Шаблоны и STL. Итераторы, множества, пары

Denis Bakin

Проблема: Неявное преобразование типов

Рассмотрим функцию сравнения двух чисел:

```
bool isGreater(int a, int b) {
    return a > b;
}

int main() {
    double a = 1.9, b = 1.1;
    // Вызовется isGreater(int, int)!
    // 1.9 -> 1, 1.1 -> 1. Результат: 1 > 1 -> false (0)
    std::cout << isGreater(a, b);
}
```

Проблема: Происходит потеря данных из-за неявного преобразования `double -> int`. Чтобы исправить, нужно написать `isGreater(double, double)`.

Мотивация: Дублирование кода

Если мы напишем перегрузки:

```
bool isGreater(int a, int b) {
    return a > b;
}

bool isGreater(double a, double b) {
    return a > b;
}
```

Мотивация: Дублирование кода

Если мы напишем перегрузки:

```
bool isGreater(int a, int b) {
    return a > b;
}

bool isGreater(double a, double b) {
    return a > b;
}
```

- Код дублируется (логика одинаковая)

Мотивация: Дублирование кода

Если мы напишем перегрузки:

```
bool isGreater(int a, int b) {
    return a > b;
}

bool isGreater(double a, double b) {
    return a > b;
}
```

- Код дублируется (логика одинаковая)
- Для каждого нового типа нужна новая функция

Мотивация: Дублирование кода

Если мы напишем перегрузки:

```
bool isGreater(int a, int b) {
    return a > b;
}

bool isGreater(double a, double b) {
    return a > b;
}
```

- Код дублируется (логика одинаковая)
- Для каждого нового типа нужна новая функция
- Сложно поддерживать

Шаблоны (Templates)

C++ позволяет написать алгоритм один раз для **произвольного** типа T.

```
template <typename T>
bool isGreater(const T& a, const T& b) {
    return a > b;
}
```

При использовании:

```
isGreater(1, 2);       // T выводится как int
isGreater(1.5, 2.5);   // T выводится как double
isGreater("a", "b");   // T выводится как const char* (или string)
```

Компилятор **сам генерирует** нужные перегрузки на этапе компиляции.

STL — Standard Template Library

Стандартная библиотека шаблонов C++. Основные компоненты:

- **Контейнеры**: хранение данных (`std::vector`, `std::array`, `std::string`, ...)

STL — Standard Template Library

Стандартная библиотека шаблонов C++. Основные компоненты:

- **Контейнеры**: хранение данных (`std::vector`, `std::array`, `std::string`, ...)
- **Итераторы**: доступ к данным и навигация по ним

STL — Standard Template Library

Стандартная библиотека шаблонов C++. Основные компоненты:

- **Контейнеры**: хранение данных (`std::vector`, `std::array`, `std::string`, ...)
- **Итераторы**: доступ к данным и навигация по ним
- **Алгоритмы**: обработка данных (`std::sort`, `std::find`, `std::reverse`, ...)

Итераторы

Итератор — объект, который указывает на элемент контейнера. Похож на "умный" указатель.

Основные операции:

- `*it` — получение значения (разыменование)
- `++it` — переход к следующему элементу
- `it1 == it2`, `it1 != it2` — сравнение

Методы контейнеров:

- `begin()` — итератор на **первый** элемент
- `end()` — итератор на позицию **после последнего** элемента

Работа с итераторами

```cpp
std::vector<int> v = {5, 2, 4, 1, 3};

// Сортировка с помощью итераторов
std::sort(v.begin(), v.end());

// Ручной проход
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << " ";
}
```

Range-based for (`for (int x : v)`) — это “сахар” поверх итераторов.

std::unordered_set

Неупорядоченное множество. Хранит **уникальные** элементы в произвольном порядке.

- Реализация: Хэш-таблица

std::unordered_set

Неупорядоченное множество. Хранит **уникальные** элементы в произвольном порядке.

- Реализация: Хэш-таблица
- Сложность операций (в среднем):

std::unordered_set

Неупорядоченное множество. Хранит **уникальные** элементы в произвольном порядке.

