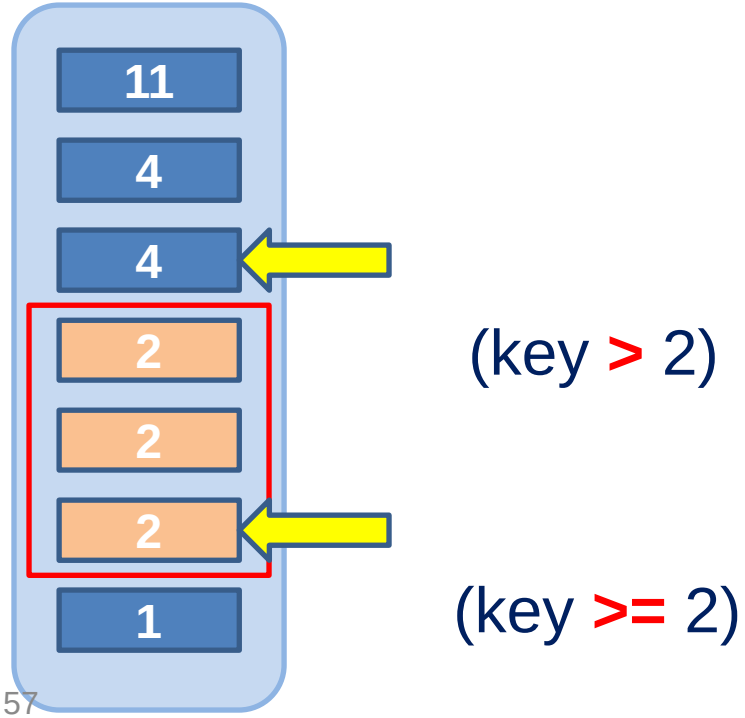



```
#include <map>

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

Collection m;

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

STL: Containers

Ассоциативные

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

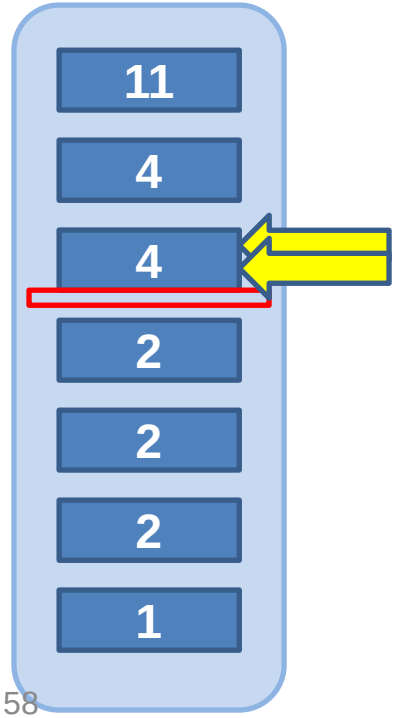



```
#include <map>

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

Collection m;

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

// auto 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>;

Collection m;

const auto 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)

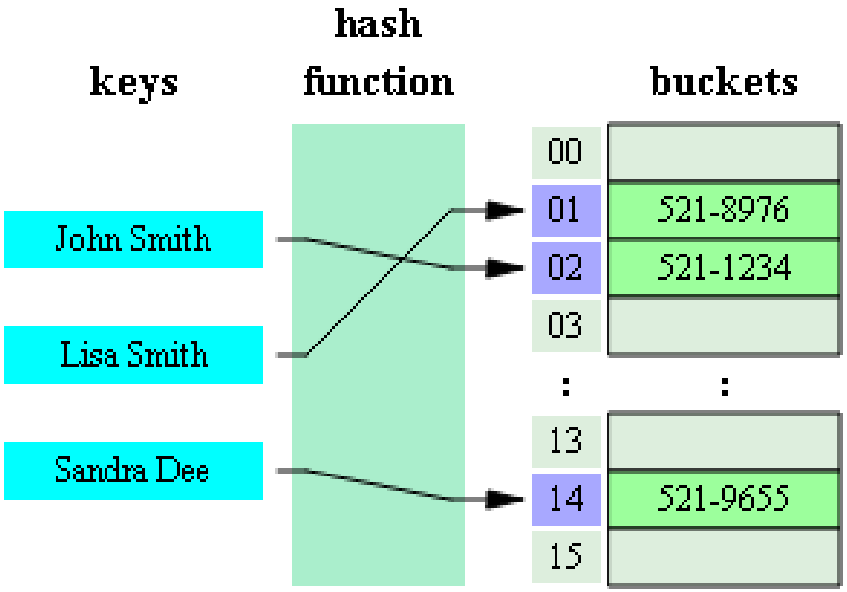
unordered_set (hashed)
unordered_map (hashed)

unordered_multiset (hashed)
unordered_multimap (hashed)

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»).

