Ассоциативные контейнеры

М. А. Ложников

11 марта 2019 г.

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Упорядоченные контейнеры

В данном разделе пойдёт речь об упорядоченных ассоциативных контейнерах. Особенностью данных контейнеров является логарифмическая сложность вставки, поиска и удаления элемента. В стандартной библиотеке C++ как правило для хранения элементов такие контейнеры используют красно-чёрные деревья. Красно-чёрное дерево является модификацией обычного бинарного дерева поиска, которое представляет из себя граф без циклов. Вершины графа называются узлами дерева. Каждый узел кроме единственного корневого узла имеет ровно одного родителя. Корневой узел родителя не имеет. Каждый узел бинарного дерева имеет не более двух прямых потомков, которые будем называть дочерними узлами. Узлы дерева, не имеющие потомков называются листами дерева или концевыми узлами. В узлах дерева может содержаться некоторая информация. В бинарном дереве поиска между данными, хранящимися в узле, введено отношение порядка, причём, гарантируется, что данные в левом потомке узла $\mathcal{P}$ строго меньше данных в самом узле $\mathcal{P}$, а данные, лежащие в правом потомке больше или равны данных узла $\mathcal{P}$. Отметим, что для того, чтобы алгоритмы поиска в бинарном дереве были эффективными, оно должно быть сбалансировано. Существует довольно много различных определений сбалансированности, которые можно нестрого обобщить следующим образом: дерево называется сбалансированным, если длина пути от каждого листа до корня "примерно одинакова". Узел дерева удобно представлять в следующем виде:

```
struct TreeNode {
  TreeNode* left;      // Левый потомок
  TreeNode* right;     // Правый потомок
  TreeNode* parent;    // Родитель
  Data data;           // Некоторые данные
};
```

Красно-чёрные деревья дополнительно обладают следующими свойствами: a) каждый узел дерева раскрашен либо в красный либо в чёрный цвет; b) корень — чёрный; c) листы не содержат данных и являются чёрными; d) дочерние узлы красного узла являются чёрными; e) любой путь из фиксированного узла дерева до любого листа, являющегося его потомком содержит одно и то же число (для этого фиксированного узла) чёрных узлов. Таким образом, красно-чёрные деревья являются сбалансированными.

1.1 Контейнер std::map

Шаблон `std::map`, определённый в заголовочном файле `<map>` хранит внутри себя пары ключ–значение. Он содержит два обязательных шаблонных параметра: тип данных ключа и тип данных

