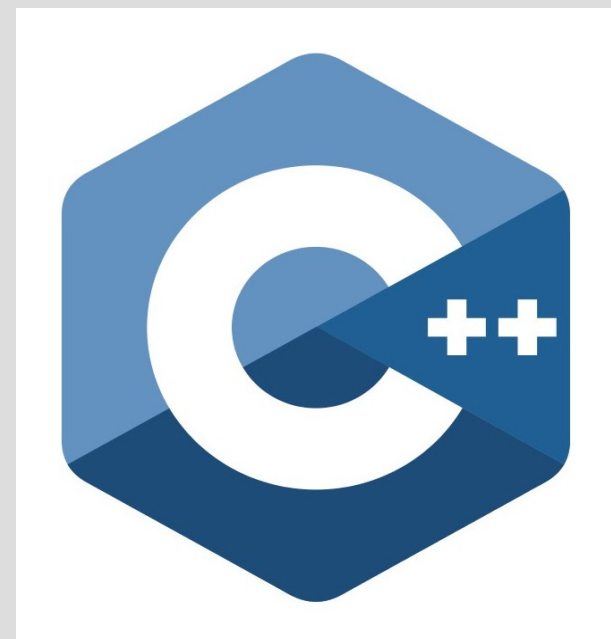
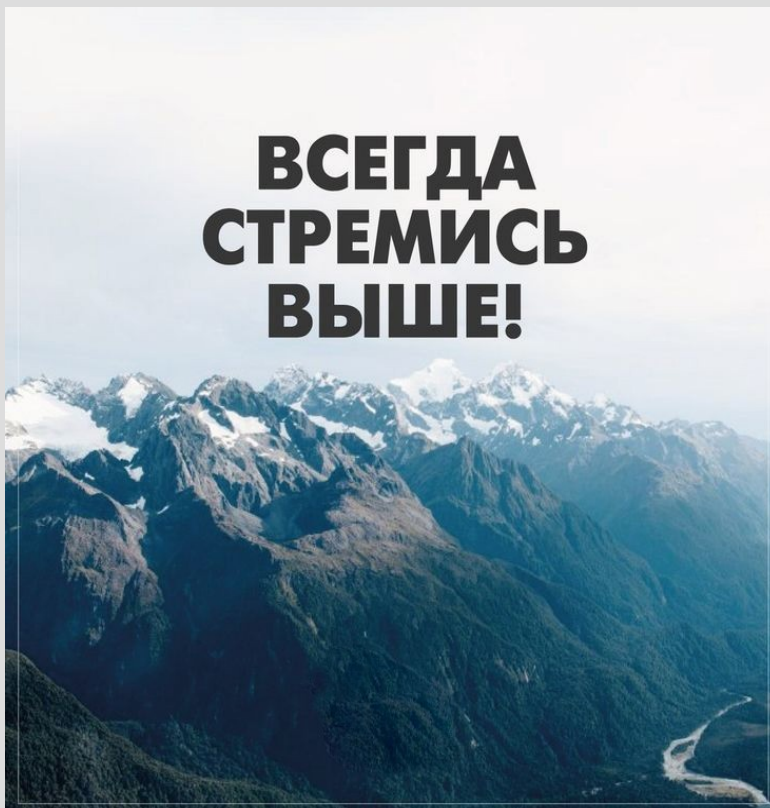


Основы C++. Вебинар №5.

Длительность: 1.5 - 2 ч.



GeekBrains

Что будет на уроке?

- Узнаем всё, или почти всё о функциях: аргументы, параметры, возвращаемые значения.
- Обсудим соглашения о вызове (calling convention).
- Научимся описывать указатели на функции и функции обратного вызова.
- Рассмотрим inline функции и ключевое слово `__fastcall`.
- Узнаем механизм перегрузки функций.
- Изучим, зачем нужны пространства имён.



Подпрограммы в С++ (функции)

Подпрограмма (англ. subroutine) — поименованная или иным образом идентифицированная часть компьютерной программы, содержащая описание определённого набора действий. Подпрограмма может быть многократно вызвана из разных частей программы. В языках программирования для оформления и использования подпрограмм существуют специальные синтаксические средства.