- Реализация: Хэш-таблица
- Сложность операций (в среднем):
  - Вставка (`insert`): $O(1)$

std::unordered_set

Неупорядоченное множество. Хранит **уникальные** элементы в произвольном порядке.

- Реализация: Хэш-таблица
- Сложность операций (в среднем):
  - Вставка (`insert`): $O(1)$
  - Удаление (`erase`): $O(1)$

std::unordered_set

Неупорядоченное множество. Хранит **уникальные** элементы в произвольном порядке.

- Реализация: Хэш-таблица
- Сложность операций (в среднем):
  - Вставка (`insert`): $O(1)$
  - Удаление (`erase`): $O(1)$
  - Поиск (`find`): $O(1)$

std::unordered_set

Неупорядоченное множество. Хранит **уникальные** элементы в произвольном порядке.

- Реализация: Хэш-таблица
- Сложность операций (в среднем):
  - Вставка (`insert`): $O(1)$
  - Удаление (`erase`): $O(1)$
  - Поиск (`find`): $O(1)$
- В худшем случае (коллизии): $O(n)$

Хэш

Хэш-функция – функция, которая преобразует объект в число.

Хэш должен удовлетворять следующим свойствам:

- $x = y \Rightarrow h(x) = h(y)$
- быстрое вычисление
- минимизация коллизий
- необратимость

В STL C++ `std::hash<T>` – это такой класс, который может преобразовать объект типа `T` в число.

Например, хэширование применяется для хранения паролей в базе данных пользователей.

Как работает хэш-таблица

1. **Массив (Buckets)**: Основа таблицы — массив списков (“корзин”).
2. **Хэш-функция**: Преобразует объект (ключ) в число (хэш).
   - `index = hash(key) % array_size`
3. **Вставка**: Вычисляем индекс и кладем элемент в соответствующий список.

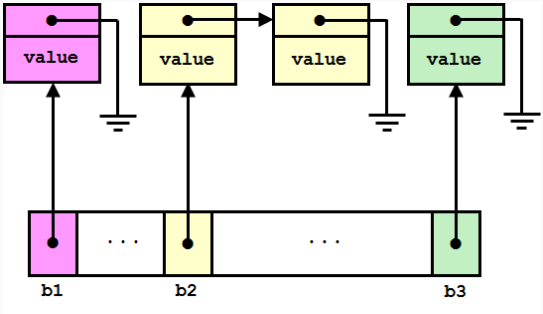

Figure 1: Хэш-таблица

Коллизии и почему это быстро

Коллизия — ситуация, когда у разных ключей совпадает индекс корзины. Элементы с одинаковым индексом хранятся в виде связного списка в одной корзине.

Почему $O(1)$?

- Вычисление хэша — быстрая математическая операция.
- Доступ к ячейке массива по индексу — мгновенно.
- Если корзины короткие (мало коллизий), поиск в списке тоже мгновенный.

Рехеширование (Rehashing)

Что если элементов станет слишком много? Списки в корзинах станут длинными $\rightarrow O(n)$.

Load Factor (Коэффициент заполнения) $= \frac{\text{кол-во элементов}}{\text{кол-во корзин}}$

Механизм рехеширования:

Рехеширование (Rehashing)

Что если элементов станет слишком много? Списки в корзинах станут длинными $\rightarrow O(n)$.

Load Factor (Коэффициент заполнения) $= \frac{\text{кол-во элементов}}{\text{кол-во корзин}}$

Механизм рехеширования:

1. Если Load Factor превышает порог (обычно 1.0), таблица увеличивается (обычно в 2 раза).

Рехеширование (Rehashing)

Что если элементов станет слишком много? Списки в корзинах станут длинными $\rightarrow O(n)$.

Load Factor (Коэффициент заполнения) $= \frac{\text{кол-во элементов}}{\text{кол-во корзин}}$

Механизм рехеширования:

1. Если Load Factor превышает порог (обычно 1.0), таблица увеличивается (обычно в 2 раза).
2. Для **всех** элементов заново вычисляются индексы (так как `array_size` изменился).

Рехеширование (Rehashing)

Что если элементов станет слишком много? Списки в корзинах станут длинными $\rightarrow O(n)$.