значения. На тип данных ключа накладывается единственное ограничение: между элементами этого типа данных должна быть определена операция “меньше” (<). Этот шаблон (также называемый словарём) позволяет записать значение по некоторому ключу, просмотреть значение по ключу, а также удалить пару с заданным ключом. Ключи в этом контейнере являются уникальными, их нельзя модифицировать. Рассмотрим следующий пример

```cpp
#include <iostream>
#include <map>
#include <string>
#include <vector>

int main() {
  std::map<int, int> intToInt;  // Объявляем словарь с парами int ---> int

  // Операция [] позволяет записать значение по ключу
  intToInt[5] = 3; // Записали значение 3 по ключу 5
  intToInt[8] = 6; // Записали значение 6 по ключу 8
  intToInt[1] = 2; // Записали значение 2 по ключу 1

  intToInt[1] = 20; // Можно поменять значение по ключу

  // Операцию [] можно использовать и для чтения
  std::cout << "Value with key 8 is equal to " << intToInt[8] << std::endl;

  /* Если пары с указанным ключом не было, то операция [] добавит эту пару в контейнер
  со значением по умолчанию для соответствующего типа данных значения. */
  intToInt[4]; // добавили пару (4, 0)

  /* Таким образом, операция [] может модифицировать контейнер, даже если используется
   для чтения. Если нужен немодифицирующий аналог, то можно использовать метод at(),
   также принимающий ключ и возвращающий значение. В случае отсутствия пары с указанным
   ключом метод at() выбросит исключение std::out_of_range. */
  std::cout << intToInt[4] << std::endl; // выведет 0

  try {
    std::cout << intToInt.at(10) << std::endl;
  }
  catch (std::out_of_range&) {
    std::cout << "Map doesn't contain a pair with key 10!" << std::endl;
  }

  /* Для поиска элемента контейнера по ключу можно использовать метод find(),
    принимающий ключ и возвращающий итератор на соответствующую пару. В случае
    отсутствия элемента с указанным ключом возвращается итератор на конец контейнера. */
  std::map<int, int>::iterator it = intToInt.find(1);
  if (it == intToInt.end()) {
    std::cout << "Map doesn't have pair with key = 1!" << std::endl;
  }
  else {
    std::cout << "intToInt[1] = " << intToInt.at(1) << std::endl;
  }
```

```
46   /* Для удаления элемента из контейнера можно использовать метод erase(),
47     принимающий ключ или итератор. */
48   intToInt.erase(5); // удалили пару с ключом 5.
49
50   /* Можно инициализировать при помощи списка в фигурных скобках. Каждая пара
51    инициализируется в отдельных скобках. */
52   std::map<std::string, int> stringToInt =
53     { {"one", 1}, {"two", 2}, {"three", 3}, {"four", 4}, {"five", 5} };
54
55   std::cout << "One is equal to " << stringToInt["one"] << std::endl;
56   stringToInt["ten"] = 10;
57
58   stringToInt.erase("one"); // удаление элемента по ключу
59   std::map<std::string, int>::iterator it2 = stringToInt.find("two");
60   stringToInt.erase(it2);   // удаление элемента по итератору
61
62   /* Для вектора определена операция "меньше", следовательно его также можно
63     использовать в качестве ключа при условии, что для элементов вектора определена
64     операция "меньше". */
65   std::map<std::vector<int>, double> vectorToDouble = {
66     {{1, 2, 3}, 3.14},                  // внутренний инициализатор для вектора
67     {{4}, 5},
68     {{1, 2, 3, 4, 5, 6}, 8.123}
69   };
70   return 0;
71 }
```

Метод `erase()` и операция `[]` для словаря не инвалидируют ссылки и итераторы (разумеется, кроме ссылок и итераторов на удалённый элемент).

Задача 1 *Напишите класс, задающий неориентированный граф с произвольным числом вершин и рёбер.*

```
1 class Graph {
2  public:
3   void AddEdge(int vertex1, int vertex2);
4   bool HasVertex(int vertex) const;
5   const std::vector<int>& ConnectionsOfVertex(int vertex) const;
6  private:
7   std::map<int, std::vector<int>> data;
8 };
```

Метод `AddEdge()` *добавляет ребро между двумя вершинами, следует учитывать, что связь двусторонняя. Если функция получила на вход две одинаковые вершины, или ребро уже присутствует в графе, то метод ничего не делает. Метод* `HasVertex()` *проверяет, присутствует ли вершина в графе. Будем считать, что граф содержит вершину, если в него добавлено ребро, соединяющее эту вершину с какой-либо другой. Метод* `ConnectionsOfVertex()` *возвращает список вершин, с которыми соединена указанная вершина. Список должен быть отсортирован в том же порядке, в котором были добавлены соответствующие рёбра. Если вершина в графе отсутствует, то следует выбросить исключение* `std::out_of_range`.

