Лабораторная работа 5. STL: контейнеры vector, set и map

1. Цель работы

Ознакомиться с основными возможностями контейнера `std::vector`, выполнить оценку производительности основных методов контейнера, сравнить скорость выполнения основных операций `std::vector` с другими аналогичными структурами данных.

Ознакомиться с основными возможностями контейнеров `std::set` и `std::map`, выполнить оценку производительности основных методов контейнеров, сравнить скорость выполнения основных операций `std::set` и `std::map` с другими аналогичными структурами данных.

2. Краткие теоретические положения

В стандартной библиотеке C++ *вектором* (std::vector) называется динамический массив, обеспечивающий быстрое добавление новых элементов в конец и меняющий свой размер при необходимости. Вектор гарантирует отсутствие утечек памяти. Для работы с вектором нужно подключить заголовочный файл vector. Элементы вектора должны быть одинакового типа, и этот тип должен быть известен при компиляции программы.

Рассмотрим пример программы, которая заполняет вектор элементами и печатает их через пробел:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> data = {1, 2, 3, 4, 5};
   for (int elem : data) {
     std::cout << elem << " ";
   }
   std::cout << "\n";
}
```

Здесь мы инициализируем вектор через список инициализации, в котором элементы перечислены через запятую. Другой способ инициализации вектора — указать число элементов и (при необходимости) образец элемента:

```
#include <string>
#include <vector>
int main() {
   std::vector<std::string> v1; // пустой вектор строк
   std::vector<std::string> v2(5); // вектор из пяти пустых
строк
```

```
    std::vector<std::string> v3(5, "hello"); // вектор из пяти
строк "hello"
}
```

Обращение к элементам

Выше мы использовали для печати элементов вектора цикл `range-for`. Но иногда удобнее работать с индексами. Вектор хранит элементы в памяти последовательно, поэтому по индексу элемента можно быстро найти его положение в памяти. Индексация начинается с нуля:

```
std::vector<int> data = {1, 2, 3, 4, 5};
int a = data[0]; // начальный элемент вектора
int b = data[4]; // последний элемент вектора (в нём пять
элементов)
data[2] = -3; // меняем элемент 3 на -3
```

Чтобы узнать общее количество элементов в векторе, можно воспользоваться функцией `size`:

```
std::cout << data.size() << "\n";
```

Отрицательные индексы, как в некоторых других языках программирования, не допускаются.

Обратите внимание: когда мы обращаемся по индексу через квадратные скобки, проверки его корректности не происходит. Это ещё одно проявление принципа *«мы не должны платить за то, что не используем»*.

Встроенные валидаторы замедляют программу: предполагается, что программист пишет правильный код и уверен, что индекс i в выражении `data[i]` неотрицателен и удовлетворяет условию `i < data.size()`. В этом случае они ему не нужны.

Если всё же обратиться к вектору по некорректному индексу, то программа во время выполнения попадёт в неопределённое поведение: фактически она попробует прочитать память, не принадлежащую вектору.

Если вам не хочется делать много лишних проверок, а в корректности индекса вы не уверены, то можно использовать функцию [at](at):

```
std::vector<int> data = {1, 2, 3, 4, 5};
std::cout << data[42] << "\n"; // неопределённое поведение:
может произойти всё что угодно
std::cout << data.at(0) << "\n"; // напечатается 1
std::cout << data.at(42) << "\n"; // произойдёт исключение
std::out_of_range — его можно будет перехватить и обработать
```

Рассмотрим функции вектора `front` и back, которые возвращают его первый и последний элемент без использования индексов:

```
std::vector<int> data = {1, 2, 3, 4, 5};
std::cout << data.front() << "\n"; // то же, что data[0]
```

```
std::cout << data.back() << "\n"; // то же, что
data[data.size() - 1]
```

Важно учитывать, что вызов этих функций на пустом векторе приведёт к неопределённому поведению.

Для проверки вектора на пустоту вместо сравнения `data.size() == 0` принято использовать функцию `empty`, которая возвращает логическое значение:

```
if (!data.empty()) {
// вектор не пуст, с ним можно работать
}
```

Итерация по индексам

Так сложилось, что в стандартной библиотеке индексы и размеры контейнеров имеют беззнаковый тип. Вместо `unsigned int` или `unsigned long int` для него используется традиционный псевдоним `size_t` (а точнее, `std::vector<T>::size_type`). Тип `size_t` на самом деле совпадает с `uint32_t` или `uint64_t` в зависимости от битности платформы. Его использование в программе дополнительно подчёркивает, что мы имеем дело с индексами или с размером.

Итерацию по элементам `data` с помощью индексов можно записать так:

```
for (size_t i = 0; i != data.size(); ++i) {
   std::cout << data[i] << " ";
}
```

Во многих источниках такая форма записи считается канонической формой записи такого цикла: в ней принято использовать сравнение `!=` и префиксный `++i`.

Беззнаковость типа возвращаемого значения функции `size` порождает следующую проблему. По правилам, унаследованным ещё от языка C, результат арифметических действий над беззнаковым и знаковым типами приводится к беззнаковому типу. Поэтому выражение `data.size() - 1`, например, тоже будет беззнаковым. Если data.size() окажется нулём, то такое выражение будет вовсе не минус единицей, а самым большим беззнаковым целым (для 64-битной платформы это $2^{64}-1$).

Рассмотрим следующий ошибочный код, который проверяет, есть ли в векторе дубликаты, идущие подряд:

```
// итерация по всем элементам, кроме последнего:
for (size_t i = 0; i < data.size() - 1; ++i) {
    if (data[i] == data[i + 1]) {
       std::cout << "Duplicate value: " << data[i] << "\n";
   }
}
```

Эта программа будет некорректно работать на пустом векторе. Условие `i < data.size() - 1` на первой итерации окажется истинным, и произойдёт обращение к элементам пустого вектора. Правильнее было бы переписать это условие через `i + 1 < data.size()` или воспользоваться внешней функцией `std::ssize`, которая появилась в C++20. Она возвращает знаковый размер вектора:

```
for (std::int64_t i = 0; i < std::ssize(data) - 1; ++i) {
   if (data[i] == data[i + 1]) {
      std::cout << "Duplicate value: " << data[i] << "\n";
   }
}
```

Добавление и удаление элементов

В вектор можно эффективно добавлять элементы в конец и удалять их с конца. Для этого существуют функции `push_back` и `pop_back`. Рассмотрим программу, считывающую числа с клавиатуры в вектор и затем удаляющую все нули в конце:

```
#include <iostream>
#include <vector>
int main() {
   int x;
   std::vector<int> data;
   while (std::cin >> x) { // читаем числа, пока не закончится
ввод
   data.push_back(x); // добавляем очередное число в вектор
}
   while (!data.empty() && data.back() == 0) {
     // Пока вектор не пуст и последний элемент равен нулю
      data.pop_back(); // удаляем этот нулевой элемент
   }
}
```

Добавление элементов в другие части вектора или их удаление неэффективно, так как требует сдвига соседних элементов. Поэтому отдельных функций, например, для добавления или удаления элементов из начала у вектора нет. Это можно сделать с помощью общих функций `insert/erase` и итераторов.

Удалить все элементы из вектора можно с помощью функции `clear`.

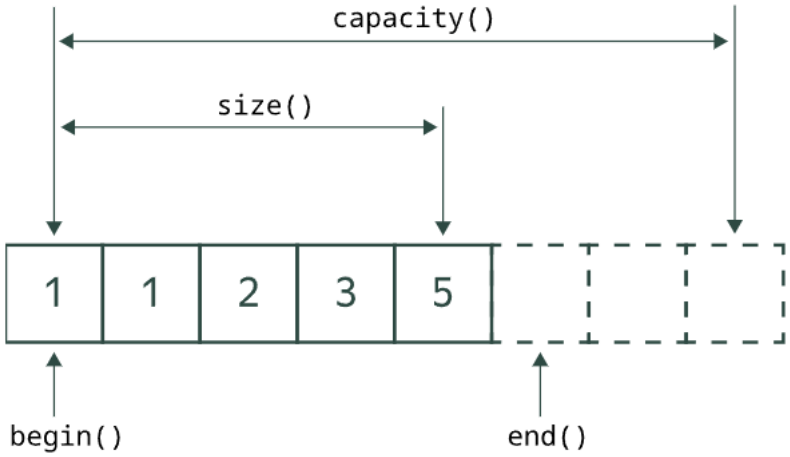
Резерв памяти

Вектор хранит элементы в памяти в виде непрерывной последовательности, друг за другом. При этом в конце последовательности резервируется дополнительное место для быстрого добавления новых элементов. Когда этот резерв

заканчивается, при вставке очередного элемента происходит *реаллокация*: элементы вектора копируются в новый, более просторный блок памяти.

Реаллокация — довольно дорогая процедура, но если она происходит достаточно редко, то её влияние незначительно. Можно доказать, что если размер нового блока выбирать в два раза больше предыдущего размера, то амортизационная сложность добавления элемента будет константной.

Текущий резерв вектора можно узнать с помощью функции `capacity` (не путайте её с функцией `size`).



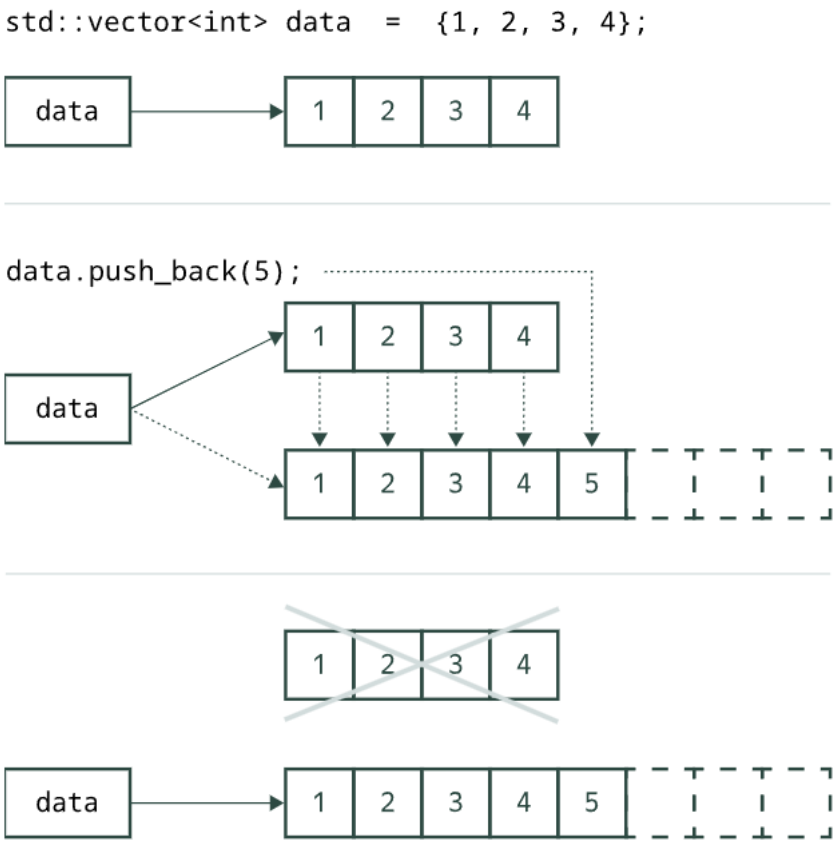
Рассмотрим программу, в которой в вектор последовательно добавляются элементы и после каждого шага печатается размер и резерв:

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> data = {1, 2};
   std::cout << data.size() << "\t" << data.capacity() << "\n";
   data.push_back(3);
   std::cout << data.size() << "\t" << data.capacity() << "\n";
   data.push_back(4);
   std::cout << data.size() << "\t" << data.capacity() << "\n";
   data.push_back(5);
   std::cout << data.size() << "\t" << data.capacity() << "\n";
}
```

Вот вывод этой программы:

```
2	2
3	4
4	4
5	8
```

Видно, что размер вектора увеличивается на единицу, а резерв удваивается после исчерпания. Так, при добавлении четвёрки используется имеющаяся в резерве память, а при добавлении тройки и пятёрки происходит *реаллокация*.