Load Factor (Коэффициент заполнения) $= \frac{\text{кол-во элементов}}{\text{кол-во корзин}}$

Механизм рехеширования:

1. Если Load Factor превышает порог (обычно 1.0), таблица увеличивается (обычно в 2 раза).
2. Для **всех** элементов заново вычисляются индексы (так как `array_size` изменился).
3. Элементы распределяются по новым корзинам.

Рехеширование (Rehashing)

Что если элементов станет слишком много? Списки в корзинах станут длинными $\rightarrow O(n)$.

Load Factor (Коэффициент заполнения) $= \frac{\text{кол-во элементов}}{\text{кол-во корзин}}$

Механизм рехеширования:

1. Если Load Factor превышает порог (обычно 1.0), таблица увеличивается (обычно в 2 раза).
2. Для **всех** элементов заново вычисляются индексы (так как `array_size` изменился).
3. Элементы распределяются по новым корзинам.

Рехеширование (Rehashing)

Что если элементов станет слишком много? Списки в корзинах станут длинными $\rightarrow O(n)$.

Load Factor (Коэффициент заполнения) $= \frac{\text{кол-во элементов}}{\text{кол-во корзин}}$

Механизм рехеширования:

1. Если Load Factor превышает порог (обычно 1.0), таблица увеличивается (обычно в 2 раза).
2. Для **всех** элементов заново вычисляются индексы (так как `array_size` изменился).
3. Элементы распределяются по новым корзинам.

Это дорого ($O(n)$), но происходит редко. Амортизированная сложность остается $O(1)$.

std::set

Упорядоченное множество. Хранит **уникальные** элементы в отсортированном порядке.

- Реализация: Сбалансированное дерево поиска (обычно Red-Black Tree)

Используйте, когда важен **порядок** элементов или нужны операции с диапазонами (lower_bound).

std::set

Упорядоченное множество. Хранит **уникальные** элементы в отсортированном порядке.

- Реализация: Сбалансированное дерево поиска (обычно Red-Black Tree)
- Сложность операций:

Используйте, когда важен **порядок** элементов или нужны операции с диапазонами (lower_bound).

std::set

Упорядоченное множество. Хранит **уникальные** элементы в отсортированном порядке.

- Реализация: Сбалансированное дерево поиска (обычно Red-Black Tree)
- Сложность операций:
  - Вставка: $O(\log n)$

Используйте, когда важен **порядок** элементов или нужны операции с диапазонами (lower_bound).

std::set

Упорядоченное множество. Хранит **уникальные** элементы в отсортированном порядке.

- Реализация: Сбалансированное дерево поиска (обычно Red-Black Tree)
- Сложность операций:
  - Вставка: $O(\log n)$
  - Удаление: $O(\log n)$

Используйте, когда важен **порядок** элементов или нужны операции с диапазонами (lower_bound).

std::set

Упорядоченное множество. Хранит **уникальные** элементы в отсортированном порядке.

- Реализация: Сбалансированное дерево поиска (обычно Red-Black Tree)
- Сложность операций:
  - Вставка: $O(\log n)$
  - Удаление: $O(\log n)$
  - Поиск: $O(\log n)$

Используйте, когда важен **порядок** элементов или нужны операции с диапазонами (lower_bound).

Сравнение множеств

Характеристика	`std::unordered_set`	`std::set`
Порядок	Нет (случайный)	Сортированный
Реализация	Хэш-таблица	Дерево
Вставка/Поиск	$O(1)$ (быстрее)	$O(\log n)$
Принцип работы	Хэш -> Индекс	Сравнение (Less)

Почему хэш-таблица быстрее? Дереву нужно сделать $C \cdot \log_2 N$ сравнений, спускаясь от корня. Хэш-таблица "прыгает" сразу в нужную ячейку памяти. При $N = 10^6$, $\log N \approx 20$ операций vs 1 операция.

std::pair

Простая структура для хранения двух разнородных значений.

```cpp
#include <utility>

std::pair<int, std::string> p = {1, "apple"};

p.first = 2;
std::cout << p.second; // "apple"

// Вектор пар
std::vector<std::pair<int, int>> points;
points.push_back({10, 20});
```

Сравнение пар (<, ==) происходит лексикографически: сначала сравниваются `first`, затем `second`.