1.1.1 Итерация по контейнеру

Итератор контейнера `std::map<K,V>` с ключом типа `K` и значением типа `V` указывает на пару вида `std::pair<const K, V>`, в которой поле `first` является ключом, а `second` — значением. Итераторы указывают на пары, отсортированные в порядке возрастания по ключу. Итераторы контейнера `std::map` поддерживают только последовательный доступ к элементам контейнера.

```
std::map<std::string, int> stringToInt =
    { {"one", 1}, {"two", 2}, {"three", 3}, {"four", 4}, {"five", 5} };

std::map<std::string, int>::iterator it = stringToInt.begin();

// Пара p ссылается на второй элемент контейнера, т.е. на ("two", 3)
std::pair<const std::string, int>& p = *std::next(it);

/* Выведет пару ("one", 1). Благодаря переопределённой операции -> для итератора
  словаря it->first то же самое, что и (*it).first. */
std::cout << "Key is equal to " << it->first
          << " value is equal to " << it->second << std::endl;

// Выведем все элементы контейнера в порядке возрастания ключа
using It = std::map<std::string, int>::iterator; // Введём для краткости
for (It it = stringToInt.begin(); it != stringToInt.end(); ++it) {
  std::cout << it->first << ": " << it->second << std::endl;
}

/* То же самое при помощи цикла range-based for. Первый элемент пары константный,
  поскольку итерируемся по ссылке и ключ нельзя модифицировать. */
for (std::pair<const std::string, int>& elem : stringToInt) {
  std::cout << elem.first << ": " << elem.second << std::endl;
}

// Можно ещё подсократить
for (auto& elem : stringToInt) {
  std::cout << elem.first << ": " << elem.second << std::endl;
}

/* В C++17 появилась удобная возможность избавиться от пары. Называется
  structured bindings, доступна при компиляции с ключом -std=c++17 в компиляторе
  g++ начиная с версии 7. */
for (auto& [key, value] : stringToInt) {
  /* В key лежит ссылка на ключ, в value ссылка на значение. Имена переменных,
    разумеется, могут быть любыми. */
  std::cout << key << ": " << value << std::endl;
}
```

Задача 2 *Напишите класс для учёта населения городов.*

```
class CityPopulation {
 public:
```

```
  void AddCity(const std::string& name, int population);
  int GetPopulation(const std::string& name) const;
  void Print(std::ostream& stream) const;
 private:
  std::map<std::string, int> data;
};
```

Метод `AddCity()` *добавляет город с указанным населением. Если город присутствует в базе, то его население обновляется. Метод* `GetPopulation()` *возвращает число жителей города. В случае отсутствия города в базе следует выбросить исключение* `std::out_of_range`*. Метод* `Print()` *выводит в поток список городов с их населением в формате*

```
Город1 население1
Город2 население2
...
ГородN населениеN
```

в алфавитном порядке по названию города. Гарантируется, что все названия городов уникальные и содержат только латинские буквы без пробелов.

1.1.2 Пользовательские ключи для контейнера

Если необходимо чтобы ключом словаря являлся пользовательский тип данных, то для него необходимо определить операцию сравнения “меньше” (<).

```
#include <iostream>
#include <map>

struct Point {
  int x;
  int y;
};

bool operator<(const Point& lhs, const Point& rhs) {
  if (lhs.x < rhs.x)
    return true;

  if (lhs.x == rhs.x && lhs.y < rhs.y)
    return true;

  return false;
}

int main() {
  std::map<Point, double> pointToDouble = {
      {{1, 20}, 401.0},    // Как обычно перечисляем пары (ключ, значение)
      {{2, 3}, 13.0},      // Внутренний инициализатор для структуры
      {{4, 5}, 41.0},
      {{1, 2}, 5.0}
  };
```

```
26
27  for (auto& elem : pointToDouble) {
28    Point pt = elem.first;
29    double val = elem.second;
30    std::cout << "Point (" << pt.x << ", " << pt.y << ") value "
31              << val << std::endl;
32  }
33  return 0;
34 }
```

Задача 3 *Напишите класс для учёта расходов и доходов.*

```
1  struct Date {int year; int month; int day; };
2
3  class ProfitCounter {
4   public:
5    void AddRecord(const Date& date, int value);
6    std::int64_t TotalProfit(const Date& from, const Date& to) const;
7   private:
8    std::map<Date, int> data;
9  };
```

Метод `AddRecord()` *добавляет запись о том, что в указанный день баланс на счёте изменился на указанную сумму. В один и тот же день может быть несколько транзакций, в таком случае результат суммируется. Метод* `TotalProfit()` *возвращает прибыль (доходы минус расходы), полученную за указанный промежуток, обе даты включаются в промежуток. Если дата* `to` *стоит в календаре раньше даты* `from`*, то следует выбросить исключение* `std::invalid_argument`*. Информация о доходах и расходах идёт вперемешку с запросами.*

Для структур `Date` *необходимо написать операцию сравнения "меньше", сравнивающую даты в календарном порядке.*

Тип данных `std::int64_t`*, объявленный в заголовочном файле* `<cstdint>` *представляет из себя 64-битное знаковое целое число.*