Иногда требуется заполнить вектор элементами, причём число элементов известно заранее. В таком случае можно сразу зарезервировать нужный размер памяти с помощью функции `reserve`, чтобы при добавлении элементов не происходили реаллокации. Пусть, например, нам сначала задаётся число слов, а потом сами эти слова, и нам требуется сложить их в вектор:

```
#include <iostream>
#include <string>
#include <vector>
int main() {
   std::vector<std::string> words;
   size_t words_count;
   std::cin >> words_count;
   // Размер вектора остаётся нулевым, меняется только резерв:
   words.reserve(words_count);
   for (size_t i = 0; i != words_count; ++i) {
      std::string word;
```

```
        std::cin >> word;
        // Все добавления будут дешёвыми, без реаллокаций:
        words.push_back(word);
    }
}
```

Если передать в `reserve` величину меньше текущего резерва, то ничего не поменяется — резерв останется прежним.

Функцию `reserve` не следует путать с функцией `resize`, которая меняет количество элементов в векторе. Если аргумент функции `resize` меньше текущего размера, то лишние элементы в конце вектора удаляются. Если же он больше текущего размера, то при необходимости происходит реаллокация и в вектор добавляются новые элементы с дефолтным значением данного типа.

```
#include <vector>
int main() {
    std::vector<int> data = {1, 2, 3, 4, 5};
    data.reserve(10); // поменяли резерв, но размер вектора остался равным пяти
    data.resize(3); // удалили последние элементы 4 и 5
    data.resize(6); // получили вектор 1, 2, 3, 0, 0, 0
}
```

Многомерные векторы

Воспользуемся вектором векторов, чтобы сохранить матрицу (таблицу) целых чисел. Пусть на вход программы сначала поступают число строк и число столбцов матрицы, а потом — сами элементы построчно:

```
#include <iostream>
#include <vector>
int main() {
    size_t m, n;
    std::cin >> m >> n; // число строк и столбцов
// создаём матрицу matrix из m строк, каждая из которых — вектор из n нулей
    std::vector<std::vector<int>> matrix(m, std::vector<int>(n));
    for (size_t i = 0; i != m; ++i) {
        for (size_t j = 0; j != n; ++j) {
        std::cin >> matrix[i][j];
        }
    }
// напечатаем матрицу, выводя элементы через табуляцию
for (size_t i = 0; i != m; ++i) {
```

```
        for (size_t j = 0; j != n; ++j) {
            std::cout << matrix[i][j] << "\t";
        }
        std::cout << "\n";
      }
    }
```

В этом примере мы заранее создали матрицу из нулей, а потом просто меняли её элементы.

Сортировка вектора

Рассмотрим типичную задачу — отсортировать вектор по возрастанию. Для этого в стандартной библиотеке в заголовочном файле `algorithm` есть готовая функция `sort`. Гарантируется, что сложность её работы в худшем случае составляет O(n log n), где n — число элементов в векторе.

```
#include <algorithm>
#include <vector>
int main() {
    std::vector<int> data = {3, 1, 4, 1, 5, 9, 2, 6};
    // Сортировка диапазона вектора от начала до конца
    std::sort(data.begin(), data.end());
    // получим вектор 1, 1, 2, 3, 4, 5, 6, 9
}
```

В функцию `sort` передаются так называемые *итераторы*, ограничивающие рассматриваемый диапазон. В нашем случае мы передаём диапазон, совпадающий со всем вектором, от начала до конца. Соответствующие итераторы возвращают функции `begin` и `end` (не путать с `front` и `back`!). Итераторы можно считать обобщёнными индексами (но они могут быть и у контейнеров, не допускающих обычную индексацию).

Для сортировки по убыванию можно передать на вход *обратные итераторы* `rbegin()` и `rend()`, представляющие элементы вектора в перевёрнутом порядке:

```
std::sort(data.rbegin(), data.rend()); // 9, 6, 5, 4, 3, 2, 1, 1
```

В C++20 доступен более элегантный способ сортировки через `std::ranges::sort`:

```
#include <algorithm>
#include <vector>
int main() {
    std::vector<int> data = {3, 1, 4, 1, 5, 9, 2, 6};
    std::ranges::sort(data); //можно передать сам вектор, а не его
```

```
диапазоны
}
```

Для сортировки по умолчанию используется сравнение элементов с помощью оператора <. Этот оператор работает и для самих векторов: они сравниваются лексикографически. Поэтому можно без проблем отсортировать, например, строки в матрице (векторе векторов целых чисел).

