Пары и кортежи
Конспект

Улучшаем поиск

Сортировка объектов по одному параметру

Информация очевидцев о потерянном животном хранится в структуре:

```
struct AnimalObservation {
      string name;   // кличка
      int days_ago;  // сколько дней назад видели
};
```

Пишем вектор структур и сортируем объекты компаратором по days_ago:

```
vector<AnimalObservation> observations = {{"Мурка"s, 3}, {"Рюрик"s, 1}, {"Веня"s, 2}};
sort(observations.begin(), observations.end(),
     [](const AnimalObservation& lhs, const AnimalObservation& rhs) {
           return lhs.days_ago < rhs.days_ago;
     });
// получим: {"Рюрик"s, 1}, {"Веня"s, 2}, {"Мурка"s, 3}
```

Сортировка объектов по двум параметрам

Информация очевидцев о потерянном животном и состоянии его здоровья хранится в структуре:

```
struct AnimalObservation {
      string name;       // кличка
      int days_ago;      // сколько дней назад видели
      int health_level;  // состояние здоровья
};
```

Чтобы сортировать объекты по этим двум параметрам, применяем условный оператор:

```
sort(observations.begin(), observations.end(),
      [](const AnimalObservation& lhs, const AnimalObservation& rhs) {
            if (lhs.days_ago == rhs.days_ago) {
                  return lhs.health_level < rhs.health_level;
           } else {
               return lhs.days_ago < rhs.days_ago;
           }
      });
```

Такой компаратор можно записать в виде сложного логического выражения:

```
sort(observations.begin(), observations.end(),
     [](const AnimalObservation& lhs, const AnimalObservation& rhs) {
         // благодаря приоритету операций скобки в выражении необязательны
           return lhs.days_ago < rhs.days_ago
                || (lhs.days_ago == rhs.days_ago
                    && lhs.health_level < rhs.health_level);
     });
```

Если `lhs.days_ago < rhs.days_ago`, первый параметр будет считаться меньше второго, так как первый аргумент операции `||` — истинный.
А если `lhs.days_ago == rhs.days_ago`, первый параметр будет меньше только при `lhs.health_level < rhs.health_level`.

Пары в компараторах

Задача сравнить объекты по двум параметрам автоматизирована. Для этого применяют пары:

```
sort(observations.begin(), observations.end(),
     [](const AnimalObservation& lhs, const AnimalObservation& rhs) {
           return pair(lhs.days_ago, lhs.health_level)
                   < pair(rhs.days_ago, rhs.health_level);
     });
```

Пары сравниваются сначала по возрастанию первой компоненты, а при её равенстве — по возрастанию второй. Такой порядок называется лексикографическим. Строки и векторы упорядочиваем сначала по первой букве, затем по второй и по третьей.

Если код с парами у вас не компилируется из-за ошибки missing template arguments before '(' token, напишите make_pair. Так компилятор сможет сам определить тип компонента пары, и указывать pair<int, int> не потребуется.

Кортежи. Начало

Для сортировки по двум параметрам подходит пара. Но когда параметров больше, применяют tuple — кортеж. В отличие от пары, он хранит неограниченное количество объектов произвольного размера. От вектора кортежи отличаются тем, что хранят объекты разного типа. Кортежи, как и пары, сравниваются лексикографически:

```
const AnimalObservation lhs = {"Степан"s, 2, 8};
const AnimalObservation rhs = {"Захар"s,  2, 8};
cout << (tuple(lhs.days_ago, lhs.health_level, lhs.name)
         < tuple(rhs.days_ago, rhs.health_level, rhs.name)) << endl;
// выведет 0, так как "Степан"s > «Захар"s
```

Если компилятор старый, и код не компилируется, используйте функцию make_tuple.

Создание кортежей

Кортежи позволяют писать простые и понятные операторы сравнения для структур. Но при создании кортежа объекты копируются в него:

```
string name = "Василий"s;
const tuple animal_info(name, 5, 4.1);
name = "Вася"s;  // в animal_info остался Василий
```

В этом его недостаток. Когда создаём кортеж для сравнения с другим кортежем, тяжёлые объекты могут быть скопированы зря.