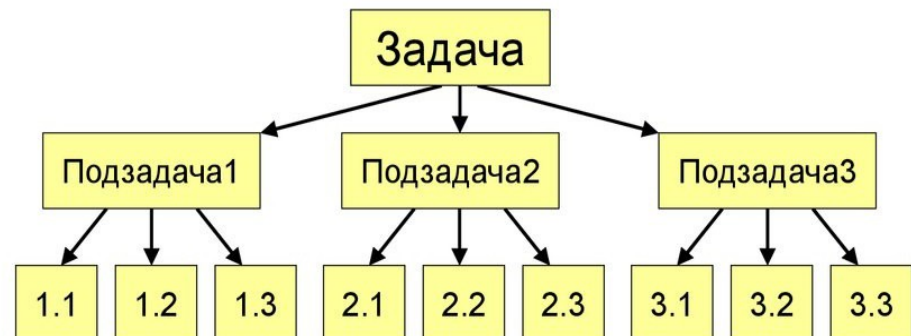
То есть весь алгоритм вашей программы можно и нужно разбивать на взаимодействующие части — подпрограммы (в терминах С++ Функции). Если вы программируете без использования ООП.

Вы уже знакомы с главной функцией с С++ функцией `main`.

Подпрограммы

■ Использование подпрограмм

- сокращает описание алгоритма (выполнение одинаковых действий в разных местах программы)
- структурирует описание алгоритма (разбивка программы (или другой подпрограммы) на подзадачи для лучшего восприятия)
- позволяет реализовать на практике принципы **структурного программирования** при построении больших программ



Параметры и аргументы функций

Во многих случаях нам нужно будет передавать данные в вызываемую функцию, чтобы она могла с ними как-то взаимодействовать. Например, если мы хотим написать функцию умножения двух чисел, то нам нужно каким-то образом сообщить функции, какие это будут числа. В противном случае, как она узнает, что на что перемножать? Здесь нам на помощь приходят параметры и аргументы.

Параметр функции — это переменная, которая используется в функции, и значение которой предоставляет caller (вызывающий объект). Параметры указываются при объявлении функции в круглых скобках.

```
int Sum(int a, int b) // Параметры a и b
{
    ...
}
```

Аргумент функции — это значение, которое передается из caller-а в функцию и которое указывается в скобках при вызове функции в caller-е:

```
int c = Sum(10, 20); // Аргументы 10 и 20
```

Обратите внимание, аргументы тоже перечисляются через запятую. Количество аргументов должно совпадать с количеством параметров, иначе компилятор выдаст сообщение об ошибке.

Функции в C++

```
ВозвращаемоеЗначение  ИмяФункции  (Параметры)
{
    Тело функции;

    return Значение;  // или просто return; для типа void
}
```

Например:

```
#include <iostream>

using namespace std;

void PrintHello()  // Функция без параметров и ничего не возвращает
{
    cout << "Hello, world!" << endl;
}

int main()
{
    PrintHello();  // Два раза ее вызываем из main
    PrintHello();

    return 0;
}
```

Функция — печать массива

В C++ можно передать аргументы в функцию 3-я способами:

- По значению (передается копия переменных).
- По ссылке (передается адрес переменных).
- По указателю (передается адрес переменных).

Передача по значению (переменная size), массивы передаются словно по указателю (int * arr):

```
bool PrintArray(int arr[], int size)    // Функция для печати произвольного массива
{
    for (size_t i = 0; i < size; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    return true;
}

int main()
{
    const int mysize = 5;
    int myarr[] = { 10, 20, 30, 20, 10 };

    if (PrintArray(myarr, mysize))    // Вызываем нашу функцию
    {
        cout << "Array was successfully printed" << endl;
    }
    return 0;
}
```

Консоль отладки Microsoft Visual Studio

10 20 30 20 10

Array was successfully printed

Простой пример

По значению, по ссылке, по указателю

```
#include <iostream>

using namespace std;

void TryToChange(int a, int & b, int * pC)
{
    a = 1001;
    b = 1002;
    *pC = 1003;
}

int main()
{
    int a = 1, b = 2, c = 3;
    cout << "a=" << a << " b=" << b << " c=" << c << endl;
    TryToChange(a, b, &c);
    cout << "a=" << a << " b=" << b << " c=" << c << endl;
    return 0;
}
```