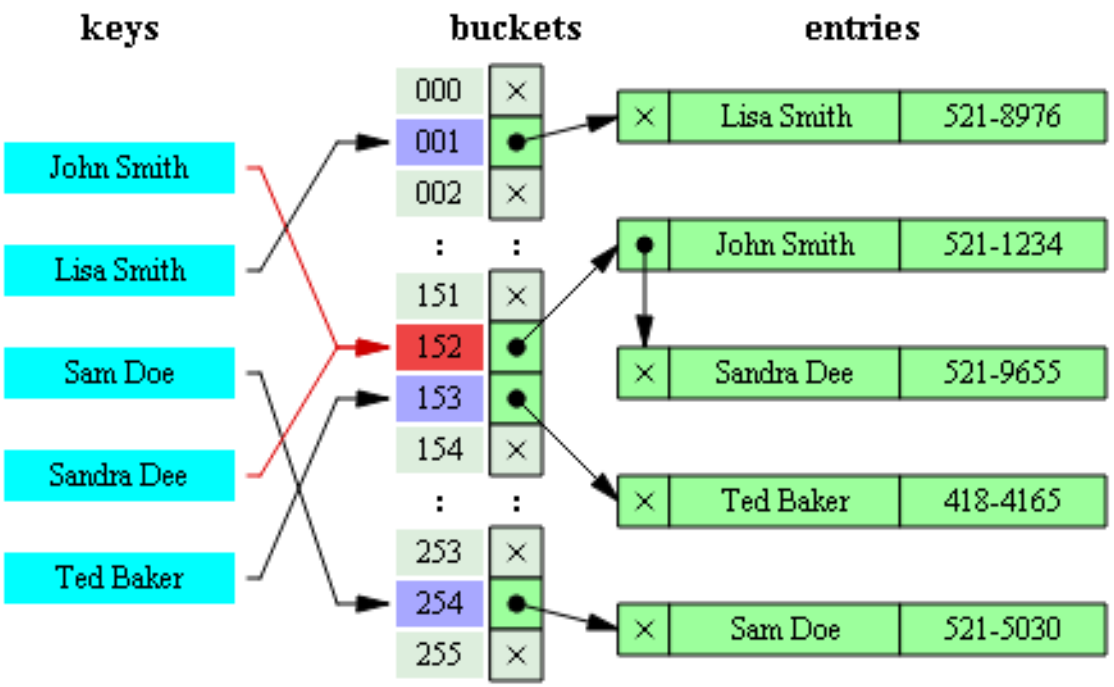
<уныние>И что теперь? Выкидывать хэш-таблицу на свалку? </уныние>

STL: Containers

Хэш-таблица: коллизии

Два метода разрешения коллизий:

“Chaining”

