Сортировки. Компараторы

Denis Bakin

Сортировка — напоминание

- `std::sort` сортирует элементы вектора в порядке возрастания
- `std::set` хранит элементы в порядке возрастания значений
- `std::map` хранит пары в порядке возрастания ключей

Сортировка — напоминание

- `std::sort` сортирует элементы вектора в порядке возрастания
- `std::set` хранит элементы в порядке возрастания значений
- `std::map` хранит пары в порядке возрастания ключей

А как изменить порядок сортировки?

Компараторы

- **Компаратор** – функция или функтор для сравнения двух объектов
- Возвращает `true`, если первый аргумент должен идти **раньше** второго
- Используется в `std::sort`, `std::set`, `std::map` и др.

Формализация упорядоченности

Набор элементов $a_1, \dots, a_n$ считаем **упорядоченным** по функции f, если:

$$\forall\ 1 \leq i < j \leq n : f(a_i, a_j) = 1$$

- Сравнение объекта, который идет раньше, с объектом, который идет позже, должно быть истинным
- Сортировка – это перестановка элементов так, чтобы выполнялось это условие

Простейшие компараторы

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

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

- customLess – сортировка по возрастанию
- customGreater – сортировка по убыванию

Компараторы для сложных типов

```
bool firstCoordOnlyLess(const std::pair<int, int>& a,
                        const std::pair<int, int>& b) {
    return a.first < b.first;
}

bool distanceToOriginLess(const std::pair<int, int>& a,
                          const std::pair<int, int>& b) {
    return a.first * a.first + a.second * a.second
         < b.first * b.first + b.second * b.second;
}
```

- Сравнение только по первой координате
- Сравнение по расстоянию до начала координат

Пример: сортировка координат

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

int main() {
    std::vector<std::pair<int, int>> coords = {
        {3, 7}, {8, 1}, {14, 6}, {9, 3}, {11, 15},
        {11, 12}, {3, 10}, {14, 4}, {17, 8}, {8, 5},
    };

    std::sort(coords.begin(), coords.end(), firstCoordOnlyLess);
    // (3, 7), (3, 10), (8, 1), (8, 5), (9, 3), ...
}
```

Использование std::sort с компаратором

```
std::sort(coords.begin(), coords.end(), firstCoordOnlyLess);
std::cout << "Sorted by first coord: \n";
printCoords(coords);

std::sort(coords.begin(), coords.end(),
          std::less<std::pair<int, int>>());
std::cout << "Sorted by built-in less: \n";
printCoords(coords);

std::sort(coords.begin(), coords.end(), distanceToOriginLess);
std::cout << "Sorted by distance to origin: \n";
printCoords(coords);
```

Результаты сортировки

```
Sorted by first coord:
(3, 7), (3, 10), (8, 1), (8, 5), (9, 3), (11, 15), ...

Sorted by built-in less:
(3, 7), (3, 10), (8, 1), (8, 5), (9, 3), (11, 12), ...

Sorted by distance to origin:
(3, 7), (8, 1), (8, 5), (9, 3), (3, 10), (14, 4), ...
```

- Разные компараторы – разный порядок!

Лямбда-функции

- **Лямбда** – анонимная функция, определяемая на месте
- Синтаксис: `[захват](параметры) { тело }`
- Удобно для компараторов

```
auto cmp = [](const int& a, const int& b) {
    return a > b;  // по убыванию
};
std::sort(vec.begin(), vec.end(), cmp);
```

Лямбда-компараторы inline

```
std::vector<int> numbers = {5, 2, 8, 1, 9};

// Сортировка по убыванию с лямбдой
std::sort(numbers.begin(), numbers.end(),
    [](const int& a, const int& b) {
        return a > b;
    }
);
// 9, 8, 5, 2, 1
```

- Лямбда передается прямо в вызов функции

Структура Person

```
struct Person {
    std::string name;
    int age;
    double height;  // рост в метрах
};
```

- Как отсортировать вектор людей по разным полям?

Компаратор как структура (функтор)

```
struct CompareHeightStruct {
    bool operator()(const Person& a, const Person& b) const {
        return a.height < b.height;
    }
};
```

- Структура с перегруженным operator()
- Можно использовать как компаратор

std::set с компаратором

- `std::set<T, Compare>` – второй шаблонный параметр задаёт компаратор
- По умолчанию `Compare = std::less<T>`

```
std::set<int, std::greater<int>> descendingSet;
descendingSet.insert(3);
descendingSet.insert(1);
descendingSet.insert(2);
// Порядок: 3, 2, 1
```

std::set с лямбда-компаратором