Консоль отладки Microsoft

a=1 b=2 c=3

a=1 b=1002 c=1003

C:\Users\Dmitry\source

.
Чтобы автоматически
томатически закрыть
Нажмите любую клавиш

Функция инициализации массива

Передача по указателю:

```
#include <iostream>
#include <cstdlib> // для функций rand()

using namespace std;

bool InitArray(int* arr, int size) // Получаем указатель на массив и его размер
{
    for (size_t i = 0; i < size; i++)
    {
        arr[i] = rand() % 100; // Случайное число от 0 до 99
    }
    return true;
}

int main()
{
    const int mysize = 5;
    int myarr[mysize];

    if (InitArray(myarr, mysize))
    {
        cout << "Array was successfully initialized" << endl;
    }
    return 0;
}
```


Инициализация структуры по ссылке

Передача по ссылке:

```
struct TPerson {
    string name;
    int id;
};

void InitStruct(TPerson & refVar)
{
    refVar.id = 1053872627;
    refVar.name = "Ivan Petrov";
}

int main()
{
    TPerson p1;
    InitStruct(p1); // Передаем структуру в функцию по ссылке для инициализации
    cout << "id = " << p1.id << " name = " << p1.name << endl;
    return 0;
}
```

Инициализация структуры по указателю

Передача по указателю:

```
struct TPerson {  
    string name;  
    int id;  
};  
  
void InitStruct(TPerson* pVar)  
{  
    pVar->id = 1053872627;  
    pVar->name = "Ivan Petrov";  
}  
  
int main()  
{  
    TPerson p1;  
  
    InitStruct(&p1); // Передаем адрес структуры в функцию для инициализации  
  
    cout << "id = " << p1.id << " name = " << p1.name << endl;  
    return 0;  
}
```

Возврат структуры по значению

```
struct Employee { // Новый тип данных Сотрудник
    long id;        // ID сотрудника
    unsigned short age; // его возраст
    double salary;   // его зарплата
};

Employee GetStruct(long id, unsigned short age, double salary)
{
    Employee em;
    em.id = id;
    em.age = age;
    em.salary = salary;

    return em; // Возвращаем по значению
}

int main(int argc, char* argv[])
{
    Employee eee = GetStruct(543452465, 33, 120'000.0);

    // используем полученную структуру

    return 0;
}
```

Объявление (прототип) и определение функции

```
#include <iostream>

int add(int x, int y); // предварительное объявление функции add() (прототип)

int main()
{
    // это работает, так как мы предварительно (выше функции main()) объявили функцию add()
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl;

    return 0;
}

int add(int x, int y) // хотя определение функции add() находится ниже её вызова
{
    return x + y;
}
```

Прототипы функций обычно помещают в заголовочный h файл и подключают в тех cpp модулях где нужно вызывать функцию из другого модуля. Таким образом прототипов в коде проекта может быть несколько/много а реализация (определение) должно быть одно — ORD никто в C++ не отменял.

Правило одного определения (One Definition Rule, ODR) — один из основных принципов языка программирования C++. Назначение ODR состоит в том, чтобы в программе не могло появиться два или более конфликтующих между собой определения одной и той же сущности (типа данных, переменной, функции, объекта, шаблона).

Соглашение о вызове

Соглашение о вызове (англ. calling convention) — описание технических особенностей вызова подпрограмм, определяющее:

Соглашение о вызове описывает следующее:



- способ передачи аргументов в функцию (регистры, стек).
- порядок размещения аргументов в регистрах и/или стеке.
- код, ответственный за очистку стека (вызывающая функция, вызываемая функция).
- конкретные инструкции, используемые для вызова и возврата.
- способ передачи в функцию указателя на текущий объект (this или self) в объектно-ориентированных языках (через регистры или стек).
- код, ответственный за сохранение и восстановление содержимого регистров до и после вызова функции (вызывающая функция, вызываемая функция).
- список регистров, подлежащих сохранению/восстановлению до/после вызова функции.

Соглашения о вызовах, используемые на x86 при 32-битной адресации:
cdecl, pascal, stdcall или winapi, fastcall, safecall, thiscall

fastcall - общее название соглашений, передающих параметры через регистры (обычно это самый быстрый способ, отсюда название). Если для сохранения всех параметров и промежуточных результатов регистров не достаточно, используется стек.

Ключевые слова: `__fastcall` и `inline`

`inline` — рекомендация компилятору встроить код функции в место ее вызова. Повышается скорость работы программы, но увеличивается ее размер.

