# Программирование на языке C++
# Лекция 9

## Стандартная библиотека шаблонов

Александр Смаль

# STL: введение

- STL = Standard Template Library.
- STL является частью стандартной библиотеки C++, описана в стандарте, но не упоминается там явно.
- Авторы: Александр Степанов, Дэвид Муссер и Менг Ли (сначала для HP, а потом для SGI).
- Основана на разработках для языка Ада.

# STL: введение

- STL = Standard Template Library.
- STL является частью стандартной библиотеки C++, описана в стандарте, но не упоминается там явно.
- Авторы: Александр Степанов, Дэвид Муссер и Менг Ли (сначала для HP, а потом для SGI).
- Основана на разработках для языка Ада.

## Основные составляющие

- контейнеры (хранение объектов в памяти),
- итераторы (доступ к элементам контейнера),
- адаптеры (обёртки над контейнерами),
- алгоритмы (для работы с последовательностями),
- функциональные объекты, функторы (обобщение функций).

# Преимущества стандартной библиотеки

- стандартизированность,
- общедоступность,
- эффективность,
- общеизвестность,
- …

# Преимущества стандартной библиотеки

- стандартизированность,
- общедоступность,
- эффективность,
- общеизвестность,
- …

## Чего нет в стандартной библиотеке?

- сложных структур данных,
- сложных алгоритмов,
- работы с графикой/звуком,
- …

# Программирование на языке C++
# Лекция 9

## Последовательные контейнеры STL

Александр Смаль

# Общие сведения о контейнерах

Контейнеры библиотеки STL можно разделить на две категории:

- последовательные,
- ассоциативные.

# Общие сведения о контейнерах

Контейнеры библиотеки STL можно разделить на две категории:

- последовательные,
- ассоциативные.

## Общие вложенные типы

- `Container::value_type`
- `Container::iterator`
- `Container::const_iterator`

# Общие сведения о контейнерах

Контейнеры библиотеки STL можно разделить на две категории:

- последовательные,
- ассоциативные.

## Общие вложенные типы

- `Container::value_type`
- `Container::iterator`
- `Container::const_iterator`

## Общие методы контейнеров

- Все „особенные методы“ и swap.
- `size`, `max_size`, `empty`, `clear`.
- `begin`, `end`, `cbegin`, `cend`.
- Операторы сравнения: `==`, `!=`, `>`, `>=`, `<`, `<=`.

**Примечание:** вся STL определена в пространстве имён `std`.

# Шаблон `array`

Класс-обёртка над статическим массивом.

- `operator[]`, `at`,
- `back`, `front`.
- `fill`,
- `data`.

Позволяет работать с массивом как с контейнером.

```
#include <array>

std::array<std::string, 3> a = {"One", "Two", "Three"};
std::cout << a.size() << std::endl;
std::cout << a[1] << std::endl;

// ошибка времени выполнения
std::cout << a.at(3) << std::endl;
```

# Общие методы остальных последовательных контейнеров

- Конструктор от двух итераторов.
- Конструктор от `count` и `defVal`.
- Конструктор от `std::initializer_list<T>`.
- Методы `back, front`.
- Методы `push_back, emplace_back`
- Методы `assign`.
- Методы `insert`.
- Методы `emplace`.
- Методы `erase` от одного и двух итераторов.

# Шаблон vector

Динамический массив с автоматическим изменением размера при добавлении элементов.

- operator[], at,
- resize,
- capacity, reserve, shrink_to_fit,
- pop_back,
- data.

Позволяет работать со старым кодом.

```
#include <vector>

std::vector<std::string> v = {"One", "Two"};
v.reserve(100);
v.push_back("Three");
v.emplace_back("Four");
legacy_function(v.data(), v.size());
std::cout << v[2] << std::endl;
```

# Шаблон deque

Контейнер с возможностью быстрой вставки и удаления элементов на обоих концах за $O(1)$. Реализован как список указателей на массивы фиксированного размера.

- `operator[]`, `at`,
- `resize`,
- `push_front`, `emplace_front`
- `pop_back`, `pop_front`,
- `shrink_to_fit`.

```
#include <deque>

std::deque<std::string> d = {"One", "Two"};
d.emplace_back("Three");
d.emplace_front("Zero");
std::cout << d[1] << std::endl;
```

# Шаблон `list`

Двусвязный список. В любом месте контейнера вставка и удаление производятся за $O(1)$.

