Шаблоны функций
Конспект

Яндекс Практикум

Обобщаем функцию

Функция ComputeTermFreqs вычисляет частоту строк по данному вектору. Посчитать частоту можно и для числовых данных. Чтобы узнать количество двуногих и четвероногих обитателей квартиры, напишем две функции с одинаковым названием для типа `int`. По типам аргументов компилятор поймёт, какую функцию вызвать:

```
map<string, int> ComputeTermFreqs(const vector<string>& terms) {
    map<string, int> term_freqs;
    for (const string& term : terms) {
        ++term_freqs[term];
    }
    return term_freqs;
}

map<int, int> ComputeTermFreqs(const vector<int>& terms) {
    map<int, int> term_freqs;
    for (int term : terms) {
        ++term_freqs[term];
    }
    return term_freqs;
}

int main() {
    const vector<int> leg_counts = {4, 2, 4, 4};
    const auto legs_stat = ComputeTermFreqs(leg_counts);
    cout << "Двуногих "s << legs_stat.at(2) << ", "s
         << "четвероногих "s << legs_stat.at(4) << endl;
      // Двуногих 1, четвероногих 3
}
```

Это перегрузка функций. Такой копипаст нежелателен. Перебор циклом for по значению можем убрать — серьёзно функция не замедлится. Отличие останется только в типе. Функцию можно написать, не зная, с каким типом для слов она будет работать.

Тип назовём Term:

```
map<Term, int> ComputeTermFreqs(const vector<Term>& terms) {
    map<Term, int> term_freqs;
    for (const Term& term : terms) {
        ++term_freqs[term];
    }
    return term_freqs;
}
```

Функция зависит не только от конкретного вектора terms, но и от типа его элементов. Такая функция называется шаблонной:

```
template <typename Term>  // шаблонный параметр-тип с названием Term
map<Term, int> ComputeTermFreqs(const vector<Term>& terms) {
    map<Term, int> term_freqs;
    for (const Term& term : terms) {
        ++term_freqs[term];
    }
    return term_freqs;
}
```

Как устроены шаблоны

Типы в C++ фиксированы. Поэтому принципы работы шаблонов могут быть непонятны. Разобраться часто помогают сообщения об ошибках. Вызовем ComputeTermFreqs для вектора структур. Это произойдёт на следующей странице.

```
struct Animal {
    string name;
    int age;
};

Animal FindMaxFreqAnimal(const vector<Animal>& animals) {
    int max_freq = 0;
    Animal max_freq_animal;

      // здесь вызываем шаблонную функцию
    for (const auto& [animal, freq] : ComputeTermFreqs(animals)) {
        if (freq > max_freq) {
            max_freq = freq;
            max_freq_animal = animal;
        }
    }
    return max_freq_animal;
}
```

Шаблонная функция не скомпилируется, так как не может сравнить двух животных операцией <. Сообщение об ошибке компиляции: required from ‘std::map<Term, int> ComputeTermFreqs(const std::vector<Term>&) [with Term = Animal]’. Из него понятно, что ComputeTermFreqs с Term = Animal — это отдельная функция. А ComputeTermFreqs с Term = int компилируется и работает.

Важные свойства шаблонных функций

- ComputeTermFreqs<int>, ComputeTermFreqs<string> и функции с любыми другими Term в угловых скобках — это разные функции. Компилятор копирует их, подставляя нужный тип вместо Term. Конкретная ComputeTermFreqs<Animal> может не скомпилироваться, но по умолчанию никаких требований к типу нет. Главное, чтобы все операции были определены.
- При вызове шаблонной функции указывать в угловых скобках значение её шаблонного параметра необязательно — компилятор постарается вывести шаблонные параметры из типов аргументов.

Универсальные функции вывода контейнеров

Чтобы вывести содержимое контейнера оператором вывода <<, оператор вывода переопределяют в поток для вектора:

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

using namespace std;

ostream& operator<<(ostream& out, const vector<string>& container) {
    for (const string& element : container) {
        out << element << " "s;
    }
    return out;
}

int main() {
    const vector<string> cats = {"Мурка"s, "Белка"s, "Георгий"s, "Рюрик"s};
    cout << cats << endl;
}
```

У функции со специальным названием operator<< два аргумента:

- ссылка на выходной поток (o в ostream значит output);
- константная ссылка на вектор.

Если написать out << container, компилятор вызовет operator<<(out, container). Вернуть ссылку на поток нужно, чтобы объединять вывод в цепочки, как в выражении cout << cats << endl.

Функциональные объекты

Любой объект, который можно использовать как функцию, называется функциональным. Mic drop.

Специализация шаблонов

Объявим enum class AnimalSortKey с типами ключа для сортировки животных:

```
enum class AnimalSortKey {
    AGE,      // по полю age
    WEIGHT,   // по полю weight
    RELATIVE_WEIGHT  // по weight / age
};
```

Попробуем вызывать функцию SortBy с элементом этого перечисления в качестве ключа:

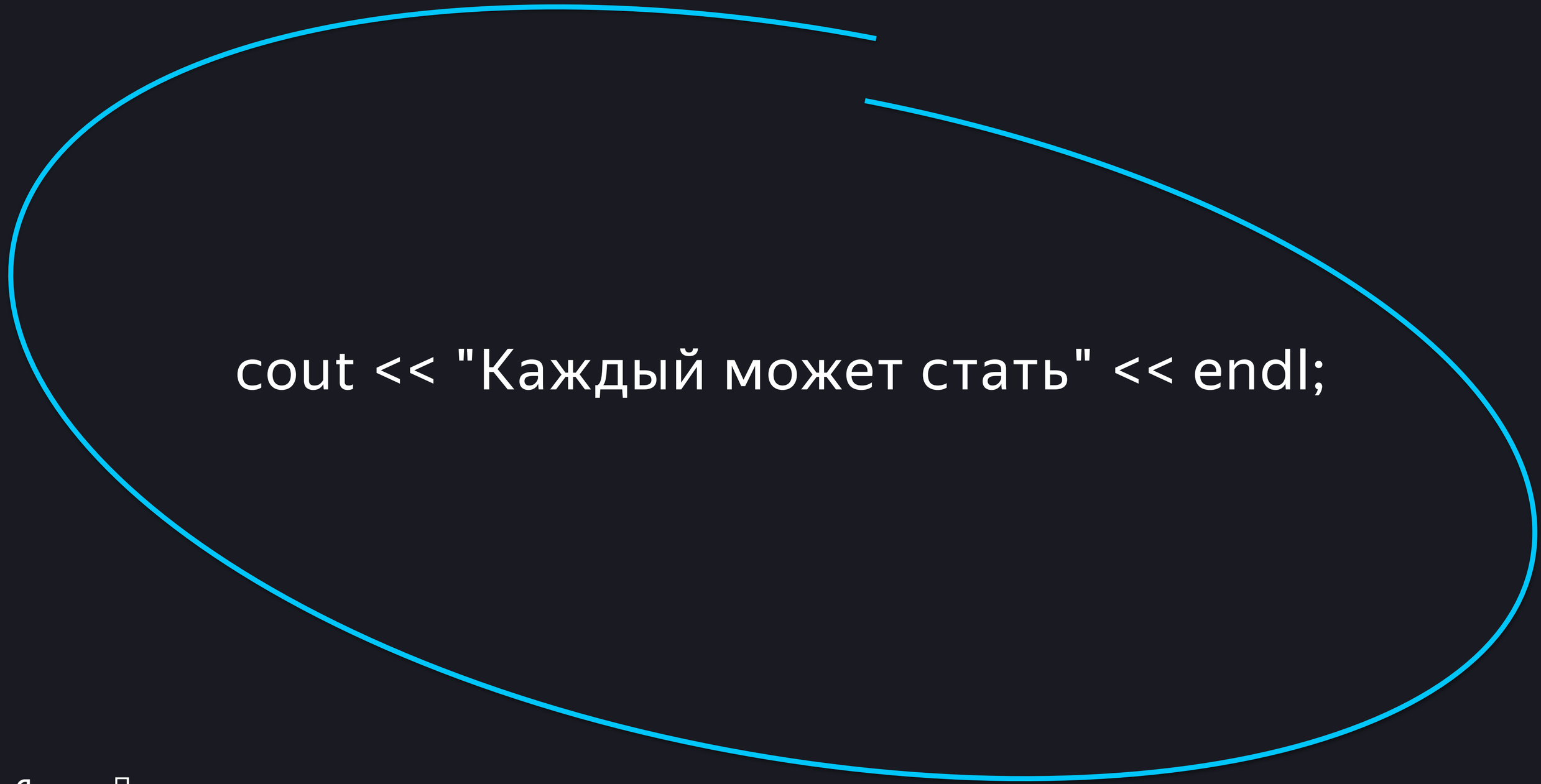
```
int main() {
    vector<Animal> animals = {
        {"Мурка"s,   10, 5},
        {"Белка"s,   5,  1.5},
        {"Георгий"s, 2,  4.5},
        {"Рюрик"s,   12, 3.1},
    };

      PrintNames(animals);
      // Мурка Белка Георгий Рюрик

    SortBy(animals, [](const Animal& animal) { return animal.name; }, true);
    PrintNames(animals);
      // Рюрик Мурка Георгий Белка

    SortBy(animals, AnimalSortKey::RELATIVE_WEIGHT);
    PrintNames(animals);
      // ожидаем вывод: Рюрик Белка Мурка Георгий
}
```

Получим ошибку компиляции.

Чтобы избежать ошибки, нужно написать конкретную версию функции SortBy с аргументами vector<Animal>& animals, AnimalSortKey sort_key, bool reverse = false:

```
void SortBy(vector<Animal>& animals, AnimalSortKey sort_key, bool reverse = false) {
    switch (sort_key) {
        case AnimalSortKey::AGE:
                  // возвращается void, но return помогает сразу выйти из функции
            return SortBy(animals, [](const auto& x) { return x.age; }, reverse);
        case AnimalSortKey::WEIGHT:
            return SortBy(animals, [](const auto& x) { return x.weight; }, reverse);
        case AnimalSortKey::RELATIVE_WEIGHT:
            return SortBy(animals, [](const auto& x) { return x.weight / x.age; }, reverse);
    }
}
```

Здесь функция с конкретными типами аргументов конкурирует с шаблонной и побеждает благодаря своей конкретности.


cout << "Каждый может стать" << endl;