```
auto cmp_name = [](const Person& a, const Person& b) {
    return std::tie(a.name, a.age, a.height)
         < std::tie(b.name, b.age, b.height);
};

std::set<Person, decltype(cmp_name)> peopleByName = {
    {"Alice", 30, 1.65},
    {"Bob", 25, 1.80},
    {"Charlie", 35, 1.75}
};
```

- `decltype(cmp_name)` – тип лямбды
- `std::tie` создаёт кортеж ссылок для лексикографического сравнения
- Множество хранит людей в порядке имён (с tie-breaker по возрасту и росту)

std::set: сортировка по возрасту

```cpp
auto cmp_age = [](const Person& a, const Person& b) {
    return std::tie(a.age, a.name, a.height)
         < std::tie(b.age, b.name, b.height);
};

std::set<Person, decltype(cmp_age)> peopleByAge = {
    {"Alice", 30, 1.65},
    {"Bob", 25, 1.80},
    {"Charlie", 35, 1.75}
};
// Bob (25), Alice (30), Charlie (35)
```

std::set: сортировка по росту

```
auto cmp_height = [](const Person& a, const Person& b) {
    return std::tie(a.height, a.name, a.age)
         < std::tie(b.height, b.name, b.age);
};

std::set<Person, decltype(cmp_height)> peopleByHeight = {
    {"Alice", 30, 1.65},
    {"Bob", 25, 1.80},
    {"Charlie", 35, 1.75}
};
// Alice (1.65m), Charlie (1.75m), Bob (1.80m)
```

std::map с компаратором

- `std::map<Key, Value, Compare>` – третий параметр задаёт компаратор ключей
- Работает аналогично `std::set`

```cpp
auto cmp = [](const std::string& a, const std::string& b) {
    if (a.length() != b.length()) {
        return a.length() < b.length();  // сначала по длине
    }
    return a < b;  // затем лексикографически
};

std::map<std::string, int, decltype(cmp)> wordCount;
wordCount["a"] = 1;
wordCount["hello"] = 2;
wordCount["hi"] = 3;
// Ключи: "a", "hi", "hello"
```

Полный пример с выводом

```
template<typename T>
void printPersons(const T& people) {
    for (const auto& person : people) {
        std::cout << person.name << "\t ("
                  << person.age << ", "
                  << person.height << "m)" << '\n';
    }
}
```

Вывод отсортированных множеств

```
People sorted by name:
Alice    (30, 1.65m)
Bob      (25, 1.8m)
Charlie  (35, 1.75m)

People sorted by age:
Bob      (25, 1.8m)
Alice    (30, 1.65m)
Charlie  (35, 1.75m)

People sorted by height:
Alice    (30, 1.65m)
Charlie  (35, 1.75m)
Bob      (25, 1.8m)
```

Сложные сортировки

Можно создавать компараторы для:

- Изменения приоритета полей: сначала по третьему, потом по первому
- Первое поле по возрастанию, второе – по убыванию
- Сортировка по метрике: среднее, медиана, расстояние

Пример: сортировка по двум полям

```
auto cmp = [](const Person& a, const Person& b) {
    if (a.age != b.age) {
        return a.age < b.age;  // сначала по возрасту
    }
    return a.name < b.name;  // затем по имени
};

// Или с использованием std::tie:
auto cmp_tie = [](const Person& a, const Person& b) {
    return std::tie(a.age, a.name) < std::tie(b.age, b.name);
};
```

- При равенстве первого критерия используется второй
- `std::tie` делает код короче и менее подверженным ошибкам

Требования к компаратору

Компаратор должен задавать **строгий слабый порядок**:

1. **Антирефлексивность**: $f(a, a) = false$
2. **Асимметричность**: если $f(a, b) = true$, то $f(b, a) = false$
3. **Транзитивность**: если $f(a, b)$ и $f(b, c)$, то $f(a, c)$

Нарушение этих правил ведёт к UB!

Ошибка: нестрогий компаратор

```
// НЕПРАВИЛЬНО!
auto bad_cmp = [](const int& a, const int& b) {
    return a <= b;  // <= вместо <
};

// ПРАВИЛЬНО
auto good_cmp = [](const int& a, const int& b) {
    return a < b;
};
```

- Используйте <, а не <=
- Используйте >, а не >=

Итоги

- Компаратор – функция сравнения для определения порядка
- Сортировка переставляет элементы так, чтобы все пары удовлетворяли компаратору
- Лямбда-функции удобны для создания компараторов на месте
- `std::set` и `std::map` принимают компаратор как шаблонный параметр
- Используйте `decltype` для указания типа лямбды
- Компаратор должен задавать строгий слабый порядок
- Для `std::set`/`std::map` используйте tie-breaker'ы (`std::tie`), чтобы различные элементы не считались равными