- `push_front`, `emplace_front`,
- `pop_back`, `pop_front`,
- `splice`,
- `merge`, `remove`, `remove_if`, `reverse`, `sort`, `unique`.

```
#include <list>

std::list<std::string> l = {"One", "Two"};
l.emplace_back("Three");
l.emplace_front("Zero");
std::cout << l.front() << std::endl;
```

# Итерация по списку

У списка нет методов для доступа к элементам по индексу.
Можно использовать range-based for:

```
using std::string;
std::list<string> l = {"One", "Two", "Three"};
for (string & s : l)
    std::cout << s << std::endl;
```

Для более сложных операций нужно использовать *итераторы*.

```
std::list<string>::iterator i = l.begin();
for ( ; i != l.end(); ++i) {
    if (*i == "Two")
        break;
}
l.erase(i);
```

Итератор списка можно перемещать в обоих направлениях:

```
auto last = l.end();
--last; // последний элемент
```

# Шаблон forward_list

Односвязный список. В любом месте контейнера вставка и удаление производятся за $O(1)$.

- `insert_after` и `emplace_after` вместо `insert` и `emplace`,
- `before_begin`, `cbefore_begin`,
- `push_front`, `emplace_front`, `pop_front`,
- `splice_after`,
- `merge`, `remove`, `remove_if`, `reverse`, `sort`, `unique`.

```
#include <forward_list>
using std::string;

std::forward_list<string> fl = {"One", "Two"};
fl.emplace_front("Zero");
fl.push_front("Minus one");
std::cout << fl.front() << std::endl;
```

# Шаблон basic_string

Контейнер для хранения символьных последовательностей.

```
typedef basic_string<char>      string;
typedef basic_string<wchar_t>   wstring;
typedef basic_string<char16_t>  u16string;
typedef basic_string<char32_t>  u32string;
```

- Метод c_str() для совместимости со старым кодом,
- поддержка неявных преобразований со строками в стиле C,
- `operator[]`, `at`,
- `reserve`, `capacity`, `shrink_to_fit`,
- `append`, `operator+`, `operator+=`,
- `substr`, `replace`, `compare`,
- `find`, `rfind`, `find_first_of`, `find_first_not_of`, `find_last_of`, `find_last_not_of` (в терминах *индексов*)

# Адаптеры и псевдоконтейнеры

Адаптеры:

- `stack` — реализация интерфейса стека.
- `queue` — реализация интерфейса очереди.
- `priority_queue` — очередь с приоритетом на куче.

Псевдо-контейнеры:

- `vector<bool>`
  - ненастоящий контейнер (не хранит `bool`-ы),
  - использует proxy-объекты.
- `bitset`
  Служит для хранения битовых масок.
  Похож на `vector<bool>`.
- `valarray`
  Шаблон служит для хранения числовых массивов и оптимизирован для достижения повышенной вычислительной производительности.

# Ещё о vector

- Самый универсальный последовательный контейнер.
- Во многих случаях самый эффективный.
- Предпочитайте vector другим контейнерам.
- Интерфейс вектора построен на итераторах, а не на индексах.
- Итераторы вектора ведут себя как указатели.

Использование reserve и capacity:

```
std::vector<int> v;
v.reserve(N); // N — верхняя оценка на размер
...
if (v.capacity() == v.size()) // реаллокация
```

Сжатие и очистка в C++03:

```
std::vector<int> & v = getData();
// shrink_to_fit
std::vector<int>(v).swap(v);
// clear + shrink_to_fit
std::vector<int>().swap(v);
```

# Программирование на языке C++
# Лекция 9

## Ассоциативные контейнеры

Александр Смаль

# Общие сведения

Ассоциативные контейнеры делятся на две группы:

- *упорядоченные* (требуют отношение порядка),
- *неупорядоченные* (требуют хеш-функцию).

## Общие методы

1. `find` по ключу,
2. `count` по ключу,
3. `erase` по ключу.

# Шаблоны set и multiset

set хранит упорядоченное множество (как двоичное дерево поиска).
Операции добавления, удаления и поиска работают за $O(\log n)$.
Значения, которые хранятся в set, неизменяемые.

