# Алгоритмические основы работы unordered_map в C++

Сафронов Ярослав

## 1. Что это такое, зачем нужна эта структура и где её можно применять

**Хеш-таблица** - достаточно популярная структура данных в современном мире. В C++ она реализована в виде контейнера `unordered_map`. Данный контейнер позволяет выполнять добавление, поиск и удаление элементов в среднем за O(1), то есть за константное время, что делает его незаменимым для задач, где нужно быстро получать данные по ключу. В повседневной жизни структуры подобного типа используются в кэшах для ускорения доступа к данным, поисковых системах, базах данных и ещё много где.

Если хранить пары "ключ - значение" в обычном массиве или векторе, поиск будет занимать O(n), так как элементы приходится просматривать последовательно. Структуры на деревьях (например, map в C++) сокращают время поиска до O(logn), но для некоторых задач этого недостаточно. Собственно, хеш-таблица имеет следующие плюсы:

- Не требует числового индекса, как массив
- Работает быстрее, чем другие структуры
- Способна принимать произвольные ключи (строки, числа, пары...)

## 2. Основная идея хеширования

Основа работы *unordered_map* - это преобразование произвольного ключа в индекс массива с помощью хеш-функции. *Хеш-функция* принимает ключ (например, строку или число) и вычисляет для него целое значение фиксированного размера. Затем это число приводится к диапазону индексов таблицы, например с помощью операции взятия остатка. Таким образом, ключ как бы переводится в позицию, где будет лежать связанное с ним значение.

Данный подход позволяет обращаться к элементам почти как по индексу массива, поэтому взаимодействие может происходить за константу. Однако, всё не так гладко. Несколько совершенно разных ключей могут попадать в один и тот же индекс (bucket) - это стандартная ситуация, которая назвается коллизией. Обычно в хеш-таблицах такие ситуации решаются благодаря хранению в каждом бакете списка элементов, которые получили одинаковый индекс. При удачном выборе хеш-функции эти цепочки короткие, что и обеспечивает высокую скорость работы.

Далее будут разобраны основные внутренние механизмы *unordered_map*: устройство хеш-таблицы, обработка коллизий, процесс вставки и поиска элементов, а также то, как и когда структура автоматически увеличивает свой размер.

## 3. Хеш-функции: какие бывают и какие требования к ним предъявляются

Начать стоит с того, что стандарт C++ не определяет конкретную hash-функцию для контейнера unordered_map. Реальная реализация зависит от стандартной библиотеки, используемой данным компилятором: GCC, Clang и MSVC применяют разные стратегии. Тем не менее стандарт предъявляет несколько важных требований:

1. Вычисление хеша должно выполняться за постоянное время и не зависеть от размера структуры данных.
2. Результат функции должен быть детерминирован для одного запуска программы (один и тот же ключ обязан давать один и тот же хеш).
3. Равномерность распределения. Хорошая хеш-функция должна распределять ключи по бакетам так, чтобы цепочки не концентрировались в нескольких местах. Это снижает вероятность длинных списков и сохраняет среднюю сложность операций около O(1).

В стандартных библиотеках компиляторы используют разные техники для построения `std::hash<T>`. Для числовых типов данных значения часто возвращаются практически без изменений, а для строк применяется более сложное перемешивание битов. Например, многие реализации используют лёгкие побитовые преобразования или модифицированные варианты алгоритмов семейства FNV (Fowler-Noll-Vo).

### Пример хеш-функции (FNV-1a)

Ниже приведён упрощённый пример одной из классических хеш-функций - FNV-1a. Она основана на побитовом XOR и умножении на фиксированное простое число, благодаря чему получается достаточно равномерное распределение значений.

```
uint32_t fnv1a_hash(const std::string& s) {
    const uint32_t FNV_offset = 0x811c9dc5;
    const uint32_t FNV_prime  = 0x01000193;

    uint32_t hval = FNV_offset;

    for (unsigned char c : s) {
        hval ^= c;            // добавляем в хеш
        hval *= FNV_prime;    // перемешиваем
    }

    return hval;
}
```

В этой функции каждый символ строки последовательно "вмешивается" в текущее хеш-значение: сначала через XOR, затем через умножение. Такое простое смешивание обеспечивает приемлемую равномерность распределения и демонстрирует общий принцип работы практичных хеш-функций.

## 4. Устройство хеш-таблицы в unordered_map

Внутри unordered_map используется массив бакетов. Каждый бакет - это ячейка, в которую попадают элементы с одинаковым индексом после

применения хеш-функции. Поскольку разные ключи могут дать одинаковый индекс, unordered_map использует стратегию цепочек (chaining): внутри каждого бакета хранится связный список элементов, которые туда попали. Быстрый доступ достигается тем, что вычисление индекса бакета происходит за константу, а длины цепочек при равномерном распределении остаются небольшими.