В стандартной библиотеке STL есть ассоциативные контейнеры, основанные на сбалансированных деревьях поиска (map, set) и контейнеры, основанные на хеш-таблицах (unordered_map, unordered_set). В этих контейнерах ключи уникальны, то есть, не могут повторяться. Также существуют и multi-версии этих контейнеров, в которых допускаются повторы ключей.

Так как C++ — статически типизированный язык, типы ключей и значений должны быть строго зафиксированы на этапе компиляции.

Контейнер **std::map**

Начнём с контейнера std::map. Он определен в заголовочном файле map Аналогично вектору, std::map является шаблонным: в угловых скобках нужно указать типы ключей и значений.

Рассмотрим пример:

```
#include <iostream>
#include <map>
#include <string>
int main() {
// инициализируем map набором пар {ключ, значение}
std::map<std::string, int> years = {
{"Moscow", 1147},
{"Rome", -753},
{"London", 47},
};
for (const auto& [city, year] : years) {
std::cout << city << ": " << year << "\n";
}
}
```

Вывод программы:

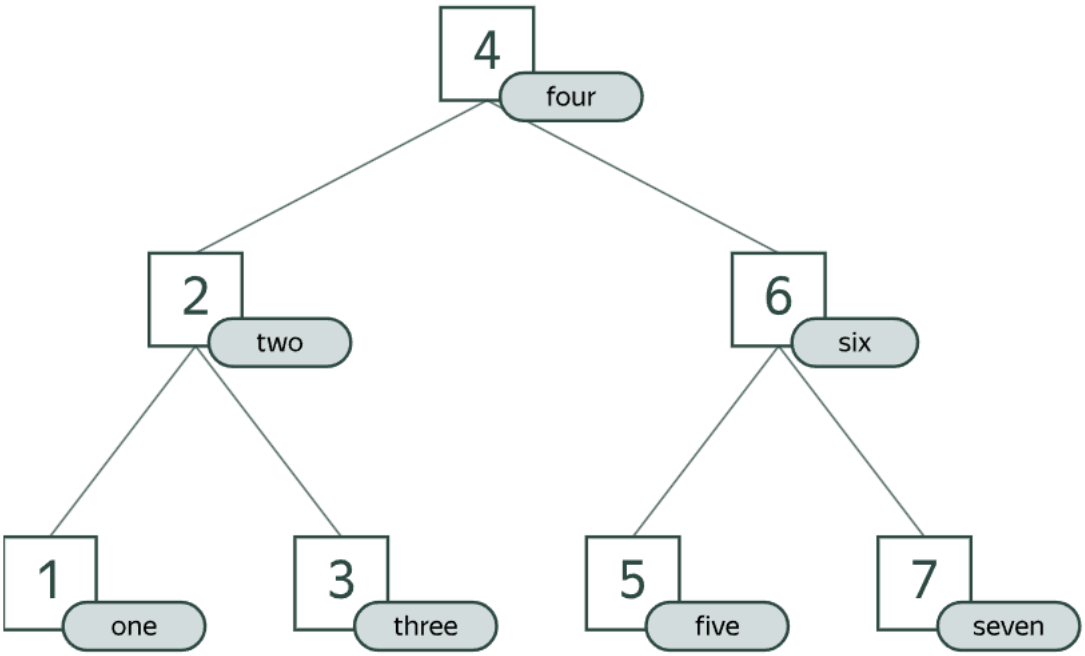
```
London: 47
Moscow: 1147
Rome: -753
```

При итерации с помощью ranged-based for возвращаются пары std::pair из константного ключа и значения. Для итерации по элементам мы использовали structured binding, прикрепив ссылки city и year к элементам

возвращаемой пары, а также auto для автоматического вывода типа. Согласитесь, это удобнее, чем такая форма записи:

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

Контейнер map реализован как красно-чёрное дерево — сбалансированное дерево поиска с особыми свойствами. Поэтому его элементы при итерации обходятся в порядке возрастания ключей, а на самих ключах должен быть определён оператор < для сравнения.