Указание. *Для решения задачи могут понадобиться методы* `lower_bound()` *и* `upper_bound()` *шаблона* `std::map`*, возвращающие итератор на первый элемент, ключ которого не меньше указанного ключа и итератор на первый элемент, ключ которого больше указанного ключа.*

1.2 Контейнер std::set

Контейнер `std::set`, определённый в заголовочном файле `<set>` является упрощённым вариантом словаря. Контейнер `std::set` (иногда называемый множеством) отличается от словаря тем, что не хранит значения, а только набор уникальных ключей. Поэтому итераторы контейнера `std::set` указывают на ключ, а не на пару. Сами ключи нельзя модифицировать, поэтому `iterator` совпадает с `const_iterator`. Для ключей должна быть определена операция сравнения "меньше". Для добавления элементов используется метод `insert()`, операция `[]` для множества не определена. В остальном интерфейс множества похож на интерфейс словаря.

```
1 #include <iostream>
2 #include <set>
3
```

```cpp
int main() {
  // Можно инициализировать при помощи списка в фигурных скобках
  std::set<int> setOfInts = {1, 3, 5, 7, 9};

  setOfInts.insert(4); // вставка элемента в множество
  std::set<int>::iterator it = setOfInts.find(3);
  if (it == setOfInts.end())
    std::cout << "Can't find key 3" << std::endl;
  else
    std::cout << "Found key 3" << std::endl;

  setOfInts.erase(1);   // Удаление элемента с ключом 1
  setOfInts.erase(it);  // Удаление элемента по итератору

  // Вывод в порядке возрастания элементов
  for (std::set<int>::iterator it = setOfInts.begin(); it != setOfInts.end(); ++it) {
    std::cout << "Elem is equal to " << *it << std::endl;
  }

  /* Можно проще. В данном случае итерируемся просто по значению, поскольку
    скопировать число типа int проще, чем прочитать по ссылке. */
  for (int elem : setOfInts)
    std::cout << "Elem is equal to " << elem << std::endl;

  /* Если ключ долго копируется (для int не актуально), то можно проитерироваться
    по константной ссылке. */
  for (const int& elem : setOfInts)
    std::cout << "Elem is equal to " << elem << std::endl;

  // Или так. Тип данных elem есть const int&.
  for (auto& elem : setOfInts)
    std::cout << "Elem is equal to " << elem << std::endl;
  return 0;
}
```

Операции вставки и добавления элемента не инвалидируют ссылки и итераторы.

Задача 4 *Напишите класс, задающий ориентированный граф с произвольным числом вершин и рёбер.*

```cpp
class OrientedGraph {
 public:
  void AddVertex(int vertex);
  void AddEdge(int vertex1, int vertex2);
  bool HasVertex(int vertex) const;
  bool HasEdge(int vertex1, int vertex2) const;
  void RemoveVertex(int vertex);
  void RemoveEdge(int vertex1, int vertex2);
  const std::set<int>& ConnectionsOfVertex(int vertex) const;

```

```
11 private:
12   std::map<int, std::set<int>> data;
13   std::set<int> vertices;
14 };
```

- *Метод* `AddVertex()` *добавляет вершину в граф.*
- *Метод* `AddEdge()` *добавляет ребро, направленное от первой вершины ко второй. Если ребро уже присутствует в графе, то метод ничего не делает. Если одной из вершин, которые соединяет ребро, нет в графе, то метод генерирует исключение* `std::out_of_range`.
- *Метод* `HasVertex()` *проверяет, есть ли вершина в графе.*
- *Метод* `HasEdge()` *проверяет есть ли в графе ребро, направленное от первой вершины до второй.*
- *Метод* `RemoveVertex()` *удаляет вершину из графа, а также все рёбра, выходящие из неё или входящие в неё. В случае отсутствия вершины в графе метод ничего не делает.*
- *Метод* `RemoveEdge()` *удаляет из графа ребро, направленное от первого аргумента ко второму. В случае отсутствия указанного ребра в графе метод ничего не делает.*
- *Метод* `ConnectionsOfVertex()` *возвращает список вершин, в которые можно прийти из указанной вершины пройдя ровно по одному ребру. Список должен быть отсортирован в порядке возрастания вершин. В случае отсутствия соответствующей вершины в графе следует выбросить исключение* `std::out_of_range`.