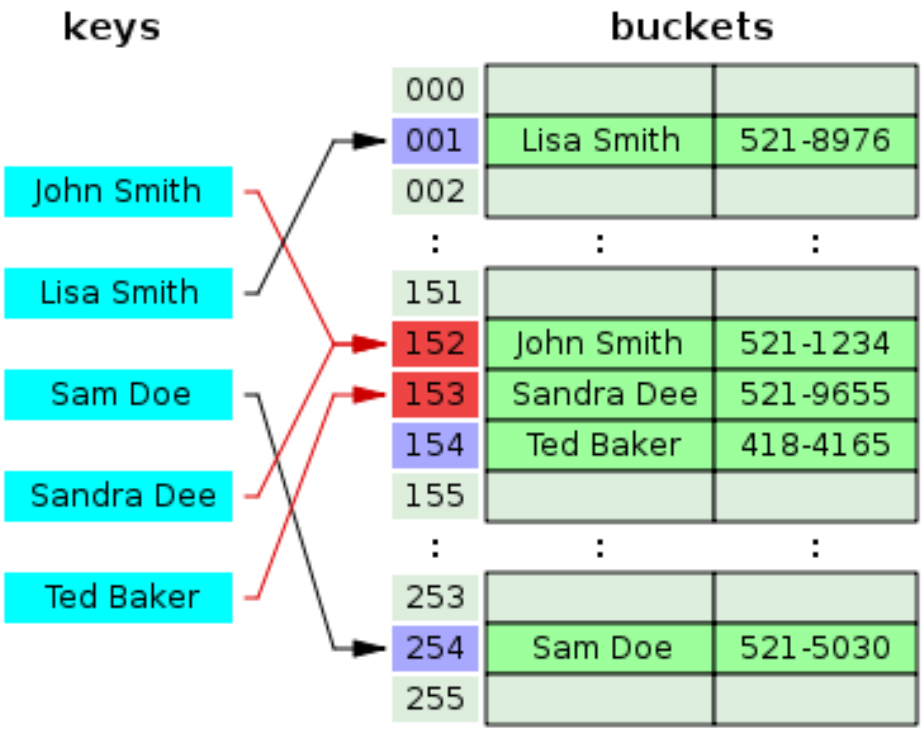


STL: Containers

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;
```

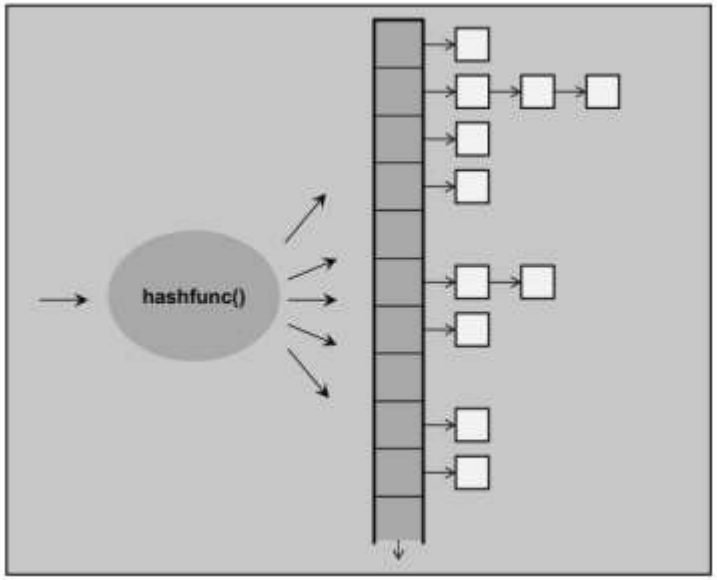
```
namespace std
{
template<>
class hash<Foo>
{
public:
    size_t operator()(const Foo& obj) const
    {
        size_t h1 = std::hash<string>()(obj.first_name);
        size_t h2 = std::hash<string>()(obj.last_name);
        return h1 ^ ( h2 << 1 );
    }
};
}
```


CQG

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;
```



