STL-2: Пары, кортежи и ассоциативные контейнеры

Denis Bakin

Вопросы для повторения

- Что такое шаблон (template)? Зачем он нужен?

Вопросы для повторения

- Что такое шаблон (template)? Зачем он нужен?
- Что такое итератор? Какие операции с ним можно делать?

Вопросы для повторения

- Что такое шаблон (template)? Зачем он нужен?
- Что такое итератор? Какие операции с ним можно делать?
- Чем отличается `std::set` от `std::unordered_set`?

Вопросы для повторения

- Что такое шаблон (template)? Зачем он нужен?
- Что такое итератор? Какие операции с ним можно делать?
- Чем отличается `std::set` от `std::unordered_set`?
- Какова сложность операций в хэш-таблице? Почему?

std::pair — Пара

Пара — составной тип из двух значений (возможно, разных типов).

```
#include <tuple>

std::pair<int, std::string> p;
```

Поля пары:

- `p.first` — первый элемент
- `p.second` — второй элемент

Создание пары

Три способа создать пару:

```
// Способ 1: конструктор
std::pair<int, std::string> p1(1, "one");

// Способ 2: std::make_pair (автовывод типов)
std::pair<int, std::string> p2 = std::make_pair(2, "two");

// Способ 3: список инициализации (C++11)
std::pair<int, std::string> p3{3, "three"};
```

Работа с парой

```cpp
std::pair<int, std::string> p = {10, "apple"};

// Чтение
std::cout << p.first;   // 10
std::cout << p.second;  // "apple"

// Изменение
p.first = 20;
p.second = "banana";
```

Сравнение пар

Пары сравниваются **лексикографически**:

1. Сначала сравниваются `first`
2. Если равны — сравниваются `second`

```
std::pair<int, int> a = {1, 5};
std::pair<int, int> b = {1, 3};
std::pair<int, int> c = {2, 1};

std::cout << (a < b);  // 0 (false), т.к. 5 > 3
std::cout << (a < c);  // 1 (true),  т.к. 1 < 2
std::cout << (a == b); // 0 (false)
```

Вектор пар

Часто пары используются в векторах:

```
std::vector<std::pair<int, int>> points;

points.push_back({10, 20});
points.push_back(std::make_pair(30, 40));
points.emplace_back(50, 60); // создаёт пару "на месте"

for (const std::pair<int, int>& p : points) {
    std::cout << "(" << p.first << ", " << p.second << ")\n";
}
```

Вывод:

```
(10, 20)
(30, 40)
(50, 60)
```

std::tuple — Кортеж

Кортеж — обобщение пары на произвольное число элементов.

```
#include <tuple>

std::tuple<int, std::string, double> t = {1, "hello", 3.14};
```

Доступ к элементам — через `std::get<index>`:

```
std::cout << std::get<0>(t);  // 1
std::cout << std::get<1>(t);  // "hello"
std::cout << std::get<2>(t);  // 3.14

std::get<0>(t) = 42;  // изменение
```

Сравнение кортежей

Кортежи тоже сравниваются лексикографически:

```
std::tuple<int, std::string> t1 = {2, "world"};

std::cout << (t1 < std::make_tuple(5, "abc"));    // 1 (true)
std::cout << (t1 < std::make_tuple(1, "zzz"));    // 0 (false)
std::cout << (t1 < std::make_tuple(2, "zebra")); // 1 (true)
```

Сравнение кортежей

Кортежи тоже сравниваются лексикографически:

```
std::tuple<int, std::string> t1 = {2, "world"};

std::cout << (t1 < std::make_tuple(5, "abc"));    // 1 (true)
std::cout << (t1 < std::make_tuple(1, "zzz"));    // 0 (false)
std::cout << (t1 < std::make_tuple(2, "zebra")); // 1 (true)
```

Правило: Сравниваем по порядку: сначала первый элемент, затем второй, и т.д.

Зачем нужны пары и кортежи?

- Возврат нескольких значений из функции

Зачем нужны пары и кортежи?

- Возврат нескольких значений из функции
- Хранение связанных данных (координаты, ключ-значение)

Зачем нужны пары и кортежи?