Три основных операции — поиск, вставка и удаление элемента — в таких структурах данных выполняются за *логарифмическое время* ($O(\log N)$) от числа элементов в контейнере. Покажем, как воспользоваться этими операциями.

```
#include <iostream>
#include <map>
#include <string>
int main() {
   std::map<std::string, int> data;
   std::string key;
   int value;
   while (std::cin >> key >> value) {
      data[key] = value; // вставка
   }
   data.erase("hello"); // удаление
```

```
// поиск
   if (auto iter = data.find("test"); iter != data.end()) {
   std::cout << "Found the key " << iter->first << " with the
value " << iter->second << "\n";
   } else {
      std::cout << "Not found\n";
     }
}
```

Рассмотрим эту программу подробнее.

Для вставки мы использовали обращение по ключу в квадратных скобках: data[key] = value. В отличие от вектора или дека, ключ теперь не обязательно является индексом: в нашем случае это строка. Альтернативные способы вставки — data.insert({key, value}) или data.insert_or_assign({key, value}).

Эти функции принимают пару из ключа и значения, поэтому нам пришлось обрамить в фигурные скобки key и value, чтобы экземпляр std::pair сконструировался на лету. Если ключ key уже существует в контейнере, то data[key] = value и функция insert_or_assign его перезапишут, а insert — нет (но вернет информацию о старом значении).

Удаляя элемент по ключу, можно не заботиться о его наличии в контейнере: если ключа нет, то функция erase просто ничего не поменяет.

Для поиска элемента мы вызываем функцию find, которая возвращает итератор. Мы пользуемся версией if с инициализатором, чтобы сразу сохранить этот итератор в переменную iter и потом проверить его значение. Такая переменная будет видна только внутри условного оператора: таким образом мы подчеркнём, что iter нам нужен только здесь. Итератор будет либо указывать на пару из найденного ключа и его значения, либо окажется равен значению data.end(), если ключ не найден. Обратиться к найденной паре можно через унарную звёздочку или стрелочку (iter->first означает (*iter).first). Это похоже на указатели, но важно понимать, что итератор ассоциативного контейнера — это не указатель, а самостоятельный объект.

Вернёмся ещё раз к конструкции data[key]. Она возвращает ссылку на значение, которому можно что-то присвоить. Сначала она проверяет, есть ли уже такой ключ в контейнере. Если ключа нет, он тут же вставляется в контейнер со значением по умолчанию (0 для int). Затем возвращается ссылка на значение в контейнере.

Такое поведение оператора [] требует, чтобы контейнер data был изменяемым. Поэтому выражение data[key] не скомпилируется, если data — константа:

```
void Check(const std::map<std::string, int>& data) {
   if (data["total"] > 0) { // ошибка компиляции!
   // ...
```

```
    }
}
```

Если мы уверены, что ключ в контейнере есть, то можно воспользоваться функцией at:

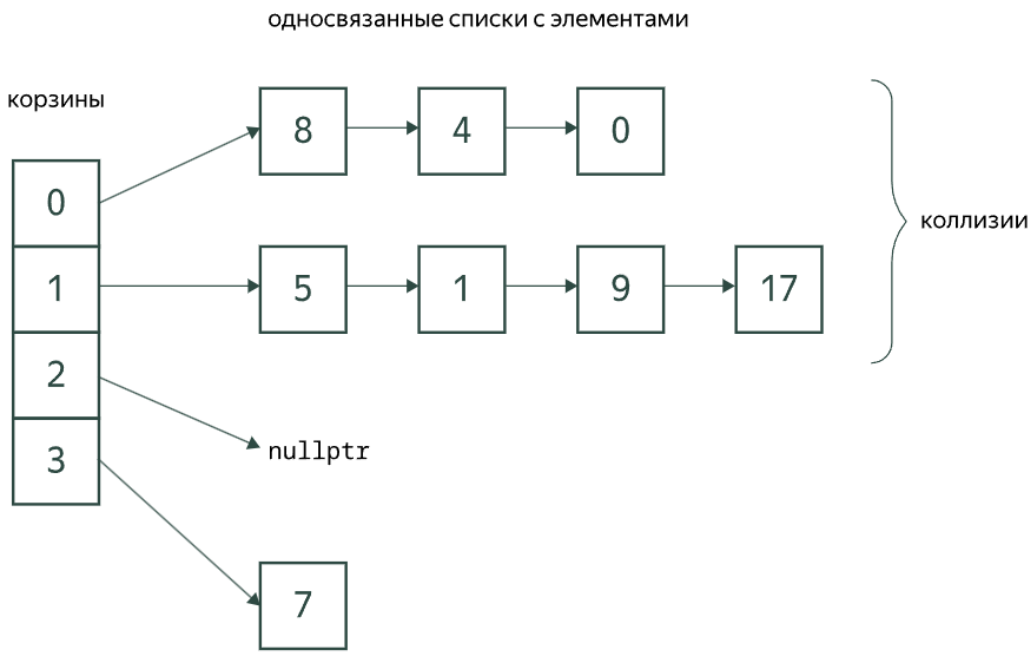
```
void Check(const std::map<std::string, int>& data) {
    if (data.at("total") > 0) { // OK, это скомпилируется
    // ...
    }
}
```