2 Неупорядоченные контейнеры

В данном разделе обсудим контейнеры стандартной библиотеки, построенные на основе хэш-таблиц. Отличительной особенностью этих контейнеров является добавление, вставка и удаление элемента, выполняющиеся в среднем за $O(1)$. Взамен приходится платить потерей упорядоченности элементов.

Опишем вкратце принцип работы хэш-таблиц. Предположим, что для каждого элемента, который можно положить в таблицу, определена некоторая функция (называемая хеш-функцией), ставящая ему в соответствие некоторое целое число. Как правило, хэш-таблица состоит из набора корзин (называемых bucket'ами), в которых лежат элементы. Распределение элементов по корзинам определяется хэш-функией, номер корзины, в которую следует положить элемент, можно определить, например, взяв остаток от деления значения хэш-функции на число корзин. После определения номера корзины её содержимое проверяется методом последовательного поиска. Таким образом, если в таблице N элементов и K корзин, операции добавления, поиска и удаления элемента имеют сложность $O(N/K)$ при условии, что элементы распределены по корзинам равномерно. Следовательно, для эффективной работы хэш-таблицы необходимо использовать "хорошие" хэш-функции. Если число элементов в корзине превышает некоторое критическое значение, то для эффективной работы таблицы необходимо увеличить число корзин и произвести рехэширование. Таким образом, сложность операций добавления, поиска и удаления элемента для контейнеров, определённых ниже равна $O(1)$ при условии, что операция не вызывает рехэшрования, сложность которого равна $O(N)$.

2.1 Контейнер std::unordered_map

Контейнер `std::unordered_map` представляет из себя словарь на базе хэш-таблиц. То есть он хранит пары ключ–значение с уникальными ключами. Для ключа должна быть определена хэш-функция и операция сравнения на равенство (==). Ключи нельзя модифицировать. Для стандартных типов

хэш-функция определена. Интерфейс контейнера во многом похож на интерфейс словаря на основе деревьев, поэтому не будем подробно его разбирать. Итераторы контейнера поддерживают только прямой последовательный доступ, то есть обход производится только в прямом направлении.

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

int main() {
  std::unordered_map<std::string, int> stringToInt = {
    {"one", 1},
    {"two", 2},
    {"three", 3},
    {"four", 4},
    {"five", 5}
  };

  stringToInt["ten"] = 10; // добавление пары с ключом "ten" и значением 10
  stringToInt.erase("three"); // удаление элемента с ключом "three"

  // Поиск элемента
  std::unordered_map<std::string, int>::iterator it = stringToInt.find("four");

  if (it == stringToInt.end())
    std::cout << "Can't find element with key 'four'" << std::endl;
  else
    std::cout << "Found element with key 'four'" << std::endl;

  stringToInt.erase(it); // Удаление элемента по итератору.

  using It = std::unordered_map<std::string, int>::iterator; // Для краткости

  // Неупорядоченный вывод элементов контейнера.
  for (It it = stringToInt.begin(); it != stringToInt.end(); ++it) {
    // Здесь всё так же, как и с обычными словарями.
    std::cout << it->first << ": " << it->second << std::endl;
  }

  /* Первый элемент пары константный поскольку итерируемся по ссылке и ключ нельзя
    модифицировать. Точно так же, как и для контейнера std::map в C++17 можно
    использовать structured bindings. */
  for (std::pair<const std::string, int>& p : stringToInt)
    std::cout << p.first << ": " << p.second << std::endl;

  return 0;
}
```

Операция `[]` инвалидирует итераторы при рехешировании, ссылки не инвалидируются. Метод `erase()` не инвалидирует ссылки и итераторы (кроме таковых для удаляемого элемента).

Задача 5 *Напишите класс для учёта сотрудников предприятия.*

