Функции

Denis Bakin

Повторение — ссылки и указатели

- Что такое ссылка?

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал
  - `T&`: например, `int&`, `std::vector<int>&`

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал
  - `T&`: например, `int&`, `std::vector<int>&`
- Что такое указатель?

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал
  - `T&`: например, `int&`, `std::vector<int>&`
- Что такое указатель?
- Указатель — адрес объекта в памяти

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал
  - `T&`: например, `int&`, `std::vector<int>&`
- Что такое указатель?
- Указатель — адрес объекта в памяти
  - Хранит адрес начала объекта

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал
  - `T&`: например, `int&`, `std::vector<int>&`
- Что такое указатель?
- Указатель — адрес объекта в памяти
  - Хранит адрес начала объекта
  - На 64-битных системах занимает **8 байт**

Что такое функция

- Функция — **именованный блок кода**, который можно вызвать
- Принимает аргументы
- Возвращает значение (через `return`)
- Повышает читаемость и повторное использование

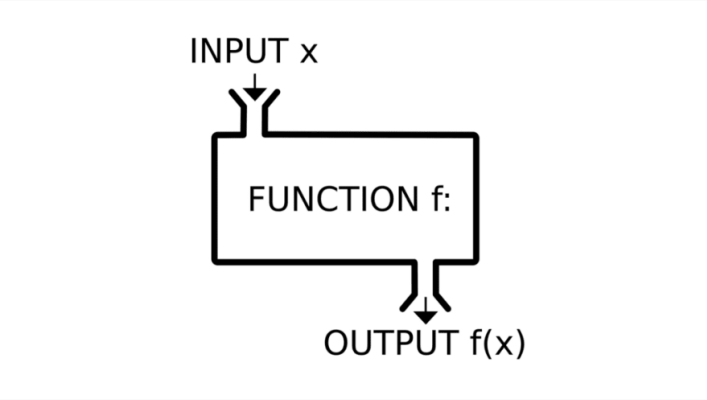

Figure 1: Иллюстрация работы функции

Что такое функция

- Функция — **именованный блок кода**, который можно вызвать
- Принимает аргументы
- Возвращает значение (через `return`)
- Повышает читаемость и повторное использование

```cpp
#include <iostream>

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

int main() {
    int first, second;
    std::cin >> first >> second;
    int result = getSum(first, second);
    std::cout << "Sum is " << result << '\n';
}
```

Функции с условиями

- Вызов функции с несколькими аргументами разных типов
- `void` — тип функции, которая “ничего не возвращает”
- такие функции называются процедурами
- Внутри можно использовать `return` для выхода

Функции с условиями

Пусть есть некоторая задача с проверками:

- подсчитать количество чётных чисел
- проверить последние цифры суммы
- вывести вердикт по результатам проверок

```
// #include ...

void outputVerdict(size_t cnt, size_t t, uint64_t last_digits, uint64_t required_last_digits)
    if (last_digits != required_last_digits && cnt > t) {
        std::cout << "all checks failed!\n";
    } else if (last_digits != required_last_digits) {
        std::cout << "last digits do not match!\n";
    } else if (cnt > t) {
        std::cout << "too many even numbers!\n";
    } else {
        std::cout << "OK!\n";
    }
}
```

Упрощённый вариант с return

- Можно завершать выполнение функции сразу после вывода
- Избегаем каскада `if -- else if`

```
void outputVerdict(size_t cnt, size_t t, uint64_t last_digits, uint64_t required_last_digits)
    if (last_digits != required_last_digits && cnt > t) {
        std::cout << "all checks failed!\n";
        return;
    }
    if (last_digits != required_last_digits) {
        std::cout << "last digits do not match!\n";
        return;
    }
    if (cnt > t) {
        std::cout << "too many even numbers!\n";
        return;
    }
    std::cout << "OK!\n";
}
```

Зачем нужны функции?

Зачем нужны функции?

- Переиспользование кода

Зачем нужны функции?

- Переиспользование кода
- Повышение читаемости. Например, названия отражают смысл (`isEven`, `getSum`)

Зачем нужны функции?

- Переиспользование кода
- Повышение читаемости. Например, названия отражают смысл (`isEven`, `getSum`)
- Разделение логики на блоки

Зачем нужны функции?

- Переиспользование кода
- Повышение читаемости. Например, названия отражают смысл (`isEven`, `getSum`)
- Разделение логики на блоки
- Упрощение отладки и тестирования

Аргументы функций

- Аргументы — данные, от которых зависит поведение функции
- Их можно передавать:
  - **по значению** – есть копирование, нельзя менять значение
  - **по ссылке** – нет копирования, можно менять значение

Передача по значению

- Значения копируются
- Изменения не влияют на внешние переменные
- Подходит для небольших типов (`int`, `char`, `bool`)

```cpp
#include <iostream>

int getSum(int a, int b) {
    a = 2;
    return a + b;
}

int main() {
    int first, second;
    std::cin >> first >> second;  // 6 7
    int result = getSum(first, second);
    std::cout << "Sum is " << result << '\n';
    std::cout << first << ' ' << second << '\n';  // 6 7
}
```

Передача по ссылке

- Копирования **нет**
- Изменение аргумента внутри функции изменяет оригинал
- Используется для больших структур и при необходимости изменять внешние переменные

```
#include <iostream>

int getSum(int& a, int& b) {
    a = 2;
    return a + b;
}

int main() {
    int first, second;
    std::cin >> first >> second;  // 5 1
    int result = getSum(first, second);
    std::cout << "Sum is " << result << '\n';
    std::cout << first << ' ' << second << '\n';  // 2 1
}
```