- lower_bound, upper_bound, equal_range.

```
#include <set>
```

```
std::set<int> primes = {2, 3, 5, 7, 11};
// дальнейшее заполнение
if (primes.find(173) != primes.end())
    std::cout << 173 << " is prime\n";

// std::pair<iterator, bool>
auto res = primes.insert(3);
```

В multiset хранится упорядоченное мультимножество.

```
std::multiset<int> fib = {0, 1, 1, 2, 3, 5, 8};
// iterator
auto res = fib.insert(13);
// pair<iterator, iterator>
auto eq = fib.equal_range(1);
```

# Шаблоны map и multimap

Хранит упорядоченное отображение (как дерево поиска по ключу).
Операции добавления, удаления и поиска работают за $O(\log n)$.

```
    typedef std::pair<const Key, T> value_type;
```

- lower_bound, upper_bound, equal_range,
- operator[], at.

```
#include <map>
```

```
std::map<std::string, int> phonebook;
phonebook.emplace("Marge", 2128506);
phonebook.emplace("Lisa",  2128507);
phonebook.emplace("Bart",  2128507);
// std::map<string,int>::iterator
auto it = phonebook.find("Maggie");
if ( it != phonebook.end())
    std::cout << "Maggie: " << it->second << "\n";
```

```
std::multimap<std::string, int> phonebook;
phonebook.emplace("Homer", 2128506);
phonebook.emplace("Homer", 5552368);
```

# Особые методы map: operator[] и at

```
auto it = phonebook.find("Marge");
if (it != phonebook.end())
     it->second = 5550123;
else
    phonebook.emplace("Marge", 5550123);
// или
phonebook["Marge"] = 5550123;
```

Метод operator[]:

1. работает только с неконстантным map,
2. требует наличие у T конструктора по умолчанию,
3. работает за $O(\log n)$ (не стоит использовать map как массив).

Метод at:

1. генерирует ошибку времени выполнения,
   если такой ключ отсутствует,
2. работает за $O(\log n)$.

# Использование собственного компаратора

Отношение строгого порядка: $\neg(x \prec y) \wedge \neg(y \prec x) \Rightarrow x = y$

```
struct Person { string name; string surname; };

bool operator<(Person const& a, Person const& b) {
    return a.name < b.name ||
           (a.name == b.name && a.surname < b.surname);
}
// уникальны по сочетанию имя + фамилия
std::set<Person> s1;

struct PersonComp {
    bool operator()(Person const& a,
                    Person const& b) const {
        return a.surname < b.surname;
    }
};
// уникальны по фамилии
std::set<Person, PersonComp> s2;
```

# Шаблоны unordered_set и unordered_multiset

unordered_set хранит множество как хеш-таблицу.
Операции добавления, удаления и поиска работают за $O(1)$ в среднем.
Значения, которые хранятся в unordered_set, неизменяемые.

- equal_range, reserve,
- методы для работы с хеш-таблицей.

```
#include <unordered_set>
```

```
unordered_set<int> primes = {2, 3, 5, 7, 11};
// дальнейшее заполнение
if (primes.find(173) != primes.end())
    std::cout << 173 << " is prime\n";

// std::pair<iterator, bool>
auto res = primes.insert(3);
```

В unordered_multiset хранится мультимножество.

```
unordered_multiset<int> fib = {0, 1, 1, 2, 3, 5, 8};
// iterator
auto res = fib.insert(13);
```

# Шаблоны unordered_map и unordered_multimap

Хранит отображение как хеш-таблицу.
Операции добавления, удаления и поиска работают за $O(1)$ в среднем.

- equal_range, reserve, operator[], at,
- методы для работы с хеш-таблицей.

```
#include <unordered_map>
```

```
unordered_map<std::string, int> phonebook;
phonebook.emplace("Marge", 2128506);
phonebook.emplace("Lisa",  2128507);
phonebook.emplace("Bart",  2128507);
// unordered_map<string,int>::iterator
auto it = phonebook.find("Maggie");
if ( it != phonebook.end())
    std::cout << "Maggie: " << it->second << "\n";
```

```
unordered_multimap<std::string, int> phonebook;
phonebook.emplace("Homer", 2128506);
phonebook.emplace("Homer", 5552368);
```

# Использование собственной хеш-функции

```
struct Person { string name; string surname; };

bool operator==(Person const& a, Person const& b) {
    return a.name    == b.name
        && a.surname == b.surname;
}

namespace std {
    template <> struct hash<Person> {
        size_t operator()(Person const& p) const {
            hash<string> h;
            return h(p.name) ^ h(p.surname);
        }
    };
}

// уникальны по сочетанию имя + фамилия
unordered_set<Person> s;
```

# Программирование на языке C++
# Лекция 9

## Итераторы и умные указатели

Александр Смаль

# Категории итераторов

*Итератор* — объект для доступа к элементам последовательности, синтаксически похожий на указатель.
Итераторы делятся на пять категорий.