Если же ключа всё же не окажется, то at во время работы программы сгенерирует исключение — «цивилизованную» ошибку, которую можно перехватить и обработать.

Контейнер std::unordered_map

Рассмотрим другую реализацию ассоциативного массива из стандартной библиотеки C++ — хеш-таблицу unordered_map. Само название этого класса подчёркивает, что данные будут храниться не упорядоченными по ключу. Предполагается, что для каждого ключа определена хеш-функция (по умолчанию std::hash<Key>()), а по ней вычисляется номер *корзины* (bucket), в которую должен попасть ключ.

Случай, когда два разных ключа оказываются в одной корзине, называется *коллизией*. В C++ для разрешения коллизий используется *метод цепочек*, то есть, внутри одной корзины все элементы выстраиваются в односвязный список.



Если хеш-функция достаточно равномерна и корзин достаточно много, то в среднем время поиска, добавления и удаления элементов для unordered_map будет *константным* (O(1)).

Интерфейс unordered_map специально сделан похожим на интерфейс map. В предыдущих примерах достаточно заменить только заголовочный файл и имя контейнера.

Контейнеры std::set и std::unordered_set

Контейнеры std::set и std::unordered_set похожи на map и unordered_map по внутреннему устройству, но они хранят только ключи, без ассоциированных значений. Вот как можно выписать повторяющиеся слова текста в алфавитном порядке (по одному разу каждое):

```
#include <iostream>
#include <set>
#include <string>
#include <unordered_set>
int main() {
// здесь будем хранить все слова (каждое по одному разу)
std::unordered_set<std::string> words;
// здесь будем хранить повторяющиеся слова
// используем set, а не unordered_set, чтобы потом напечатать их
по алфавиту
std::set<std::string> duplicate_words;
std::string word;
while (std::cin >> word) {
   if (words.contains(word)) {
   duplicate_words.insert(word);
   } else {
      words.insert(word);
      }
}
for (const auto& word : duplicate_words) {
   std::cout << word << "\n";
   }
}
```

Здесь мы применили функцию contains, которая появилась только в C++20. При использовании более старого стандарта нужно написать if (words.find(word) != words.end()).

Заметим, что при попадании в else мы ищем слово в words дважды: один раз для проверки в contains, а другой раз — в insert. Можно было бы обойтись только одним поиском, воспользовавшись тем, что insert возвращает пару из итератора на элемент и флажка с результатом поиска:

```
#include <iostream>
#include <set>
#include <string>
#include <unordered_set>
int main() {
    std::unordered_set<std::string> words;
    std::set<std::string> duplicate_words;
    std::string word;
    while (std::cin >> word) {
        auto [iter, has_been_inserted] = words.insert(word);
        if (!has_been_inserted) {
            duplicate_words.insert(word);
        }
    }
    for (const auto& word : duplicate_words) {
        std::cout << word << "\n";
    }
}
```

Название set происходит от математического понятия множества, где элементы хранятся только по одному разу. Однако никаких теоретико-множественных операций (объединения, пересечения, разности) у set и unordered_set не предусмотрено.

Мультиконтейнеры

В стандартной библиотеке C++ есть четыре мультиконтейнера:

std::multimap (в заголовочном файле map);

std::multiset (в заголовочном файле set);

std::unordered_multimap (в заголовочном файле unordered_map);

std::unordered_multiset (в заголовочном файле unordered_set).