Аргумент по ссылке — без копирования

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

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

int main() {
    std::string input_line;
    std::getline(std::cin, input_line);
    std::cout << countChar(input_line, 'a') << '\n';
}
```

Аргумент по ссылке — изменение внешней переменной

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

void addToString(std::string& line, char chr_to_fill, size_t num) {
    for (size_t i = 0; i < num; ++i) {
        line += chr_to_fill;
    }
}
```

- Функция изменяет строку из вызывающего кода
- Передача по ссылке обязательна, иначе изменения не сохранятся

Необязательные аргументы

- Можно задавать **значения по умолчанию**
- Аргументы передаются **позиционно**
- Нельзя пропускать средние аргументы

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

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

Пример вызовов с необязательными аргументами

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

void printMessage(const std::string& message, char border_char = '*', int repeat_count = 3);

...

int main() {
    printMessage("Some line of text");
    printMessage("Some line of text", '#');
    printMessage("Some line of text", '=', 5);
    printMessage("Some line of text", 'a', 0);
}
```

- Значения по умолчанию — удобный способ сократить вызовы
- Используются для часто повторяющихся настроек
- Нет именованных аргументов, как в Python

Возвращаемое значение

- При `return` **копирования не происходит** (copy elision)
- Компилятор оптимизирует размещение результата

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

uint64_t calcSum(uint64_t a, uint64_t b) {
    uint64_t ret_value = a + b; // с copy elision &result == &ret_value
    ++ret_value;
    return ret_value; // нет копирования, только выход из функции
}

int main() {
    uint64_t first, second;
    std::cin >> first >> second;
    uint64_t result = calcSum(first, second); // с copy elision &result == &ret_value
    std::cout << "Result is " << result << '\n';
}
```

Перегрузка функций

- Одинаковое имя
- Разные аргументы (тип, количество, порядок)
- Повышает читаемость и гибкость

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

void logger(const std::string& message) {
    std::cout << "[INFO]: " << message << std::endl;
}

void logger(int errorCode, const std::string& message) {
    std::cout << "[ERROR " << errorCode << "]: " << message << std::endl;
}

void logger(const std::string& message, const std::string& severity) {
    std::cout << "[" << severity << "]: " << message << std::endl;
}
```

Пример перегрузок в действии

```
void logger(const std::string& message);
void logger(int errorCode, const std::string& message);
void logger(const std::string& message, const std::string& severity);

int main() {
    logger("System started successfully.");
    logger(404, "Resource not found.");
    logger("Disk space running low", "WARNING");
}
```

Вывод:

```
[INFO]: System started successfully.
[ERROR 404]: Resource not found.
[WARNING]: Disk space running low
```

Лямбда функции — основа

- **Анонимные функции** без имени
- Можно хранить в переменной
- Синтаксис:

```
[= or &](args) { ... };
```

```
auto adder = [](int a, int b){
    return a + b;
};
```

Пример лямбда-функции

```cpp
#include <iostream>

int main() {
    int first, second;
    std::cin >> first >> second;
    auto adder = [](int a, int b){
        return a + b;
    };
    int result = adder(first, second);
    std::cout << "Sum is " << result << '\n';
}
```

Захват контекста (capture)

- [] — ничего не видно

Захват контекста (capture)

- `[]` — ничего не видно
- `[=]` — по **значению**, контекст копируется

Захват контекста (capture)

- `[]` — ничего не видно
- `[=]` — по **значению**, контекст копируется
- `[&]` — по **ссылке**, контекст доступен для изменения и без копирования

Захват по значению и по ссылке

Захват по ссылке

```
int increment_by = 15;
auto ref_incrementer = [&](int num){
    return num + increment_by;
};
cout << ref_incrementer(15) << '\n'; // 30
increment_by = 10;
cout << ref_incrementer(15) << '\n'; // 25
```

Захват по значению

```
int increment_by = 15;
auto val_incrementer = [=](int num){
    return num + increment_by;
};
cout << val_incrementer(15) << '\n'; // 30
increment_by = 10;
cout << val_incrementer(15) << '\n'; // 30
```

Функции высшего порядка

- Функция принимает или возвращает **другую функцию**
- Используется `std::function<T(Args ...)>` для типов функций

```
#include <iostream>
#include <functional>

void outputVerdict(int number, std::function<bool(int)> checker) {
    if (checker(number)) {
        std::cout  << "Checker returned true for this value!" << std::endl;
    } else {
        std::cout  << "Checker returned false for this value!" << std::endl;
    }
}
```

Пример функции высшего порядка

```
int main() {
    int a = 1;
    int b = 2;
    std::function<bool(int)> is_even = [](int number) {
        return number % 2 == 0;
    };

    outputVerdict(a, is_even);
    outputVerdict(b, is_even);
}
```

Константность

- Константная переменная — доступна только для чтения
- Значение фиксируется при присваивании (может быть известно только во время выполнения)
- Цель — явное ограничение возможности изменения для ясности и безопасности кода

```cpp
#include <iostream>

int main() {
    // известна до компиляции (compile time)
    const int const_var_1 = 42;

    int number;
    std::cin >> number;
    // определяется только во время исполнения (run time)
    const int const_var_2 = 2 * number;

    const_var_1 += 1; // невозможно, будет ошибка компиляции

}
```

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

```
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

Итоги

- Функции делают код понятным, модульным и масшабируемым
- Передача по ссылке экономит память
- Необязательные аргументы делают интерфейс гибким
- Перегрузка упрощает использование одной логики для разных наборов аргументов
- Константные переменные, аргументы и ссылки повышают безопасность кода