Вложенные вектора. Структуры

Denis Bakin

Константность

Зачем использовать `const`?

- документирует намерение (переменная не будет менять значение)
- даёт гарантии при вызове функций/методов
- можно ли изменить константный объект – конечно, нет ~~(можно)~~

Константные объекты (пример)

- Методы, изменяющие объект, недоступны для `const`-объекта
- Используйте `const`, чтобы ограничить интерфейс

```
#include <iostream>
#include <vector>

int main() {
    const std::vector<int> v = {1, 3, 5};
    std::cout << v.size() << "\n"; // 3
    v.clear();  // nope: Compilation Error
    v[0] = 0;  // nope again: Compilation Error
}
```

Константные ссылки

- Ссылка — псевдоним переменной
- `const T&` — ссылка только для чтения на объект `T`
- Полезно для передачи больших объектов без копирования и без возможности изменить их

```
int main() {
    int x = 42;

    int& ref = x;
    const int& const_ref = x;  // константная ссылка
    ++x;  // инкремент оригинала -- OK
    ++ref;  // инкремент обычной ссылки -- OK
    ++const_ref;  // изменение по константной ссылке невозможно -- CE
}
```

Константные ссылки — преимущества

- Расширяют допустимые аргументы: константные объекты, литералы, временные объекты
- Гарантия неизменности — ошибка компиляции при попытке изменить
- Семантика функции понятна по сигнатуре: что изменяется, а что нет

Константные ссылки в аргумах функций

- Идеальны для `std::string`, `std::vector` и других тяжёлых типов
- Нет копирования, но и нет возможности изменения

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

size_t countChar(const std::string& line, char chr_to_count) {
    size_t cnt = 0;
    for (const char& chr : line) {
        if (chr == chr_to_count) {
            ++cnt;
        }
    }
    return cnt;
}
```

Константные ссылки в аргумах функций

```cpp
size_t countChar(const std::string& line, char chr_to_count);

int main() {
    std::string input_line;
    std::getline(std::cin, input_line);
    // мы уверены, что строка не будет изменена
    std::cout << countChar(input_line, 'a') << '\n';

    // было бы невозможно с неконстантной ссылкой: переменная константна
    const std::string const_line = "another constant line";
    std::cout << countChar(const_line, 'a') << '\n';

    // было бы невозможно с неконстантной ссылкой: переменной по факту нет
    std::cout << countChar("some random line with many aaaaa", 'a') << '\n';
}
```

Константность — практические правила

- Это good practice использовать `const`, всегда, когда значение не должно меняться
- Для больших объектов передавайте `const T&` вместо `T`. Если нужно менять — `T&`
- `const` на уровне интерфейса повышает читаемость и безопасность кода

Вложенные `std::vector<T>`

- **Вложенные вектора** — это вектора, элементами которых являются другие вектора
- Тип: `std::vector<std::vector<int>>`
- Можно рассматривать как **двумерную таблицу (матрицу)**

Вложенные std::vector<T>

Пример создания и индексации:

```
#include <iostream>
#include <vector>

int main() {
    const std::vector<std::vector<int>> data = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    std::cout << data[0][0] << ' ' << data[1][1] << ' ' << data[2][2] << '\n';
    // 1 5 9
}
```

- Доступ к элементу: `matrix[row][column]`
- Сначала выбираем строку, затем элемент в ней

Индексация двумерного вектора

- Каждый элемент внешнего вектора — это отдельный std::vector<int>
- Обращение по индексу выполняется поэтапно:
  - matrix[i] — строка (вектор)
  - matrix[i][j] — элемент внутри строки
- Аналог двумерного массива

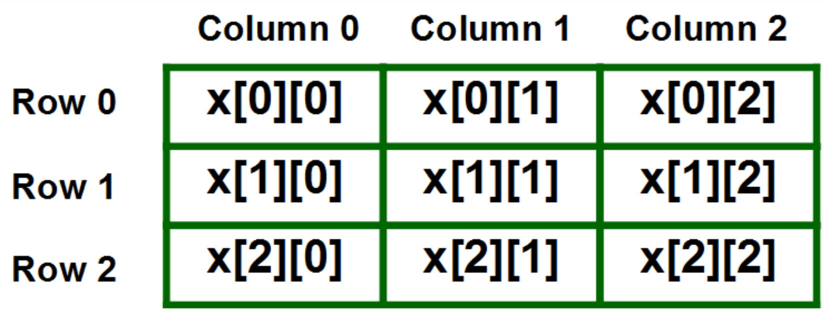

Figure 1: Индексация в двумерном векторе

Двумерный вектор: ввод

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    size_t m, n;
    std::cin >> m >> n;

    std::vector<std::vector<int>> matrix(m);

    for (size_t i = 0; i < m; ++i) {
        matrix[i].resize(n);
        for (size_t j = 0; j < n; ++j) {
            std::cin >> matrix[i][j];
        }
    }
```

Двумерный вектор: вывод

```
// std::vector<std::vector<int>> matrix(m);

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';
}
```

Двумерный вектор: вывод диагонали

- каким свойством обладают элементы на диагонали?
- индексы строки и столбца совпадают: `i == j`

Двумерный вектор: вывод диагонали

- каким свойством обладают элементы на диагонали?
- индексы строки и столбца совпадают: `i == j`

```
// std::vector<std::vector<int>> matrix(m);
for (size_t i = 0; i < std::min(n, m); ++i) {
    std::cout << matrix[i][i] << ' ';
}
std::cout << '\n';
}
```

Двумерный вектор: вывод

```
1 2 3
4 5 6
7 8 9

1 5 9
```

- Первая часть — печать всей матрицы
- Вторая — вывод диагонали, где `i == j`

Поиск максимального значения

- Часто нужно найти **наибольший элемент** во вложенном векторе

Поиск максимального значения

- Часто нужно найти **наибольший элемент** во вложенном векторе
- Используем **два вложенных цикла**

Поиск максимального значения

- Часто нужно найти **наибольший элемент** во вложенном векторе
- Используем **два вложенных цикла**
- Перебор не отличается от одномерных случаев

Поиск максимума: вариант 1

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

int findMaxValue(const std::vector<std::vector<int>>& matrix) {
    int maxValue = matrix[0][0];
    for (const std::vector<int>& row : matrix) {
        for (const int& num : row) {
            if (num > maxValue) {
                maxValue = num;
            }
        }
    }
    return maxValue;
}
int main() {
    std::vector<std::vector<int>> matrix = {{1, 5, 3}, {8, 2, 6}, {4, 9, 7}};
    std::cout << findMaxValue(matrix) << '\n';  // 9
}
```

Поиск максимума: вариант 2 (по индексам)

```cpp
int findMaxValue(const std::vector<std::vector<int>>& matrix) {
    int maxValue = matrix[0][0];
    for (size_t i = 0; i < matrix.size(); ++i) {
        for (size_t j = 0; j < matrix[i].size(); ++j) {
            if (matrix[i][j] > maxValue) {
                maxValue = matrix[i][j];
            }
        }
    }
    return maxValue;
}
```

- Подходит, если нужны индексы `i` и `j`
- Логика полностью совпадает

Структуры

- Позволяют **объединять логически связанные данные** под одним именем
- Можно создавать **новые типы данных**
- Каждый объект структуры содержит **поля** (переменные)

Пример структуры Person

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

struct Person {
    std::string name;
    int height;
    int age;
    bool expelled;
};
```