```cpp
struct Employee {
  std::string name;
  int age;
};

class JobManager {
 public:
  void AddEmployee(const Employee& employee, const std::string& job);
  double AverageAge(const std::string& job) const;
  void PrintJobEmployees(const std::string& job, std::ostream& stream) const;
 private:
  std::unordered_map<std::string, std::set<Employee>> data;
};
```

Метод `AddEmployee()` *добавляет сотрудника с указанной должностью. Если сотрудник с указанным именем уже занимает данную должность, то следует выдать исключение* `std::runtime_error`. *Метод* `AverageAge()` *считает средний возраст сотрудников, занимающих данную должность. Если указанной должности в базе нет, то следует выбросить исключение* `std::out_of_range`. *Метод* `PrintJobEmployees()` *выводит в поток список сотрудников, занимающих указанную должность, в формате*

```
Имя1 возраст1
Имя2 возраст2
...
ИмяN возрастN
```

Список должен быть отсортирован в алфавитном порядке по имени сотрудника. Если указанной должности в базе нет, то метод ничего не делает.

2.1.1 Пользовательские типы данных для контейнера std::unordered_map

Для того, чтобы использовать в качестве ключа пользовательский тип данных, нужно определить для этого типа данных операцию сравнения на равенство и хэш-функцию.

Хэш-функцию можно определить двумя способами. По-умолчанию компилятор использует константную операцию `()` (`std::size_t operator()(const K& key) const`) структуры `std::hash<K>` для подсчёта хэш-кодов объектов типа данных `K`. Эта операция принимает ссылку на ключ и возвращает его хэш-код. Первый способ заключается в определении соответствующей специализации шаблона `std::hash` для нашего типа данных. Второй способ заключается в написании структуры или класса, в котором определена публичная константная операция `()`, принимающая ключ по константной ссылке и возвращающая `std::size_t`. Затем имя этого класса или структуры (типа данных, а не объекта) подставляется в качестве третьего шаблонного параметра шаблона `std::unordered_map`, который для класса `std::unordered_map<K, V>` по умолчанию равен `std::hash<K>`.

Приведём в качестве примера оба варианта. Первый вариант.

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

struct Point {
  int x;
```

```
  int y;
};

// Определяем специализацию std::hash в пространстве имён std
namespace std {
  template<>
  struct hash<Point> {
    size_t operator()(const Point& point) const {
      const std::size_t p = 40319;
      return point.x * p + point.y;
    }
  };
}

// Определяем операцию сравнения на равенство
bool operator==(const Point& lhs, const Point& rhs) {
  return lhs.x == rhs.x && lhs.y == rhs.y;
}

int main() {
  std::unordered_map<Point, double> pointToDouble = {
    {{1, 20}, 401.0},    // Как обычно перечисляем пары (ключ, значение)
    {{2, 3}, 13.0},      // Внутренний инициализатор для структуры
    {{4, 5}, 41.0},
    {{1, 2}, 5.0}
  };

  for (auto& elem : pointToDouble) {
    Point pt = elem.first;
    double val = elem.second;
    std::cout << "Point (" << pt.x << ", " << pt.y << ") value "
    << val << std::endl;
  }
  return 0;
}
```

В примере выше используется функция хэширования, соответствующая переводу числа из системы исчисления по основанию 40319 (некоторое простое число, равное $8! - 1$) в десятичную систему исчисления. Таким образом, если элементы структуры `Point` неотрицательны и меньше `40319`, то хэш-функция выдаст уникальное число для каждого элемента.

Если ключ содержит несколько элементов с хэшами $h_0, h_1, \ldots, h_n$, то в качестве хэш-кода ключа можно использовать значение многочлена

$$h_0 + h_1 p + h_2 p^2 + \ldots + h_n p^n,$$

где p —достаточно большое простое число. Простые числа можно брать по формуле $n! - 1$ для любого n. Переполнения целочисленного типа данных при расчётах можно не обрабатывать, поскольку в результате всё равно будет посчитан остаток от деления на максимально возможное число.