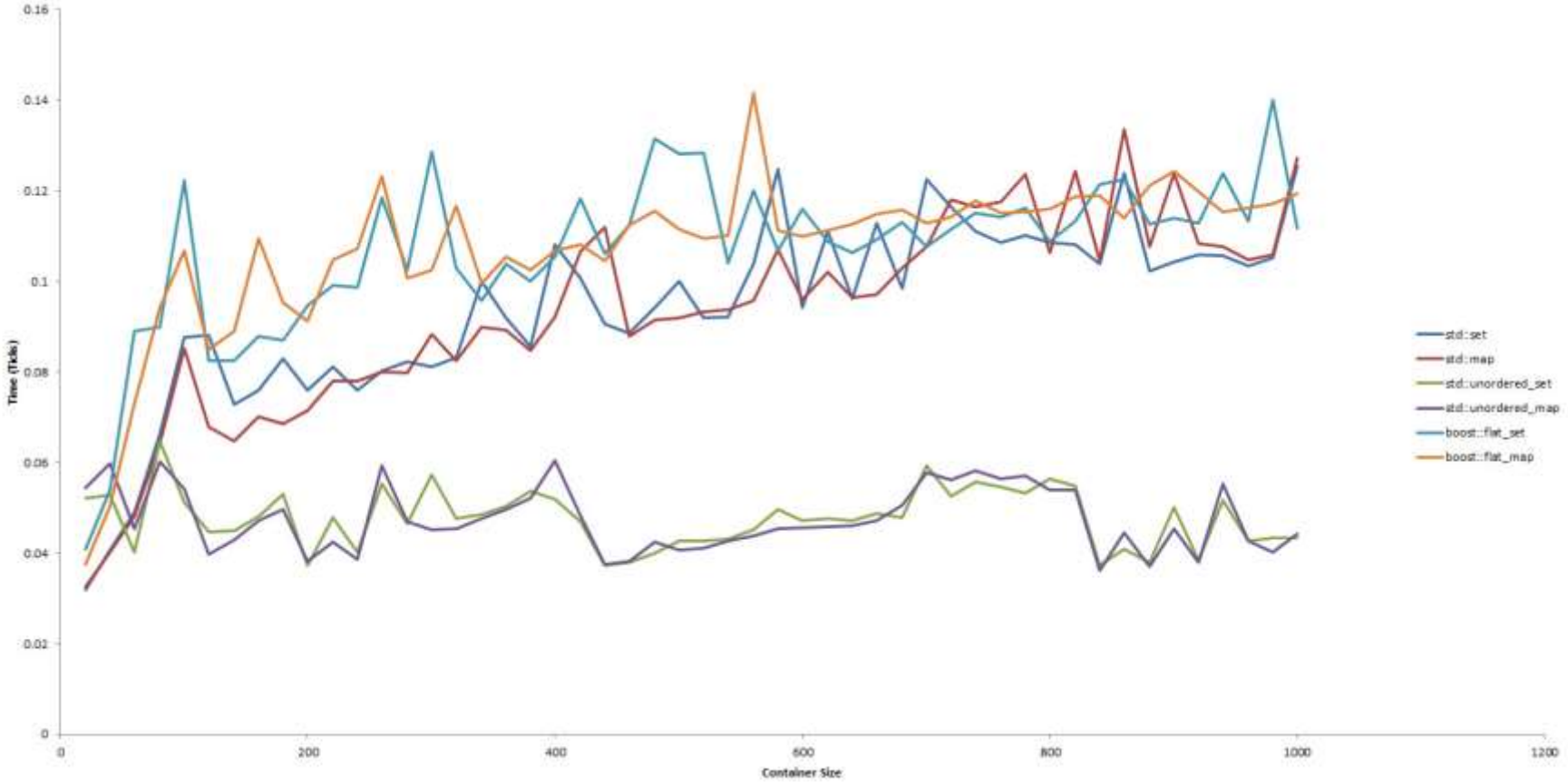
STL: Containers

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;
```

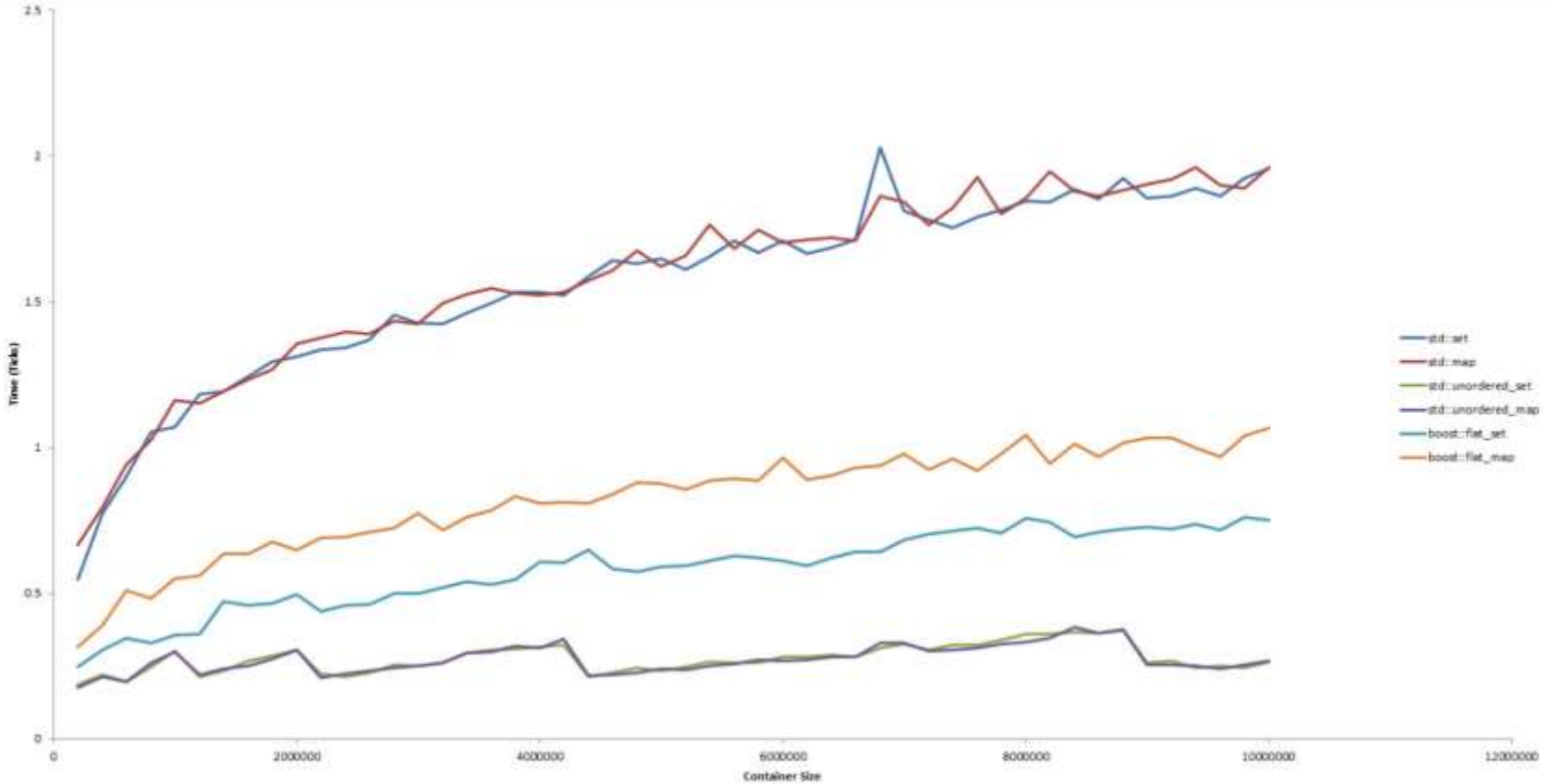
STL: Containers

unordered_set/unordered_map: performance



STL: Containers

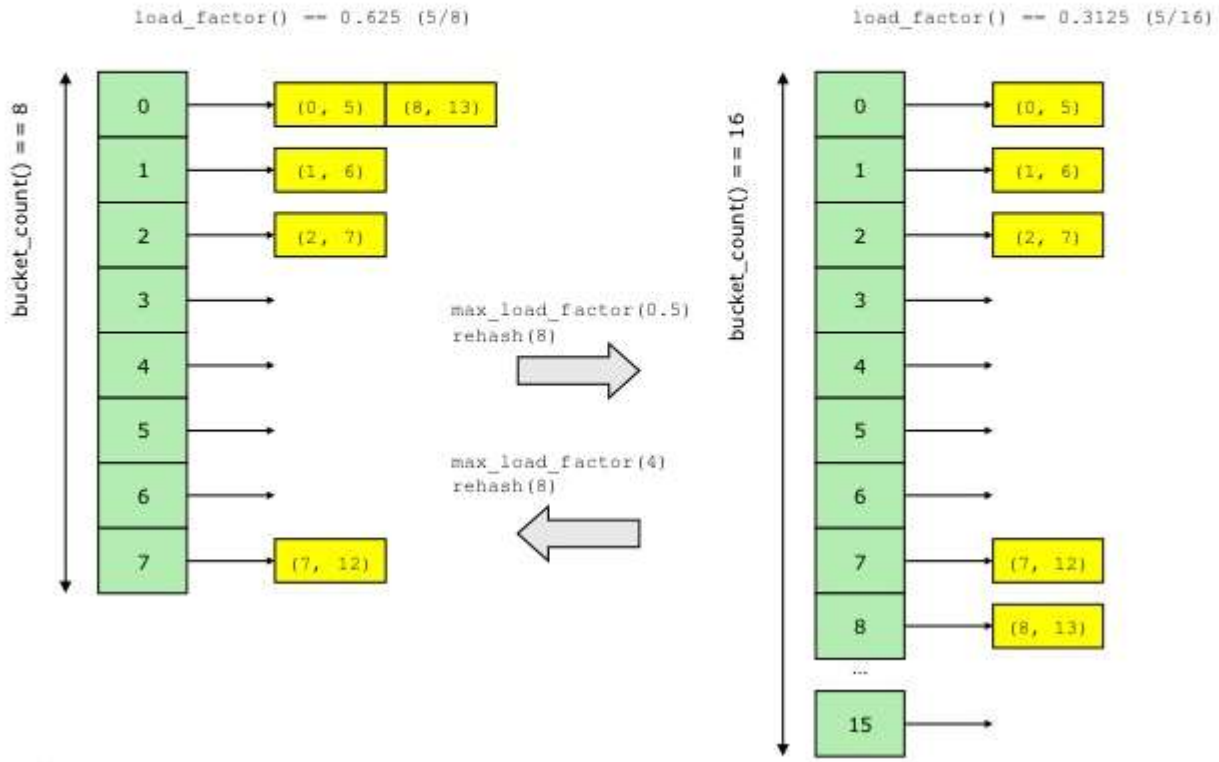
unordered_set/unordered_map: performance



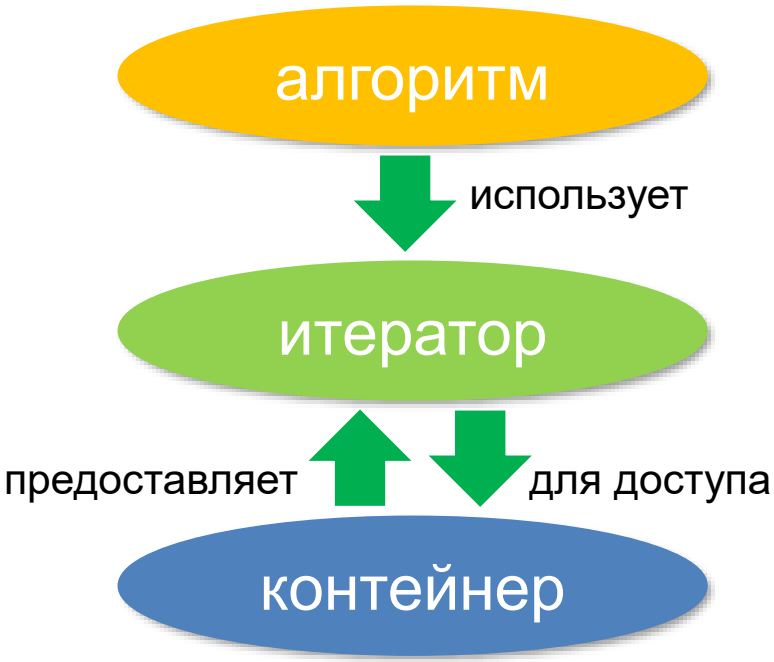
STL: Containers

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



CQG

STL: Algorithms

count, count_if, find, find_if, adjacent_find, for_each, mismatch, equal, search
copy, copy_backward, swap, iter_swap, swap_ranges, fill, fill_n, generate,
generate_n, replace, replace_if, transform, remove, remove_if, remove_copy,
remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate,
rotate_copy, random_shuffle, partition, stable_partition,
sort, stable_sort, partial_sort, partial_sort_copy, nth_element, binary_search,
lower_bound, upper_bound, equal_range, merge, inplace_merge, includes,
set_union, set_intersection, set_difference, set_symmetric_difference,
make_heap, push_heap, pop_heap, sort_heap, min, max, min_element,
max_element, lexographical_compare, next_permutation, prev_permutation
accumulate, inner_product, partial_sum, adjacent_difference,
…

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);
```