- Random access iterator: `++`, `--`, арифметика, `<`, `>`, `<=`, `>=`. (`array`, `vector`, `deque`)
- Bidirectional iterator: `++`, `--`. (`list`, `set`, `map`)
- Forward iterator: ++. (`forward_list`, `unordered_set`, `unordered_map`)
- Input iterator: ++, read-only.
- Output iterator: ++, write-only.

Функции для работы с итераторами:

```
void   advance  (Iterator & it, size_t n);
size_t distance (Iterator f, Iterator l);
void   iter_swap(Iterator i, Iterator j);
```

# iterator_traits

```
// заголовочный файл <iterator>
template <class Iterator>
struct iterator_traits {
   typedef difference_type   Iterator::difference_type;
   typedef value_type        Iterator::value_type;
   typedef pointer           Iterator::pointer;
   typedef reference         Iterator::reference;
   typedef iterator_category Iterator::iterator_category;
};

template <class T>
struct iterator_traits<T *> {
    typedef difference_type   ptrdiff_t;
    typedef value_type        T;
    typedef pointer           T*;
    typedef reference         T&;
    typedef iterator_category
            random_access_iterator_tag;
};
```

# iterator_category

```
// <iterator>
struct random_access_iterator_tag {};
struct bidirectional_iterator_tag {};
struct forward_iterator_tag {};
struct input_iterator_tag {};
struct output_iterator_tag {};
```

```
template<class I>
void advance_(I & i, size_t n,
              random_access_iterator_tag)
{ i += n; }
template<class I>
void advance_(I & i, size_t n, ... ) {
   for (size_t k = 0; k != n; ++k, ++i );
}
template<class I>
void advance(I & i, size_t n) {
   advance_(i, n, typename
     iterator_traits<I>::iterator_category());
}
```

# reverse_iterator

У некоторых контейнеров есть обратные итераторы:

```
list<int> l = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// list<int>::reverse_iterator
for(auto i = l.rbegin(); i != l.rend(); ++i)
    cout << *i << endl;
```

Конвертация итераторов:

```
list<int>::iterator i = l.begin();
advance(i, 5); // i указывает на 5
// ri указывает на 4
list<int>::reverse_iterator ri(i);
i = ri.base();
```

Есть возможность сделать обратный итератор из random access или bidirectional при помощи шаблона `reverse_iterator`.

```
// <iterator>
template <class Iterator>
class reverse_iterator {...};
```

# Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

# Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.

# Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.
2. В `vector` и `string` добавление потенциально инвалидирует все итераторы (может произойти выделение нового буфера), иначе инвалидируются только итераторы на все следующие элементы.

# Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.
2. В `vector` и `string` добавление потенциально инвалидирует все итераторы (может произойти выделение нового буфера), иначе инвалидируются только итераторы на все следующие элементы.
3. В `vector` и `string` удаление элемента инвалидирует итераторы на все следующие элементы.

# Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.
2. В `vector` и `string` добавление потенциально инвалидирует все итераторы (может произойти выделение нового буфера), иначе инвалидируются только итераторы на все следующие элементы.
3. В `vector` и `string` удаление элемента инвалидирует итераторы на все следующие элементы.
4. В `deque` удаление/добавление инвалидирует все итераторы, кроме случаев удаления/добавления первого или последнего элементов.

# Advanced итераторы

Для пополнения контейнеров:
`back_inserter, front_inserter, inserter`.

```
// в классе Database
    template<class OutIt>
    void findByName(string name, OutIt out);
```

```
// размер заранее неизвестен
vector<Person> res;
Database::findByName("Rick", back_inserter(res));
```

Для работы с потоками:
`istream_iterator, ostream_iterator`.

```
ifstream file("input.txt");
vector<double> v((istream_iterator<double>(file)),
                 istream_iterator<double>());

copy(v.begin(), v.end(),
     ostream_iterator<double>(cout, "\n"));
```

# Как написать свой итератор

```
// <iterator>
template
<class Category, // iterator::iterator_category
 class T,        // iterator::value_type
 class Distance = ptrdiff_t,// iterator::difference_type
 class Pointer = T*,        // iterator::pointer
 class Reference = T&       // iterator::reference
> class iterator;
```

```
#include <iterator>

struct PersonIterator
    : std::iterator<forward_iterator_tag, Person>
{
// operator++, operator*, ...
};
```

# Умные указатели

## `unique_ptr`

- Умный указатель с уникальным владением.
- Нельзя копировать, можно перемещать.
- Не подходит для разделяемых объектов.