- Возврат нескольких значений из функции
- Хранение связанных данных (координаты, ключ-значение)
- Сортировка по нескольким критериям (лексикографическое сравнение)

Зачем нужны пары и кортежи?

- Возврат нескольких значений из функции
- Хранение связанных данных (координаты, ключ-значение)
- Сортировка по нескольким критериям (лексикографическое сравнение)
- Элементы ассоциативных контейнеров (`std::map`, `std::unordered_map`)

std::unordered_map — Словарь

Ассоциативный контейнер: хранит пары **Ключ** → **Значение**.

```
#include <unordered_map>

std::unordered_map<std::string, int> ages;
```

- Ключи уникальны

std::unordered_map — Словарь

Ассоциативный контейнер: хранит пары **Ключ** → **Значение**.

```
#include <unordered_map>

std::unordered_map<std::string, int> ages;
```

- Ключи уникальны
- Доступ по ключу за $O(1)$ (хэш-таблица)

std::unordered_map — Словарь

Ассоциативный контейнер: хранит пары **Ключ** → **Значение**.

```
#include <unordered_map>

std::unordered_map<std::string, int> ages;
```

- Ключи уникальны
- Доступ по ключу за $O(1)$ (хэш-таблица)
- Порядок элементов не определён

Добавление элементов

```cpp
std::unordered_map<std::string, int> ages;

// Способ 1: оператор []
ages["Alice"] = 25;
ages["Bob"] = 30;

// Способ 2: insert с парой
ages.insert({"Charlie", 35});
ages.insert(std::make_pair("Diana", 28));
```

Добавление элементов

```cpp
std::unordered_map<std::string, int> ages;

// Способ 1: оператор []
ages["Alice"] = 25;
ages["Bob"] = 30;

// Способ 2: insert с парой
ages.insert({"Charlie", 35});
ages.insert(std::make_pair("Diana", 28));
```

Важно: `ages["Eve"]` создаст элемент со значением 0, если ключа “Eve” не было!

Чтение и изменение

```cpp
std::unordered_map<std::string, int> ages = {
    {"Alice", 25}, {"Bob", 30}
};

// Чтение
std::cout << ages["Alice"];  // 25

// Изменение
ages["Alice"] = 26;
ages["Bob"] += 1;  // Bob теперь 31
```

Поиск элемента: find()

Метод `find(key)` возвращает **итератор**:

- На найденный элемент, если ключ существует
- На `end()`, если ключа нет

```
std::unordered_map<std::string, int> ages = {{"Alice", 25}};

auto it = ages.find("Alice");
if (it != ages.end()) {
    std::cout << "Найден: " << it->first << " = " << it->second;
} else {
    std::cout << "Не найден";
}
```

Оператор стрелка (->)

Итератор указывает на **пару** {ключ, значение}.

Два эквивалентных способа доступа:

```
auto it = ages.find("Alice");

// Способ 1: разыменование + точка
std::cout << (*it).first << " " << (*it).second;

// Способ 2: стрелка (удобнее!)
std::cout << it->first << " " << it->second;
```

Оператор стрелка (->)

Итератор указывает на **пару** {ключ, значение}.

Два эквивалентных способа доступа:

```
auto it = ages.find("Alice");

// Способ 1: разыменование + точка
std::cout << (*it).first << " " << (*it).second;

// Способ 2: стрелка (удобнее!)
std::cout << it->first << " " << it->second;
```

Правило: `ptr->field` эквивалентно `(*ptr).field`

Удаление элемента

```
std::unordered_map<std::string, int> ages = {
    {"Alice", 25}, {"Bob", 30}, {"Charlie", 35}
};

// Удаление по ключу
ages.erase("Bob");

// Удаление по итератору
auto it = ages.find("Charlie");
if (it != ages.end()) {
    ages.erase(it);
}
```

Итерация по map

```cpp
std::unordered_map<std::string, int> ages = {
    {"Alice", 25}, {"Bob", 30}
};

// Range-based for (каждый элемент - пара)
for (const std::pair<const std::string, int>& p : ages) {
    std::cout << p.first << ": " << p.second << "\n";
}

// Или с auto
for (const auto& [name, age] : ages) {  // C++17
    std::cout << name << ": " << age << "\n";
}
```