CQG

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

Пример:

```
int A[100];

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

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; }
};


C obj;

int a = obj(1);

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


CQG

STL: Functional objects

Пример:

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

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: Algorithms

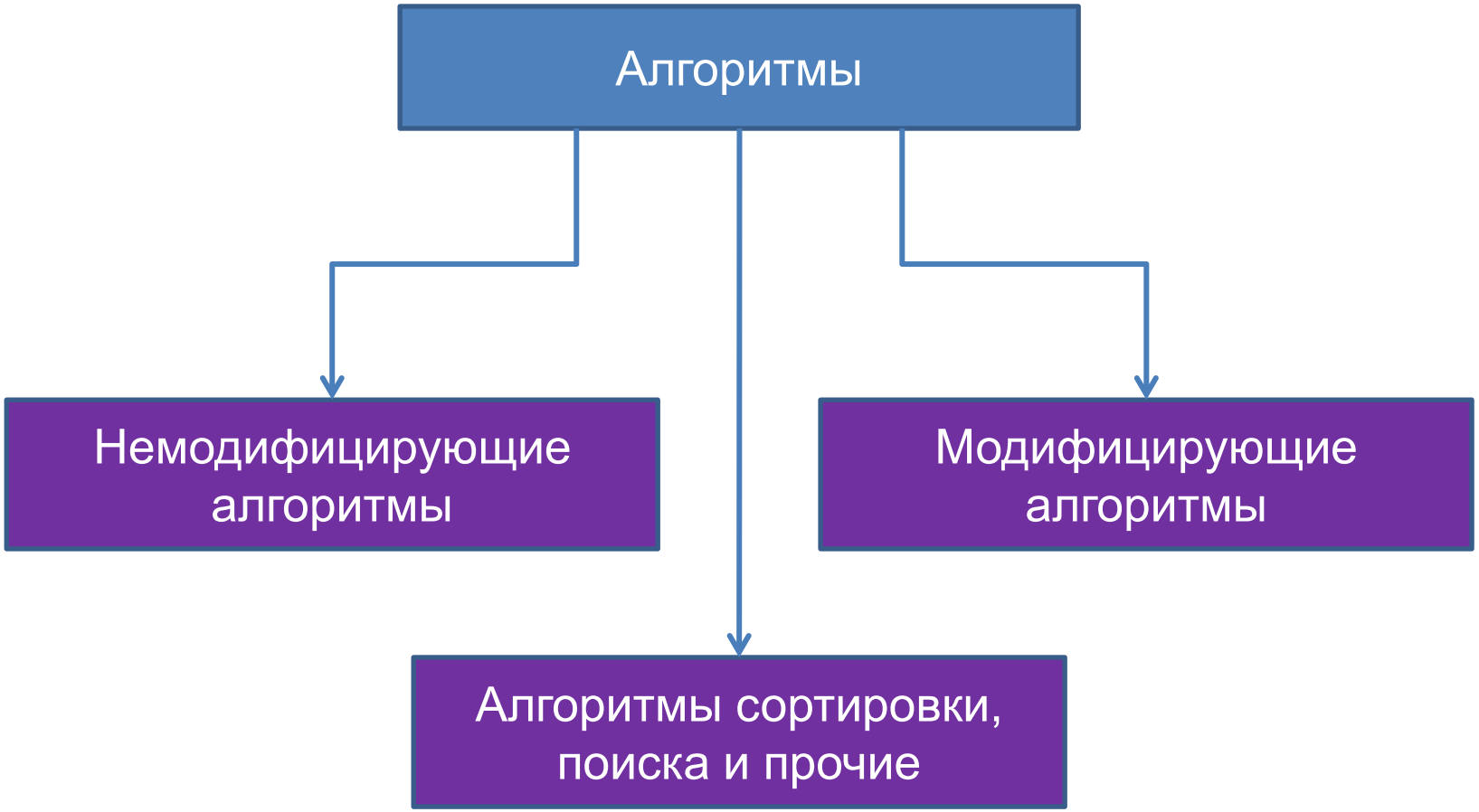
Алгоритмы (почти все) являются **внешними по отношению к классам контейнеров .**

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



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)**.

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).**

CQG

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: Iterator adapters

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: Iterator adapters

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: Iterator adapters

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: Iterator adapters

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: Iterator adapters

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: Removing elements

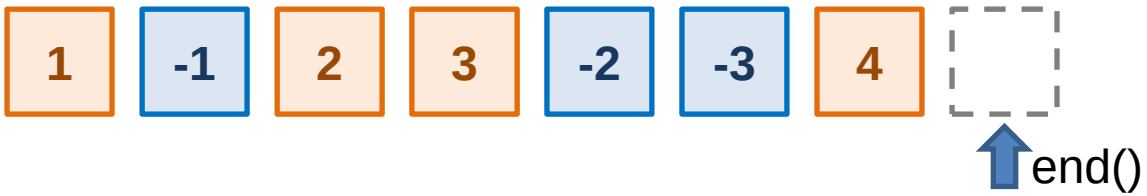
```
template < class ForwardIt, class T >
ForwardIt remove(ForwardIt first, ForwardIt last,
                 const T& value);

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.