## `shared_ptr`

- Умный указатель с подсчётом ссылок.
- Универсальный указатель.

## `weak_ptr`

- Умный указатель с для создания *слабых ссылок*.
- Работает вместе с `shared_ptr`.

# Программирование на языке C++
# Лекция 9

## Алгоритмы

Александр Смаль

# Функторы и min/max алгоритмы

- *Функтор* — класс, объекты которого ведут себя как функции, т.е. имеет перегруженные `operator()`.
- *Предикат* — функтор, возвращающий `bool`.

# Функторы и min/max алгоритмы

- *Функтор* — класс, объекты которого ведут себя как функции, т.е. имеет перегруженные `operator()`.
- *Предикат* — функтор, возвращающий `bool`.

Функторы в стандартной библиотеке:

- `less`, `greater`, `less_equal`, `greater_equal`, `not_equal_to`, `equal_to`,
- `minus`, `plus`, `divides`, `modulus`, `multiplies`,
- `logical_not`, `logical_and`, `logical_or`
- `bit_and`, `bit_or`, `bit_xor`,
- `hash`.

# Функторы и min/max алгоритмы

- *Функтор* — класс, объекты которого ведут себя как функции, т.е. имеет перегруженные `operator()`.
- *Предикат* — функтор, возвращающий `bool`.

Функторы в стандартной библиотеке:

- `less`, `greater`, `less_equal`, `greater_equal`, `not_equal_to`, `equal_to`,
- `minus`, `plus`, `divides`, `modulus`, `multiplies`,
- `logical_not`, `logical_and`, `logical_or`
- `bit_and`, `bit_or`, `bit_xor`,
- `hash`.

## Алгоритмы min/max

- `min`, `max`, `minmax`,
- `min_element`, `max_element`, `minmax_element`.

# Немодифицирующие алгоритмы

- `all_of`, `any_of`, `none_of`,
- `for_each`,
- `find`, `find_if`, `find_if_not`, `find_first_of`,
- `adjacent_find`,
- `count`, `count_if`,
- `equal`, `mismatch`,
- `is_permutation`,
- `lexicographical_compare`,
- `search`, `search_n`, `find_end`.

## Для упорядоченных последовательностей

- `lower_bound`, `upper_bound`, `equal_range`,
- `set_intersection`, `set_difference`, `set_union`, `set_symmetric_difference`,
- `binary_search`, `includes`.

# Примеры

```
vector<int> v = {2,3,5,7,13,17,19};
size_t c = count_if(v.begin(), v.end(),
                    [](int x) {return x % 2 == 0;});

auto it = lower_bound(v.begin(), v.end(), 11);

bool has7 = binary_search(v.begin(), v.end(), 7);
```

```
vector<string> & db = getNames();
for_each(db.begin(), db.begin() + db.size() / 2,
        [](string & s){cout << s << "\n";});

auto w = find(db.begin(), db.end(), "Waldo");

string agents[3] = {"Alice", "Bob", "Eve"};
auto it = find_first_of(db.begin(), db.end(),
                        agents, agents + 3);
```

# Модифицирующие алгоритмы

- `fill`, `fill_n`, `generate`, `generate_n`,
- `random_shuffle`, `shuffle`,
- `copy`, `copy_n`, `copy_if`, `copy_backward`,
- `move`, `move_backward`,
- `remove`, `remove_if`, `remove_copy`, `remove_copy_if`,
- `replace`, `replace_if`, `replace_copy`, `replace_copy_if`,
- `reverse`, `reverse_copy`,
- `rotate`, `rotate_copy`,
- `swap_ranges`,
- `transform`,
- `unique`, `unique_copy`,
- * `accumulate`, `adjacent_difference`, `inner_product`, `partial_sum`, `iota`.

