

Функции в C++

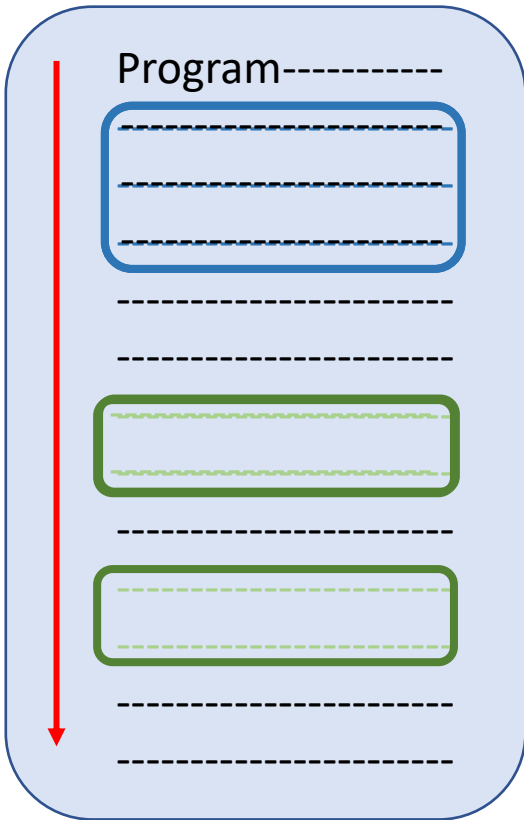
Функция (процедура, метод) –

отдельно оформленный блок программного кода, который

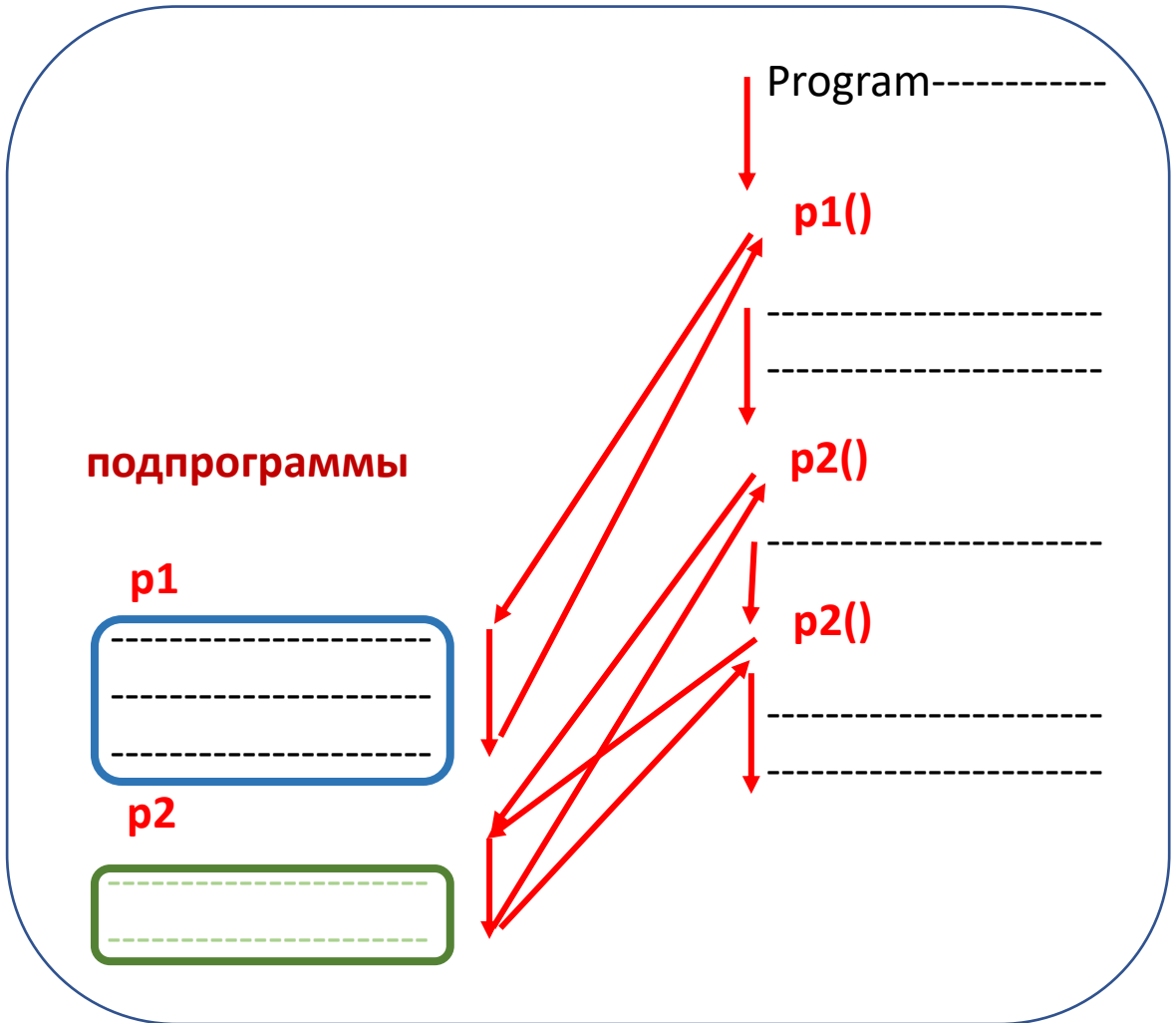
- выполняет некоторое законченное по смыслу действие
- имеет уникальное имя (идентификатор) или его аналог
- состоит из набора инструкций (операторов)
- может быть многократно выполнен (вызван)
в ходе работы программы
- имеет средства для обмена данными с остальным кодом
(параметры, возвращаемый результат)