STL: Removing elements

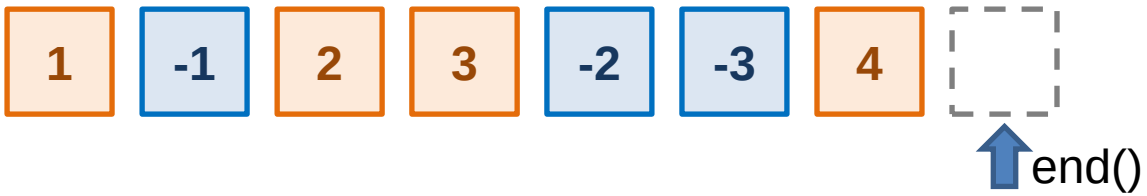
```
vector<int> vec;

// fill it here

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



CQG

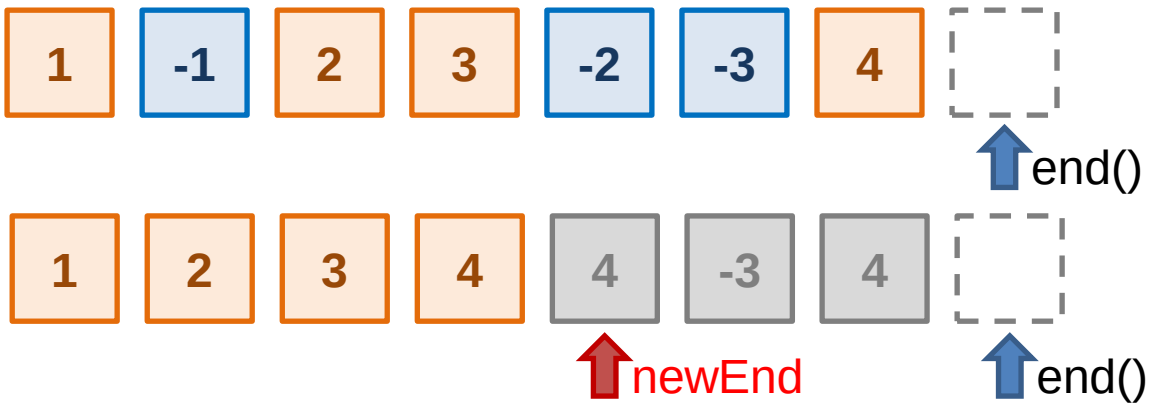
STL: Removing elements

```
vector<int> vec;