`__fastcall` — рекомендуем компилятору передавать аргументы функции через регистры CPU.

Пример:

```
#include <iostream>

using namespace std;

inline int Add(int a, int b) // Функция суммирования с inline
{
    return a + b;
}

void __fastcall PrintAB(int a, int b) // Передаем аргументы функции через регистры CPU
{
    cout << "a=" << a << " b=" << b << endl;
}

int main()
{
    int c = Add(100, 200);
    PrintAB(c, 2*c);        // Вывод на экран: a=300 b=600
    return 0;
}
```

Переменное число параметров функции (только для продвинутых)

Язык программирования C допускает использование функций, которые имеют нефиксированное количество параметров. Более того может быть неизвестным не только количество, но и типы параметров. То есть точное определение параметров становится известным только во время вызова функции. Для определения параметров неопределенной длины в таких функциях используется **многоточие**. При этом надо учитывать, что функция должна иметь как минимум один обязательный параметр.

Функция, которая вычисляет сумму чисел, количество чисел нефиксировано:

```
#include <iostream>
#include <cstdarg>

int add_nums(int count, ...) // сложение списка чисел
{
    int result = 0;
    std::va_list args;
    va_start(args, count); // начинаем работать с параметрами
    for (int i = 0; i < count; ++i) {
        result += va_arg(args, int); // получить следующий параметр
    }
    va_end(args); // заканчиваем работать
    return result;
}

int main()
{
    std::cout << add_nums(4, 25, 25, 50, 50) << std::endl;
    std::cout << add_nums(2, 11, 12) << std::endl;
    return 0;
}
```

Перегрузка функций

Перегрузка функций — это возможность определять несколько функций с одним и тем же именем, но с разными параметрами. Это возможно благодаря тому что компилятор делает: name mangling (изменение имени, декорация имени).

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


using namespace std;

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

string sum(string a, string b)
{
    return a + b;
}

string sum(string a, string b, string c)
{
    return a + b + c;
}

int main(void)
{
    cout << "sum=" << sum(4, 1) << endl;
    cout << "sum=" << sum("aaa", "bbb") << " sum=" << sum("123", "ASD", "poi") << endl;
    return 0;
}
```

 Консоль отладки Microsoft Visual Studio

```
sum=5
sum=aaabbb sum=123ASDpoi
```


Указатели на функцию (только для продвинутых)

Можно использовать объектную обертку `std::function` (since C++11) способ 1, или объявлять сырые указатели на функцию (способ 2). Если указатель на функцию передается в другую функцию как параметр то это называется **функцией обратного вызова** (callback function).

```
#include <iostream>
#include <functional> // Для std::function
```

```
using namespace std;
```

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

```
int Subtract(int a, int b)
{
    return a - b;
}
```

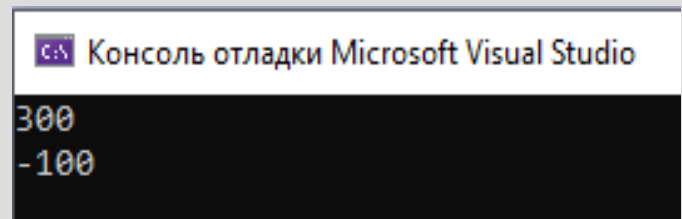
```
int main(void)
{
    std::function<int(int, int)> fun = Add; // Способ 1. Настраиваем "обертку" fun на функцию Add

    cout << fun(100, 200) << endl;

    int (*funPtr)(int, int) = Subtract; // Способ 2. Настраиваем указатель на функцию Subtract

    cout << funPtr(100, 200) << endl;

    return 0;
}
```



Callback functions

(информация для продвинутых)

```
typedef double (*MyFunPtr)(double, double); // Создаем тип данных MyFunPtr

double Add(double a, double b) // Функция сложения
{
    return a + b;
}

double Subtract(double a, double b) // Функция вычитания
{
    return a - b;
}

double PerformCalculation(MyFunPtr PtrF) // Сложные вычисления, параметр callback функция
{
    const double pi = 3.1415926535, g = 9.80665;

    double ret = PtrF(pi, g) + pi * g * 2 - g * g; // сложная формула

    return ret;
}

int main(void)
{
    cout << PerformCalculation(Add) << endl;
    cout << PerformCalculation(Subtract) << endl;
    return 0;
}
```

Консоль отладки Microsoft Visual Studio

-21.6051

-41.2184

Пространства имен - namespace

Конфликт имен возникает, когда два одинаковых идентификатора находятся в одной области видимости, и компилятор не может понять, какой из этих двух следует использовать в конкретной ситуации. Компилятор или линкер выдаст ошибку, так как у них недостаточно информации, чтобы решить эту неоднозначность. Как только программы увеличиваются в объемах, количество идентификаторов также увеличивается, следовательно, увеличивается и вероятность возникновения конфликтов имен.