Пример копирования тяжелых объектов в сортировке AnimalObservation:

```
struct AnimalObservation {
      string name;
      int days_ago;
      int health_level;

      auto MakeKey() const {
        return tuple(days_ago, health_level, name);
            // потенциально тяжёлая строка name копируется в создаваемый кортеж
    }
};

// ...

sort(observations.begin(), observations.end(),
     [](const AnimalObservation& lhs, const AnimalObservation& rhs) {
           return lhs.MakeKey() < rhs.MakeKey();
     });
```

Избавимся от лишнего копирования функцией tie из библиотеки <tuple>:

```
auto MakeKey() const {
    return tie(days_ago, health_level, name);
}
```

Функци tie возвращает кортеж, хранящий не сами объекты, а ссылки на них: const int& и const string&. Тип string записан своим оригинальным именем с применением basic_string. Ссылки при этом окажутся константными из-за константности самого метода MakeKey. Создание кортежа из ссылок не требует копировать объекты. Однако этот подход неуниверсален, так как создать ссылку на временный объект нельзя:

```
struct Document {
    int id;
    Status status;
    double relevance;
    int rating;

    auto MakeKey() const {
        return tie(status, -rating, -relevance);
    }
};

// ошибка компиляции: cannot bind non-const lvalue reference of type 'int&'
// to an rvalue of type `int'
```

Объекты -rating и -relevance создались внутри метода. Они уничтожаются сразу по окончании выражения, где были созданы. Ссылку на такие временные объекты создать нельзя. Если создаёте кортеж и из ссылок, и из самих значений, укажите типы явно:

```
struct AnimalObservation {
    string name;
    int days_ago;
    int health_level;

    // в заголовке метода теперь указан полный тип
     tuple<int, int, const string&> MakeKey() const {
        // создаём и сразу возвращаем объект:
        // явно указывать его тип не нужно, достаточно указать аргументы конструктора
        // в фигурных скобках
        return {days_ago, -health_level, name};
    }
};
```

Тогда код скомпилируется.

Возврат нескольких значений из функции

Получить доступ к отдельным элементам кортежа можно так:

```
const tuple animal_info("Василий"s, 5, 4.1);
cout  << "Пациент "s << get<0>(animal_info)
      << ", "s << get<1>(animal_info) << " лет"s
      << ", "s << get<2>(animal_info) << " кг"s << endl;
// Пациент Василий, 5 лет, 4.1 кг
```

Другой способ — обратиться к полю по его типу вместо индекса, если это единственное поле указанного типа:

```
const tuple animal_info("Василий"s, 5, 4.1);
cout << "Пациент "s << get<string>(animal_info)
     << ", "s << get<int>(animal_info) << " лет"s
     << ", "s << get<double>(animal_info) << " кг"s << endl;
// Пациент Василий, 5 лет, 4.1 кг
```

У обоих способов такие же недостатки, как у пары с полями first и second: неясно, что лежит в месте использования. Больше подойдёт структура с полями name, age и weight. Но кортежи применяют для возврата нескольких значений из функции. Чтобы обратиться к содержимому кортежей, используют распаковку:

```
class SearchServer {
public:
      tuple<vector<string>, DocumentStatus> MatchDocument(const string& raw_query, int document_id) const {
            // ...
      }

      // ...
};

// ...
const auto [words, status] = search_server.MatchDocument("белый кот"s, 2);
```

Из метода возвращаются два объекта: vector<string> и DocumentStatus. В отличие от случаев, когда структура специально определена, у них нет самостоятельных названий. Догадаться о смысле объектов можно по типам и названию метода. Первый объект — это вектор слов запроса, которые нашлись в документе document_id, а второй объект — статус документа.

Применим кортеж, чтобы вернуть объекты одинаковых или более тривиальных типов. Для этого определим структуру с конкретными названиями полей:

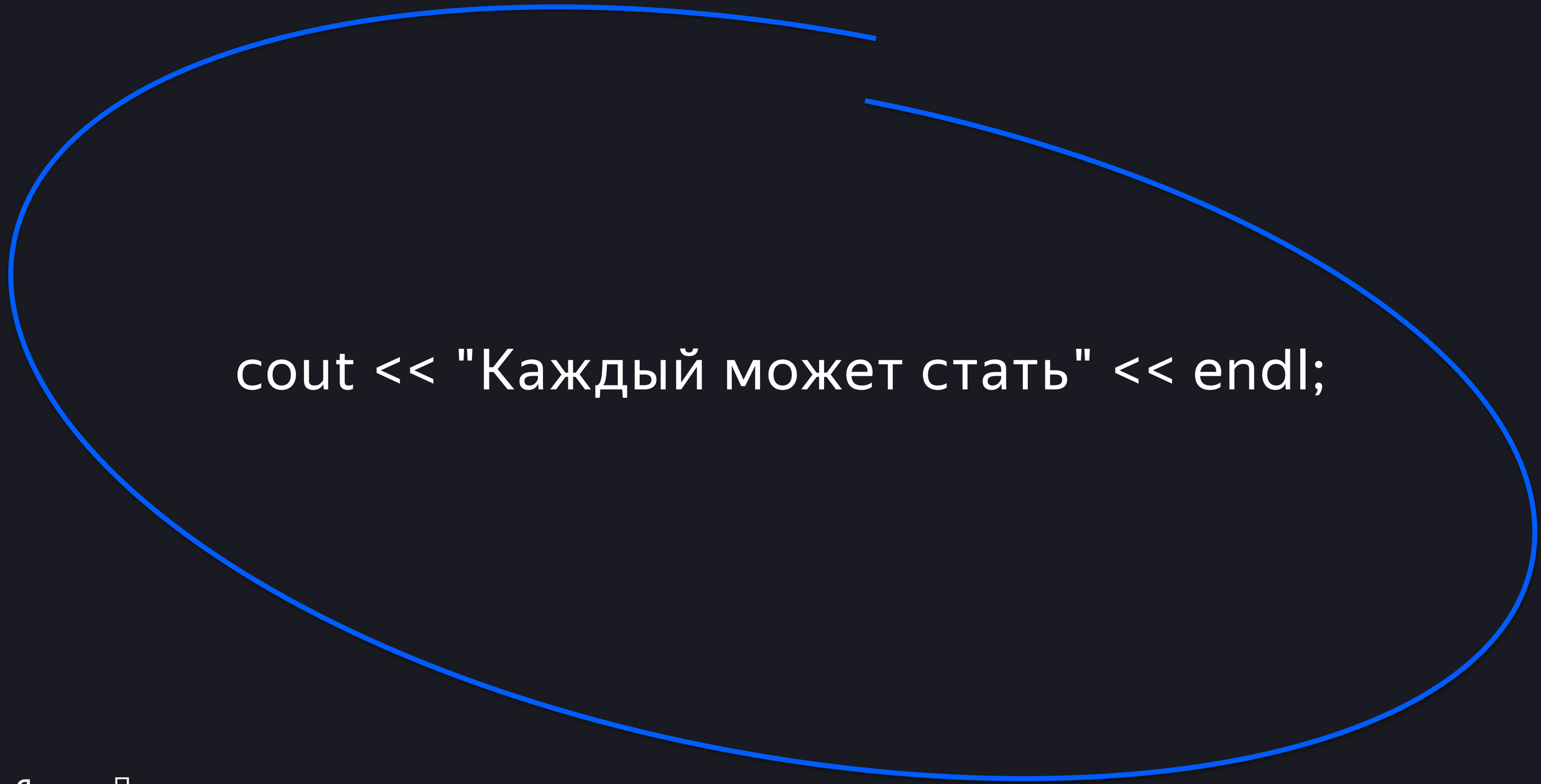
```
// так непонятно
tuple<int, int, double> CheckDocument(/* ... */);

struct DocumentCheckResult {
      int word_count;
      int rating;
      double relevance;
};

// а так гораздо лучше
DocumentCheckResult CheckDocument(/* ... */);
```

Распаковка справится и с кортежем, и со своей структурой:

```
const auto [word_count, rating, relevance] = CheckDocument(/* ... */);
```

Вещественные числа и задача о задачах

Релевантность измеряется вещественными числами. Память, отводимая под переменную типа double, ограничена, а числа хранятся в двоичной записи. Их точность достаточно высока, но неидеальна. Поэтому не стоит сравнивать вещественные числа на равенство операторами ==, !=, <= и >=. Если сделать это нужно, вместо применения == вычислите разность чисел и проверьте, укладывается ли она в погрешность. Функция abs из библиотеки <math> вычисляет модуль числа.


cout << "Каждый может стать" << endl;