// fill it here

auto isNonPositive = [](int x) { return x <= 0; }

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

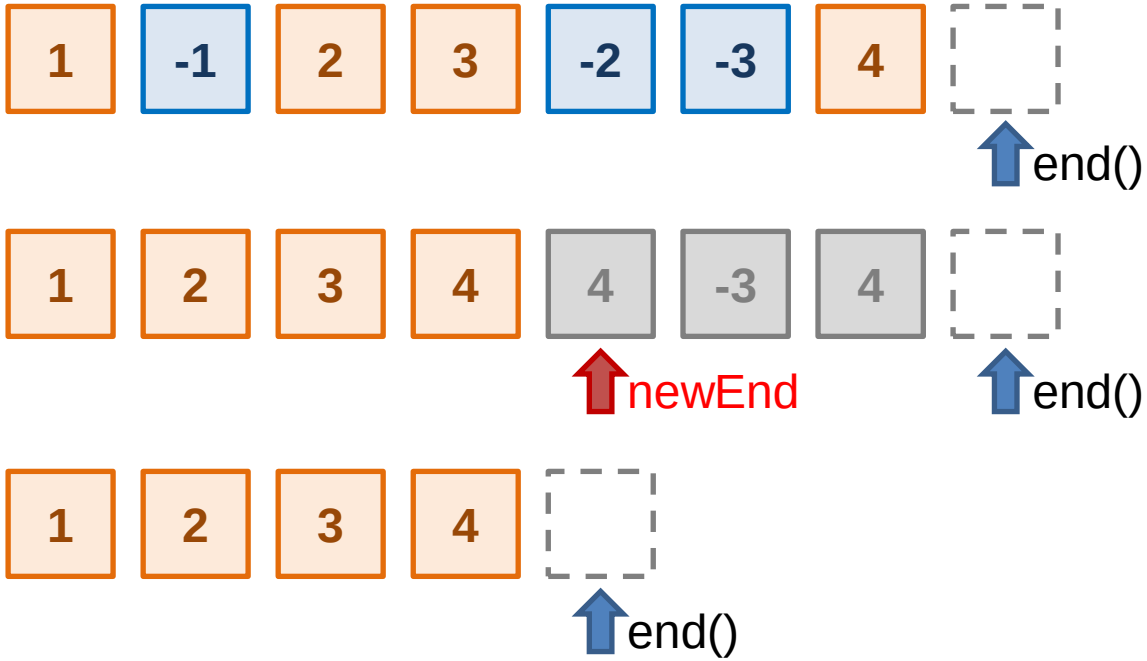


STL: Removing elements

```
vector<int> vec;

// fill it here

auto isNonPositive = [](int x) { return x <= 0; }

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

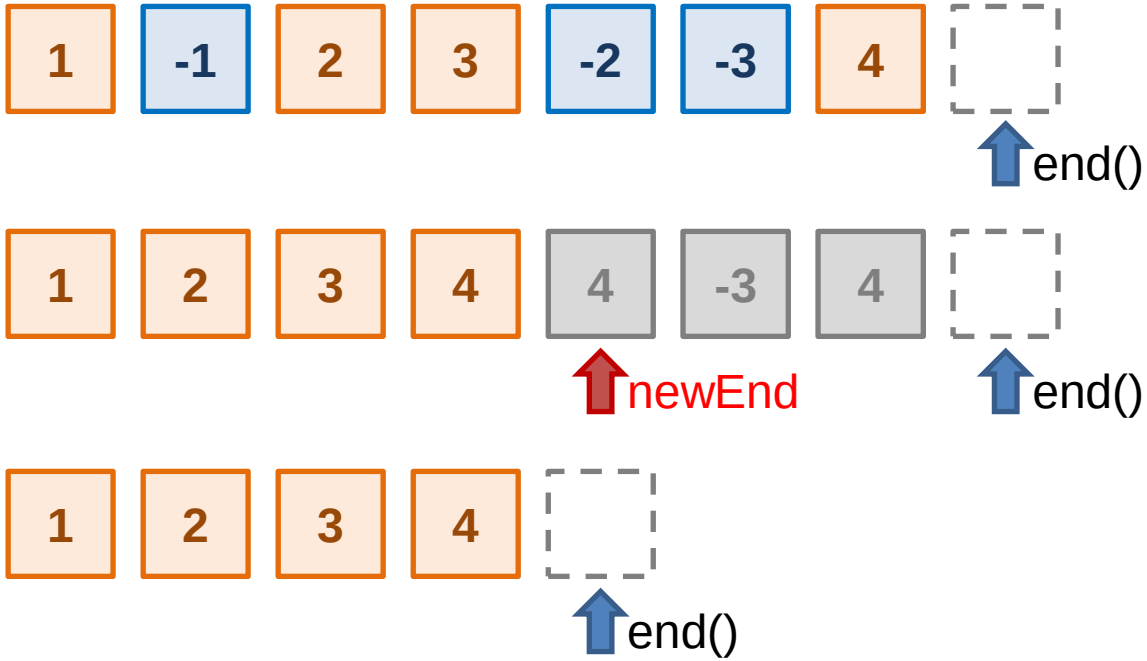


STL: Removing elements



CQG

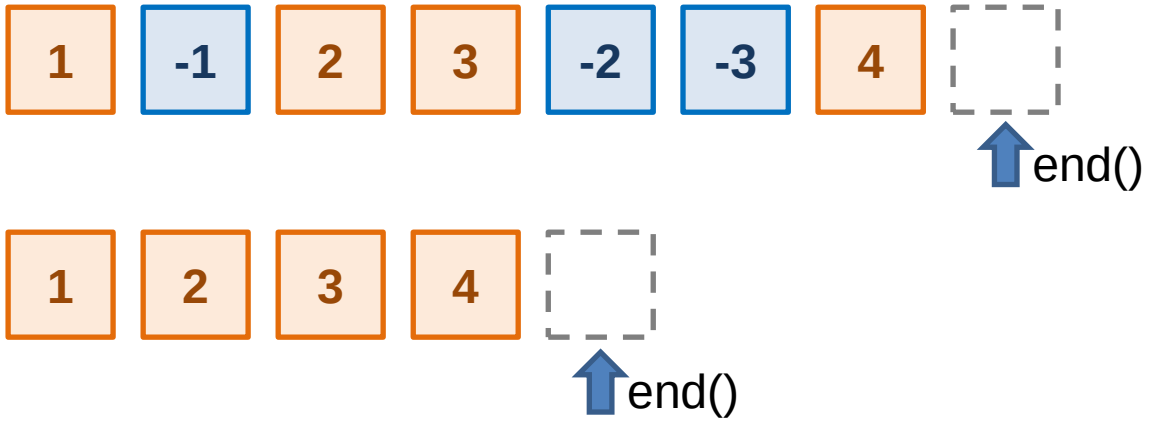
STL: Removing elements




```
const auto 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 );
```

(C)

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/

CQG