```
#include <iostream>

namespace MyStd // Объявляем наше пространство имен
{
    int foo = 100;

    void bar(float f)
    {
        std::cout << "Some useful function: " << f;
    }
}

int main()
{
    MyStd::foo = 1000;

    MyStd::bar(8.888);

    return 0;
}
```

Бывают также безымянные пространства имен.
Можно погулить или почитать тут: <https://it.wikireading.ru/35901>



Рекурсия

Рекурсия — кода функция вызывает саму себя. Чтоб не получилась бесконечная рекурсия и переполнение стека нужно предусмотреть остановку рекурсии.

Некоторые сложные алгоритмы упрощаются если использовать рекурсию.

```
#include <iostream>

using namespace std;

unsigned long factorial(unsigned long f) // рекурсивная функция для нахождения n!
{
    if (f == 1 || f == 0) // базовое или частное решение
    {
        return 1; // все мы знаем, что 1!=1 и 0!=1
    }

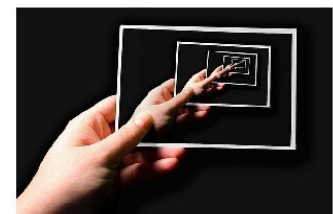
    // функция вызывает саму себя, причём её аргумент уже на 1 меньше
    unsigned long result = f * factorial(f - 1);

    return result;
}

int main(int argc, char* argv[])
{
    unsigned long n;
    cout << "Enter n!: ";
    cin >> n;
    cout << n << "!" << " = " << factorial(n) << endl;
    return 0;
}
```

Рекурсия

«Чтобы понять рекурсию,
надо сначала понять рекурсию»



Аргументы функции по умолчанию

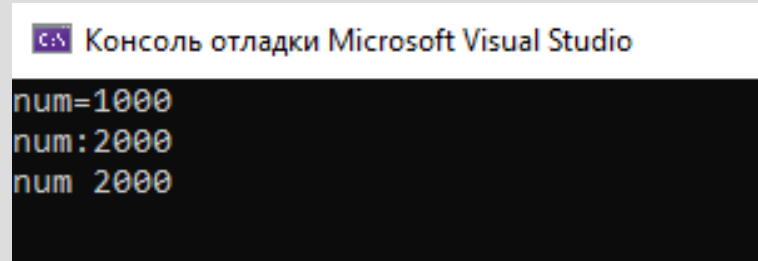
```
#include <iostream>

using namespace std;

void PrintNumber(int num, char separator = '=')
{
    cout << "num" << separator << num << endl;
}

int main(int argc, char* argv[])
{
    PrintNumber(1000);          // Срабатывает параметр по умолчанию

    // Задаем другое значение для параметра по умолчанию
    PrintNumber(2000, ':');
    PrintNumber(2000, ' ');
    cout << endl;
    return 0;
}
```



Консоль отладки Microsoft Visual Studio

```
num=1000
num:2000
num 2000
```

Замечания:

- Параметры по умолчанию всегда должны быть **в конце списка** параметров функции.
- Если вы объявляете прототип функции – надо определить параметры по умолчанию **именно в прототипе**. В самом определении функции этого делать уже не надо.

Класс памяти static

У нас в программе 2 модуля:

Source.cpp

```
int foo = 100;
```

Main.cpp

```
#include <iostream>
```

```
static int foo = 100; // 1. Обходим правило ORD
```

```
void fun()
```

```
{  
    static int bar = 10; // 2. Переменная не уничтожается после выхода из fun  
    bar++;  
    std::cout << "bar=" << bar << std::endl;  
}
```

```
int main()
```

```
{  
    fun();  
    fun();  
    fun();  
    std::cout << "foo=" << foo << std::endl;  
    return 0;  
}
```



Консоль отладки Microsoft Visual Studio