Важно, что количество бакетов - динамическая величина, и она определяется политикой _Prime_rehash_policy. Эта политика заставляет таблицу выбирать количество бакетов не произвольным, а равным ближайшему простому числу, взятому из заранее подготовленного массива простых чисел внутри библиотеки.

## 5. Алгоритм вставки элемента

При добавлении нового элемента в *unordered_map* выполняется несколько последовательных шагов:

1. Для ключа вычисляется хеш с помощью выбранной хеш-функции.
2. Полученное хеш-значение приводится к индексу бакета с помощью операции *hash % bucket_count*.
3. Внутри выбранного бакета просматривается цепочка элементов:
    - если ключ уже присутствует, его значение обновляется
    - если ключ отсутствует, создается новый элемент, который добавляется в эту цепочку
4. После вставки контейнер проверяет текущее значение load factor. По умолчанию в реализации GCC *max_load_factor* равен 1.0.
5. Если общее количество элементов превышает допустимый порог, запускается операция *rehash*. На этом этапе таблица выбирает новый размер, который определяется политикой *_Prime_rehash_policy*. Реальный размер выбирается как ближайшее простое число, большее чем удвоенная предыдущая вместимость. После выбора нового размера все существующие элементы заново распределяются по обновленной таблице.

Благодаря тому, что операции рехеша происходят нечасто, вставка в среднем занимает O(1), хотя в момент перераспределения элементов может временно стоить O(n).

## 6. Алгоритм поиска и удаления элемента

Поиск в *unordered_map* устроен аналогично вставке:

1. Вычисляется хеш от ключа.
2. Хеш приводится к индексу бакета с помощью *hash % bucket_count*.
3. Просматривается цепочка внутри выбранного бакета. Если найден элемент с совпадающим ключом, операция считается успешной. Если цепочка закончилась, значит элемента в таблице нет.

Средняя сложность поиска и удаления составляет O(1). В худшем случае, когда все элементы попадают в один бакет, операции деградируют до O(n), но при достаточно хорошей хеш-функции такие ситуации встречаются редко.

## 7. Load factor и механизм rehash

Одним из ключевых параметров хеш-таблицы является load factor. Это отношение числа элементов к количеству бакетов. Формально:

load factor = size / bucket_count

В реализации GCC значение по умолчанию для *max_load_factor* равно 1.0. Это означает, что когда количество элементов превышает число бакетов, таблица считает себя слишком плотной и запускает операцию *rehash*.

Во время *rehash* хеш-таблица выбирает новый размер. В стандартной библиотеке GCC для этого используется политика *_Prime_rehash_policy*. Она определяет новый размер как ближайшее простое число, которое больше чем удвоенная предыдущая вместимость. Массив таких простых чисел заранее встроен в библиотеку, и таблица просто выбирает следующее подходящее значение.

После выбора нового числа бакетов происходит перераспределение всех элементов. Каждый ключ заново хешируется и попадает в свой новый бакет. Такая операция стоит O(n), но вызывается нечасто. Поэтому амортизированная сложность операций остается около O(1).

Использование простых чисел в качестве размеров таблицы уменьшает вероятность неудачного распределения при использовании операции `% bucket_count` и помогает сохранять равномерность заполнения бакетов.

## 8. Недостатки структуры unordered_map

Несмотря на высокую среднюю скорость работы, *unordered_map* имеет несколько важных ограничений:

- Порядок элементов не определен. Таблица не хранит ключи в отсортированном виде, поэтому невозможно выполнять операции, которые опираются на порядок данных.
- Производительность сильно зависит от качества хеш-функции. Если разные ключи дают одинаковые хеши, цепочки становятся длинными, и операции могут деградировать до O(n).
- Операции могут вызывать резкие скачки по стоимости. Когда происходит *rehash*, таблица на короткий момент выполняет работу порядка O(n). Это не влияет на среднюю сложность, но может быть критично для задач с жесткими ограничениями по времени.

Несмотря на эти недостатки, *unordered_map* остается одним из самых удобных и быстрых контейнеров для задач, где требуется быстрый доступ к элементам по ключу.

## Источники и полезные статьи:

std::unordered_map (cppreference.com)

Синтаксис и базовые операции unordered_map (GeeksforGeeks)

Анализ реализаций хеш-таблиц в популярных языках программирования (rcoh.me)

FNV Hash function - общее описание и примеры алгоритмов (wikipedia)

Разбор условия rehash в unordered_map (habr.com)

## Более глубокие исследования:

A Proposal to Add Hash Tables to the Standard Library

Phil Bagwell – Ideal Hash Trees: структура HAMT как альтернатива классическим хеш-таблицам

Hash Tables – Theory and Practice. Linux Journal, 2015.