Безопасный доступ: at()

Метод `at(key)` выбрасывает исключение, если ключа нет:

```
std::unordered_map<std::string, int> ages = {{"Alice", 25}};

std::cout << ages.at("Alice");  // OK: 25
std::cout << ages.at("Bob");    // Ошибка! std::out_of_range
```

Безопасный доступ: at()

Метод `at(key)` выбрасывает исключение, если ключа нет:

```
std::unordered_map<std::string, int> ages = {{"Alice", 25}};

std::cout << ages.at("Alice");  // OK: 25
std::cout << ages.at("Bob");    // Ошибка! std::out_of_range
```

Используйте `at()` с константными map:

```
void print(const std::unordered_map<std::string, int>& m) {
    // m["Alice"] - ошибка компиляции ([] не const)
    std::cout << m.at("Alice");  // OK
}
```

[] vs find() vs at()

Метод	Если ключа нет	Можно с const?
`map[key]`	Создаёт элемент (0)	Нет
`map.find(key)`	Возвращает `end()`	Да
`map.at(key)`	Исключение	Да

[] vs find() vs at()

Метод	Если ключа нет	Можно с const?
`map[key]`	Создаёт элемент (0)	Нет
`map.find(key)`	Возвращает `end()`	Да
`map.at(key)`	Исключение	Да

Рекомендация:

- Для проверки существования — `find()`
- Для гарантированного доступа — `at()`
- Для вставки/изменения — `[]`

Пример: подсчёт слов

```cpp
#include <iostream>
#include <unordered_map>
#include <string>

int main() {
    std::unordered_map<std::string, int> word_counts;
    std::string word;

    while (std::cin >> word) {
        word_counts[word]++;  // создаёт 0 и увеличивает
    }

    for (const auto& [w, count] : word_counts) {
        std::cout << w << ": " << count << "\n";
    }
}
```

Пример: подсчёт слов (трассировка)

Ввод: `apple banana apple cherry apple`

Шаг	word	word_counts
1	apple	{apple: 1}
2	banana	{apple: 1, banana: 1}
3	apple	{apple: 2, banana: 1}
4	cherry	{apple: 2, banana: 1, cherry: 1}
5	apple	{apple: 3, banana: 1, cherry: 1}

std::map — Упорядоченный словарь

Аналог unordered_map, но ключи **отсортированы**.

```cpp
#include <map>

std::map<std::string, int> ages;
ages["Charlie"] = 35;
ages["Alice"] = 25;
ages["Bob"] = 30;

for (const auto& [name, age] : ages) {
    std::cout << name << ": " << age << "\n";
}
// Вывод в алфавитном порядке:
// Alice: 25
// Bob: 30
// Charlie: 35
```

Сравнение map и unordered_map

Характеристика	`std::unordered_map`	`std::map`
Порядок ключей	Произвольный	Отсортированный
Реализация	Хэш-таблица	Дерево (Red-Black)
Вставка/Поиск	$O(1)$	$O(\log n)$
Требования к ключу	`hash()`, `==`	`<` (сравнение)

Когда что использовать?

- `std::unordered_map`: когда важна скорость, порядок не нужен

Когда что использовать?

- `std::unordered_map`: когда важна скорость, порядок не нужен
- `std::map`: когда нужен порядок ключей или операции с диапазонами

Когда что использовать?

- `std::unordered_map`: когда важна скорость, порядок не нужен
- `std::map`: когда нужен порядок ключей или операции с диапазонами
- как поступить, если нужна сортировка пар по значению?

Когда что использовать?

- `std::unordered_map`: когда важна скорость, порядок не нужен
- `std::map`: когда нужен порядок ключей или операции с диапазонами
- как поступить, если нужна сортировка пар по значению?
- `std::vector<std::pair>`: нужна сортировка по значению

Итоги

- `std::pair` — два значения, доступ через `.first`, `.second`
- `std::tuple` — несколько значений, доступ через `std::get<i>()`
- Пары и кортежи сравниваются лексикографически
- `std::unordered_map` — словарь (ключ $\to$ значение), $O(1)$
- `std::map` — упорядоченный словарь, $O(\log n)$
- Итератор на пару: `it->first`, `it->second`