```
bar=11  
bar=12  
bar=13  
foo=100
```

Аббревиатура MVP

MVP — (англ. minimum viable product) — минимально жизнеспособный продукт.

Версия продукта, обладающая минимальными, но достаточными для удовлетворения первых потребителей функциями.

Основная задача — получение обратной связи (feedback) для формирования гипотез дальнейшего развития продукта.



Сайт CppReference.com

Отличная современная справка по C++ и C, с хорошими примерами:

Russian:

<https://ru.cppreference.com/w/>

English:

<https://en.cppreference.com/w/>

C++ reference

C++98, C++03, C++11, C++14, C++17, C++20, C++23

Compiler support (11, 14, 17, 20)
Freestanding implementations

Language

Basic concepts
Keywords
Preprocessor
Expressions
Declaration
Initialization
Functions
Statements
Classes
Overloading
Templates
Exceptions

Headers

Named requirements

Feature test macros (C++20)

Language support library

Type support — traits (C++11)
Program utilities
Relational comparators (C++20)
numeric_limits — type_info
initializer_list (C++11)

Concepts library (C++20)

Diagnostics library

General utilities library

Smart pointers and allocators
unique_ptr (C++11) — shared_ptr (C++11)
Date and time
Function objects — hash (C++11)
String conversions (C++17)
Utility functions
pair — tuple (C++11)
optional (C++17) — any (C++17)
variant (C++17) — format (C++20)

Strings library

basic_string
basic_string_view (C++17)
Null-terminated strings:
byte — multibyte — wide

Containers library

array (C++11) — vector
map — unordered_map (C++11)
priority_queue — span (C++20)

Other containers:

sequence — associative
unordered associative — adaptors

Iterators library

Ranges library (C++20)

Algorithms library

Numerics library

Common math functions
Mathematical special functions (C++17)
Numeric algorithms
Pseudo-random number generation
Floating-point environment (C++11)
complex — valarray

Localizations library

Input/output library

Stream-based I/O
Synchronized output (C++20)
I/O manipulators

Filesystem library (C++17)

Regular expressions library (C++11)

basic_regex — algorithms

Atomic operations library (C++11)

atomic — atomic_flag
atomic_ref (C++20)

Thread support library (C++11)

thread — mutex — condition_variable

Technical specifications

Standard library extensions (library fundamentals TS)

resource_adaptor — invocation_type

Standard library extensions v2 (library fundamentals TS v2)

propagate_const — ostream_joiner — randint
observer_ptr — detection idiom

Standard library extensions v3 (library fundamentals TS v3)

scope_exit — scope_fail — scope_success — unique_resource

Concurrency library extensions (concurrency TS) — Transactional Memory (TM TS)

Concepts (concepts TS) — Ranges (ranges TS) — Reflection (reflection TS)

External Links — Non-ANSI/ISO Libraries — Index — std Symbol Index

Домашнее задание

1. Написать функцию которая выводит массив `double` чисел на экран. Параметры функции это сам массив и его размер. Вызвать эту функцию из `main`.
2. Задать целочисленный массив, состоящий из элементов 0 и 1. Например: `[1, 1, 0, 0, 1, 0, 1, 1, 0, 0]`. Написать функцию, заменяющую в принятом массиве 0 на 1, 1 на 0. Выводить на экран массив до изменений и после.
3. Задать пустой целочисленный массив размером 8. Написать функцию, которая **с помощью цикла** заполнит его значениями 1 4 7 10 13 16 19 22. Вывести массив на экран.

Для продвинутых:

4. * Написать функцию, которой на вход подаётся одномерный массив и число `n` (может быть положительным, или отрицательным), при этом метод должен циклически сместить все элементы массива на `n` позиций.
5. ** Написать функцию, которой передается не пустой одномерный целочисленный массив, она должна вернуть истину если в массиве есть место, в котором сумма левой и правой части массива равны. Примеры: `checkBalance([1, 1, 1, || 2, 1]) → true`, `checkBalance ([2, 1, 1, 2, 1]) → false`, `checkBalance ([10, || 1, 2, 3, 4]) → true`. Абстрактная граница показана символами `||`, эти символы в массив не входят.



Основы C++. Вебинар №5.

Успеха с домашним заданием!



GeekBrains