Функции

Denis Bakin

Повторение: ссылки и указатели

Ссылки

- Ссылка — это **псевдоним** существующей переменной
- Не занимает дополнительной памяти
- Не копирует значение, работает напрямую с оригиналом
- Использование идентично переменной, на которую ссылается

```
int a = 10;
int& ref = a; // ref - это "другое имя" для a
ref = 20;     // теперь a == 20
```

Повторение: ссылки и указатели

Указатели

- Указатель хранит **адрес** объекта в памяти
- На 64-битных системах обычно занимает 8 байт
- Доступ к значению осуществляется через оператор * (разыменование)

```
int a = 10;
int* p = &a;   // p хранит адрес переменной a
*p = 20;       // меняем значение a через указатель
std::cout << a; // 20
```

Функция: определение

Что такое функция

- Блок кода, который можно вызвать по имени
- Может принимать аргументы и возвращать значение
- Позволяет переиспользовать и структурировать код

```
int getSum(int a, int b) {
    return a + b;
}
```

Функция: пример использования

```cpp
#include <iostream>

int getSum(int a, int b) {
    return a + b;
}

int main() {
    int x = 5, y = 7;
    std::cout << getSum(x, y) << '\n'; // 12
}
```

- `getSum` принимает два `int`
- Возвращает их сумму
- Ключевое слово `return` завершает выполнение функции

Пример: функция без возвращаемого значения

```
void printHello() {
    std::cout << "Hello!\n";
}

int main() {
    printHello(); // выводит Hello!
}
```

- Тип `void` означает, что функция **ничего не возвращает**

Зачем нужны функции

- **Переиспользование кода** — вызываем один и тот же блок из разных мест
- **Читаемость** — осмысленные имена (`isEven`, `getSum`) делают код понятнее
- **Упрощение отладки** — можно тестировать и исправлять отдельные части программы

Аргументы функции

Передача по значению

- Создаётся **копия** аргумента
- Изменения не влияют на оригинальную переменную

```
int f(int a) {
    a = 100;
    return a;
}

int main() {
    int x = 5;
    f(x);
    std::cout << x; // 5
}
```

Аргументы функции

Передача по ссылке

- Аргумент передаётся как **ссылка на оригинал**
- Изменения внутри функции изменяют исходную переменную
- Нет лишнего копирования

```
void setToZero(int& a) {
    a = 0;
}

int main() {
    int x = 5;
    setToZero(x);
    std::cout << x; // 0
}
```

Когда использовать ссылки

1. **Избежать копирования** больших объектов
   - `std::vector`, `std::string`, структуры
2. **Изменить внешнюю переменную**
   - Например, функция, дополняющая строку или изменяющая счетчик

Аргументы по ссылке: пример без копирования

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

- Строка передаётся как `const std::string&`
- Нет копирования
- `const` запрещает изменение строки внутри функции

Аргументы по ссылке: изменение внешних данных

```
void addToString(std::string& s, char c, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        s += c;
    }
}
```

- Функция изменяет строку s напрямую
- Не создаёт копий, изменения сохраняются снаружи

Необязательные аргументы

- Можно задавать **значения по умолчанию**
- При вызове можно опускать необязательные параметры (с конца)

```
void printMessage(const std::string& msg, char border = '*', int repeat = 3) {
    for (int i = 0; i < repeat; ++i) {
        std::cout << border;
    }
    std::cout << " " << msg << " ";
    for (int i = 0; i < repeat; ++i) {
        std::cout << border;
    }
    std::cout << '\n';
}

printMessage("Hi");         // *** Hi ***
printMessage("Hi", '#');    // ### Hi ###
printMessage("Hi", '=', 5);// ===== Hi =====
```

Возвращаемое значение

- Тип возвращаемого значения указывается перед именем функции
- **Copy elision**: лишнего копирования нет — компилятор оптимизирует передачу

```
uint64_t calcSum(uint64_t a, uint64_t b) {
    uint64_t res = a + b;
    return res;
}
```

Перегрузка функций

- Одинаковое имя функции
- Разный набор или тип аргументов

```
void logger(const std::string& msg);
void logger(int code, const std::string& msg);
void logger(const std::string& msg, const std::string& severity);
```

- Компилятор выбирает нужную версию по аргументам вызова

Лямбда-функции

- **Функция без имени**, которую можно хранить в переменной
- Синтаксис: `[capture](args){ тело }`

```
auto adder = [](int a, int b){ return a + b; };
std::cout << adder(2, 3); // 5
```

Захват переменных (capture)

- [] — без доступа к внешним переменным
- [=] — захват по значению
- [&] — захват по ссылке

```
int inc = 10;
auto f = [&](int x){ return x + inc; };
```

Функции высшего порядка

- Функция может принимать **другую функцию** в качестве аргумента
- Для хранения функции удобно использовать `std::function`

```cpp
#include <functional>

void apply(int n, std::function<bool(int)> check) {
    if (check(n)) std::cout << "OK\n";
}

int main() {
    std::function<bool(int)> isEven = [](int x){ return x % 2 == 0; };
    apply(4, isEven); // OK
}
```