- Каждое поле имеет собственный тип
- Можно инициализировать по-разному

Использование структуры Person

```cpp
int main() {
    Person person1{"Алексей", 180, 22, false};
    Person person2{"Мария", 165, 20, true};
    Person person3{"Иван", 175, 23, false};
    Person person4{"Екатерина", 170, 21, false};
    Person person5{"Петр", 185, 24, true};

    std::vector<Person> people = {person1, person2, person3, person4, person5};

    for (const Person& person : people) {
        std::cout << "Имя: " << person.name << "\n"
                  << "Рост: " << person.height << "\n"
                  << "Возраст: " << person.age << "\n"
                  << "Отчислен: " << (person.expelled ? "Да" : "Нет") << "\n"
                  << "-----------------\n";
    }
}
```

Структура Point

- Частный случай — хранение координат точки в 3D
- Структура упрощает передачу данных в функции

```
struct Point {
    double x = 0.0;
    double y = 0.0;
    double z = 0.0;
};
```

Работа с точками: расстояние между двумя точками

- как найти расстояние между двумя точками в пространстве?

Работа с точками: расстояние между двумя точками

- как найти расстояние между двумя точками в пространстве?

```
#include <cmath>

double getDistance(const Point& first, const Point& second) {
    double sq = std::pow(first.x - second.x, 2)
              + std::pow(first.y - second.y, 2)
              + std::pow(first.z - second.z, 2);
    return std::sqrt(sq);
}
```

Работа с точками: найти ближайшую точку

- как найти расстояние между двумя точками в пространстве?

Работа с точками: найти ближайшую точку

- как найти расстояние между двумя точками в пространстве?

```
#include <vector>

Point findClosestPoint(const Point& target, const std::vector<Point>& points) {
    Point closest = points[0];
    double minDist = getDistance(closest, target);
    for (const Point& p : points) {
        double d = getDistance(p, target);
        if (d < minDist) {
            minDist = d;
            closest = p;
        }
    }
    return closest;
}
```

Пример использования Point

```cpp
int main() {
    Point p1(1.0, 2.0, 3.0);
    Point p2(4.0, 5.0, 6.0);
    Point p3(7.0, 8.0, 9.0);
    Point p4(2.0, 2.0, 2.0);
    std::vector<Point> points = {p1, p2, p3};

    std::cout << "Расстояние: " << getDistance(p1, p2) << '\n';
    Point closest = findClosestPoint(p4, points);
    std::cout << "Ближайшая: (" << closest.x << ", "
              << closest.y << ", " << closest.z << ")\n";
}
```

Выравнивание структур

- Сумма размеров полей $\neq$ общий размер структуры
- Причина: **выравнивание по адресам**
- Процессору удобнее читать данные, расположенные по кратным адресам

Пример: выравнивание

```
#include <iostream>
struct TestStruct1 {
    double a;
    double b;
};

struct TestStruct2 {
    double a;
    char b;
};
int main() {
    std::cout << sizeof(TestStruct1) << '\n';  // 16
    std::cout << sizeof(TestStruct2) << '\n';  // 16, не 9!
}
```

- Вторую структуру компилятор **дополняет байтами** выравнивания
- Эти байты невидимы и недоступны напрямую

Визуализация выравнивания

```
struct TestStruct2 {
    double a;
    char b;
    char gap[7];  // добавлено компилятором для выравнивания
};
```

- Добавлены “пустые” байты до кратности 8
- Это делает работу процессора эффективнее

Сложное выравнивание

```
struct TestStruct1 { // 24 байта
    char a;
    int b;
    char c;
    double d;
};
```

Эквивалентно:

```
struct TestStruct1 {
    char a;
    char gap_1[3];
    int b;
    char c;
    char gap_2[7];
    double d;
};
```

Оптимизация структуры

```
struct TestStruct1 { // 16 байт
    double d;
    int b;
    char c;
    char a;
};
```

- Перестановка полей может **уменьшить размер** структуры
- Правильный порядок = экономия памяти

Итоги

- Константные переменные, аргументы и ссылки повышают безопасность кода
- Вложенные `std::vector` — основа для двумерных данных
- Структуры — новый тип, объединяющий связанные переменные
- Поля структур могут быть любого типа, включая `std::vector`
- При проектировании важно помнить о **выравнивании и порядке полей**