Рассмотрим второй пример.

```
#include <iostream>
#include <string>
```

```cpp
#include <unordered_map>

struct Student {
  std::string name;
  std::string surname;
  unsigned age;
};

// Определяем операцию сравнения на равенство.
bool operator==(const Student& lhs, const Student& rhs) {
  return lhs.name == rhs.name && lhs.surname == rhs.surname && lhs.age == rhs.age;
}

// Определяем структуру с операцией () для хэширования.
struct StudentHasher {
  std::size_t operator()(const Student& student) const {
    std::hash<std::string> stringHasher; // Объект для хэширования строк
    std::hash<unsigned> unsignedHasher;  // Объект для хеширования беззнаковых чисел
    const std::size_t p = 3628799;       // Некоторое простое число (10! - 1)

    return stringHasher(student.name) +
           p * stringHasher(student.surname) +
           p * p * unsignedHasher(student.age);
  }
};

int main() {
  // Третьим шаблонным параметром указываем структуру, задающую хэш-функцию для ключа.
  std::unordered_map<Student, int, StudentHasher> db = {
    {{"Ivan", "Petrov", 20}, 4},
    {{"Petr", "Ivanov", 18}, 3},
    {{"Ivan", "Sidorov", 21}, 2},
    {{"Petr", "Sidorov", 19}, 1}
  };

  for (std::pair<const Student, int>& p : db) {
    const Student& student = p.first;
    int numPassedExams = p.second;

    std::cout << "Student " << student.name << " "
              << student.surname << ", " << student.age << " years old "
              << " passed " << numPassedExams << " exams." << std::endl;
  }
  return 0;
}
```

Структуры, используемые для хэширования достаточно легковесны, их можно объявлять прямо в функции хеширования, никаких накладных расходов на вызов конструктора и деструктора при этом возникать не должно. Для хэширования целого числа можно было использовать само число.

2.2 Контейнер std::unordered_set

Контейнер `unordered_set`, определённый в заголовочном файле `<unordered_set>`, является упрощённой версией контейнера `std::unordered_map`. Он не хранит значений, только уникальный набор ключей. Точно так же, как и в контейнере `std::unordered_map` для ключей должна быть определена операция сравнения на равенство (==), а также хэш-функция. Его интерфейс похож на интерфейс контейнера `std::set`.

```cpp
#include <iostream>
#include <unordered_set>

int main() {
  // Можно инициализировать при помощи списка в фигурных скобках
  std::unordered_set<int> setOfInts = {1, 3, 5, 7, 9};

  setOfInts.insert(4); // вставка элемента в множество
  std::unordered_set<int>::iterator it = setOfInts.find(3);
  if (it == setOfInts.end())
    std::cout << "Can't find key 3" << std::endl;
  else
    std::cout << "Found key 3" << std::endl;

  setOfInts.erase(1);   // Удаление элемента с ключом 1
  setOfInts.erase(it);  // Удаление элемента по итератору

  using It = std::unordered_set<int>::iterator; // Для краткости
  // Вывод в порядке возрастания элементов
  for (It it = setOfInts.begin(); it != setOfInts.end(); ++it) {
    std::cout << "Elem is equal to " << *it << std::endl;
  }

  /* Можно проще. В данном случае итерируемся просто по значению, поскольку
  скопировать число типа int проще, чем прочитать по ссылке. */
  for (int elem : setOfInts)
    std::cout << "Elem is equal to " << elem << std::endl;

  /* Если ключ долго копируется (для int не актуально), то можно проитерироваться
    по константной ссылке. */
  for (const int& elem : setOfInts)
    std::cout << "Elem is equal to " << elem << std::endl;

  // Или так. Тип данных elem есть const int&.
  for (auto& elem : setOfInts)
    std::cout << "Elem is equal to " << elem << std::endl;
  return 0;
}
```

Точно так же, как и в контейнере `std::unordered_map`, операция вставки не инвалидирует итераторы при условии, что не произошло рехэширование, ссылки не инвалидируются. При удалении элемента ссылки и итераторы на все элементы за исключением удаляемого не инвалидируются.

3 Другие ассоциативные контейнеры

До сих пор речь шла о контейнерах, хранящих элементы с уникальным ключом. В стандартной библиотеке также есть классы, позволяющие хранить элементы с повторяющимися ключами, а именно, `std::multimap`, `std::multiset`, `std::unordered_multimap`, `std::unordered_multiset`.

Список литературы

[1] `https://en.cppreference.com/w/cpp/container`