Они аналогичны обычным ассоциативным контейнерам, которые мы рассматривали выше, но в мультиконтейнерах один и тот же ключ может встретиться несколько раз.

Итераторы ассоциативных контейнеров

Контейнеры map, set и их мультиверсии предоставляют двусторонние итераторы, которые можно сдвигать на соседние позиции вперёд и назад. Как и в случае последовательных контейнеров, запрещено выходить за пределы диапазона, ограниченного begin() и end(), и разыменовывать итератор, равный end(). Итераторы таких контейнеров и ссылки (указатели) на элементы никогда не инвалидируются.

```
#include <iostream>
#include <iterator>
#include <map>
#include <string>
int main() {
   std::map<int, std::string> numbers = {
      {100, "hundred"},
      {3, "three"},
      {42, "forty two"},
      {11, "eleven"},
   };
   auto iter = numbers.find(11);
   if (iter != numbers.end()) {
      // печатаем найденный элемент
      const auto& [key, value] = *iter;
      std::cout << "Found: " << key << ": " << value << "\n";
      // Found: 11: eleven
   // печатаем предыдущий элемент
      if (iter != numbers.begin()) {
       const auto& [key, value] = *std::prev(iter);
      std::cout << "Previous: " << key << ": " << value << "\n";
      // Previous: 3: three
      } else {
         std::cout << "No previous element\n";
     }
// печатаем следующий элемент
    if (auto nextIter = std::next(iter); nextIter !=
numbers.end()) {
      const auto& [key, value] = *nextIter;
     std::cout << "Next: " << key << ": " << value << "\n";
     // Next: 42: forty two
```

```
        } else {
            std::cout << "No next element\n";
         }
       } else {
    std::cout << "Not found\n";
   }
}
```

Может показаться, что в строке const auto& [key, value] = *std::prev(iter) мы строим висячие ссылки, так как возвращаемое значение функции prev после вычисления всего выражения сразу станет невалидным. Однако константные ссылки продлевают жизнь объекта до конца текущего блока.

Итераторы unordered-контейнеров однонаправленные: их можно сдвигать только вперёд. Это связано с тем, что коллизии в хеш-таблице обычно разрешаются с помощью односвязного списка элементов, а по односвязному списку нельзя двигаться назад. Итераторы unordered-контейнеров могут инвалидироваться только если произошло рехеширование при вставке. Ссылки и указатели никогда не инвалидируются.

Отдельно отметим функцию erase у ассоциативных контейнеров. У неё есть несколько перегруженных версий. Одна версия принимает ключ, другая — итератор удаляемого элемента, третья — диапазон итераторов. Разница будет для мультиконтейнеров: если какой-то ключ повторяется, то первая версия erase удалит все вхождения таких ключей, а вторая — только конкретные:

```
#include <unordered_map>
int main() {
    std::unordered_multimap<std::string, int> data = {
        {"a", 1},
        {"a", 2},
        {"a", 3},
        {"b", 4},
    };
    auto iter = data.find("a");
    if (iter != data.end()) {
        data.erase(iter);
        // удаляем первое найденное вхождение с ключом "a"
     }
    data.erase("a");
    // удаляем все остальные вхождения с ключом "a"
}
```

3. Порядок выполнения работы

Задание 1. Ознакомьтесь с теоретическим материалом, приведенным в пункте «Краткие теоретические положения» данных методических указаний и конспектом лекций по данной теме.

Задание 2. Разработайте программы по своему варианту. **Согласуйте точное задание по своему варианту перед тем, как начать его выполнять.**

Покажите результаты работы преподавателю. Оформите отчет по работе.

4. Оформление отчета

Отчет по работе должен содержать:

- название и цель работы;

для каждого задания:

- текст задания;
- тексты программ с комментариями;
- оценка временной сложности использованных в работе методов контейнеров;
- анализ результатов выполнения задания.

5. Список рекомендуемых источников

https://academy.yandex.ru/handbook/cpp/article/vectors-and-strings
https://en.cppreference.com/w/cpp/container/vector
https://metanit.com/cpp/tutorial/7.2.php
https://academy.yandex.ru/handbook/cpp/article/associative-containers
https://codeforces.com/blog/entry/9702?locale=ru
https://brestprog.by/topics/containers/