# Примеры

```
// случайные
vector<int> a(100);
generate(a.begin(), a.end(), [](){return rand() % 100;});

// 0,1,2,3,...
vector<int> b(a.size());
iota(b.begin(), b.end(), 0);

// c[i] = a[i] * b[i]
vector<int> c(b.size());
transform(a.begin(), a.end(), b.begin(),
          c.begin(), multiplies<int>());

// c[i] *= 2
transform(c.begin(), c.end(), c.begin(),
          [](int x) {return x * 2;});

// сумма c[i]
int sum = accumulate(c.begin(), c.end(), 0);
```

# Удаление элементов из последовательности

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
remove(v.begin(), v.end(), 5);
```

Как изменится `v.size()`?

Какое содержимое вектора v?

# Удаление элементов из последовательности

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
remove(v.begin(), v.end(), 5);
```

Как изменится `v.size()`? Не изменится.

Какое содержимое вектора v? `{2,1,8,2,8,5,2,5,8}`

# Удаление элементов из последовательности

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
remove(v.begin(), v.end(), 5);
```

Как изменится `v.size()`? Не изменится.
Какое содержимое вектора v? `{2,1,8,2,8,5,2,5,8}`

Удаление элемента по значению:

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
v.erase(remove(v.begin(), v.end(), 5), v.end());
```

```
list<int> l = {2,5,1,5,8,5,2,5,8};
l.remove(5);
```

# Удаление элементов из последовательности

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
remove(v.begin(), v.end(), 5);
```

Как изменится `v.size()`? Не изменится.
Какое содержимое вектора v? `{2,1,8,2,8,5,2,5,8}`

Удаление элемента по значению:

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
v.erase(remove(v.begin(), v.end(), 5), v.end());
```

```
list<int> l = {2,5,1,5,8,5,2,5,8};
l.remove(5);
```

Удаление одинаковых элементов:

```
vector<int> v = {1,2,2,2,3,4,5,5,5,6,7,8,9};
v.erase(unique(v.begin(), v.end()), v.end());
```

```
list<int> l = {1,2,2,2,3,4,5,5,5,6,7,8,9};
l.unique();
```

# Удаление из ассоциативных контейнеров

Неправильный вариант

```
map<string, int> m;
for (auto it = m.begin(); it != m.end(); ++it)
    if (it->second == 0)
        m.erase(it);
```

# Удаление из ассоциативных контейнеров

Неправильный вариант

```
map<string, int> m;
for (auto it = m.begin(); it != m.end(); ++it)
    if (it->second == 0)
        m.erase(it);
```

Правильный вариант

```
for (auto it = m.begin() ; it != m.end(); )
    if (it->second == 0)
        it = m.erase(it);
    else
        ++it;
```

# Удаление из ассоциативных контейнеров

Неправильный вариант

```
map<string, int> m;
for (auto it = m.begin(); it != m.end(); ++it)
    if (it->second == 0)
        m.erase(it);
```

Правильный вариант

```
for (auto it = m.begin() ; it != m.end(); )
    if (it->second == 0)
        it = m.erase(it);
    else
        ++it;
```

Альтернативный вариант (для старого стандарта)

```
for (map<string,int>::iterator it = m.begin();
                               it != m.end();)
    if (it->second == 0)
        m.erase(it++);
    else
        ++it;
```

# Сортировка

- `is_sorted`, `is_sorted_until`,
- `sort`, `stable_sort`,
- `nth_element`, `partial_sort`,
- `merge`, `inplace_merge`,
- `partition`, `stable_partition`, `is_partitioned`, `partition_copy`, `partition_point`.

# Сортировка

- is_sorted, is_sorted_until,
- sort, stable_sort,
- nth_element, partial_sort,
- merge, inplace_merge,
- partition, stable_partition, is_partitioned, partition_copy, partition_point.

```
vector<int> v = randomVector<int>();

auto med = v.begin() + v.size() / 2;
nth_element(v.begin(), med, v.end());
cout << "Median: " << *med;

auto m = partition(v.begin(), v.end(),
          [](int x){return x % 2 == 0;});
sort(v.begin(), m);
v.erase(m, v.end());
```

# Что есть ещё?

- Операции с кучей:
  - `push_heap`,
  - `pop_heap`,
  - `make_heap`,
  - `sort_heap`
  - `is_heap`,
  - `is_heap_until`.
- Операции с неинициализированными интервалами:
  - `raw_storage_iterator`,
  - `uninitialized_copy`,
  - `uninitialized_fill`,
  - `uninitialized_fill_n`.
- Операции с перестановками
  - `next_permutation`,
  - `prev_permutation`.