Процедурный подход

Код без подпрограмм



Код с подпрограммами



Функции в C++

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;

int main() {

    setlocale(0, "");

    double x, y;
    cout << "x = "; cin >> x;
    cout << "y = "; cin >> y;

    double z = sqrt(abs(exp(2.0 * y) + pow(x, 3.0) - 17.0));

    cout << setprecision(5) << z;

    return 0;
}
```

Объявление и определение функций

Объявление функций

тип результата	имя функции	(список формальных параметров);
-------------------	----------------	---------------------------------

- Содержит заголовок (прототип) функции
- Размещается до функции main или в отдельном заголовочном файле

Примеры:

```
double calc(double x, double y);
```

```
void printHello(string name);
```

```
void printAsterisks(int n);
```

Определение функций

```
тип      имя      (список формальных параметров)  {  
результата  функции  
  
    // инструкции тела функции  
}
```

- Содержит заголовок (прототип) функции и блок тела функции
- Размещается после функции main

Примеры:

```
double calc(double x, double y) {  
    return x + y;  
}
```

Определение функций

Примеры:

```
double calc(double x, double y) {  
  
    return x + y;  
}
```

```
void printHello(string name) {  
  
    cout << "Hello, " << name << endl;  
    return;  
}
```

```
void printAsterisks(int n) {  
  
    for (int i = 0; i < n; i++) {  
        cout << "*";  
    }  
    cout << endl;  
}
```


Объявление и определение функций

Функции должны быть объявлены до их вызова

Функции не обязаны быть определены до момента их использования

Объявление и определение обычно делают раздельно, но определение может служить одновременно и объявлением функции

Вызов функций

имя функции	(список фактических параметров)
----------------	---------------------------------

Указание имени функции и списка фактических параметров (аргументов)

- может быть отдельным оператором или использоваться в составе выражения
- передает управление в начало блока инструкций тела функции
- инструкции тела функции выполняются до первого return или до конца их блока
- после чего управление возвращается в точку вызова

```
setlocale(0, "");
```

```
double x, y;  
cout << "x = "; cin >> x;  
cout << "y = "; cin >> y;
```

```
double z = sqrt(abs(exp(2.0 * y) + pow(x, 3.0) - 17.0));
```

```
cout << setprecision(5) << z;
```

Вызов функций

Примеры:

```
#include <iostream>
using namespace std;
```

```
// объявления функций
```

```
double calc(double x, double y);
```

```
void printHello(string name);
```

```
void printAsterisks(int n);
```

```
// определения функций
```

```
double calc(double x, double y) {
    return x + y;
}
```

```
void printHello(string name) {
    cout << "Hello, " << name << endl;
    return;
}
```

```
void printAsterisks(int n) {
    for (int i = 0; i < n; i++) {
        cout << "*";
    }
    cout << endl;
}
```

```
int main() {
```

```
    printHello("");
```

```
    printHello("World");
```

```
    printAsterisks(30);
```

```
    printAsterisks(0);
```

```
    printAsterisks(5);
```

```
    cout << calc(4.0, 5.0);
```

```
    double z = calc(3.0, 10.0) * calc(4.0, 6.0);
```

```
    calc(100.0, 100500.0);
```

```
    return 0;
```

```
}
```

Оператор return

- может быть записан в любом месте тела функции;
- может быть записан несколько раз,
но выполняться будет только один раз для каждого вызова
- останавливает выполнение инструкций тела функции
- передает управление в точку вызова
- возвращает в точку вызова значение выражения, указанного в return
- имеет две формы записи:

```
return;
```

Если тип результата `void` и оператор `return` не указан, то будут выполнены все инструкции тела метода, до конца

```
return выражение;
```

Если тип результата функции отличен от `void`, то `return` обязателен. Тип выражения должен совпадать или приводиться к типу возвращаемого результата из заголовка функции

Параметры функций

Формальные параметры – указываются в заголовке функции при объявлении и определении

```
double calc(double x, double y) {  
    return x + y;  
}
```

Фактические параметры (аргументы) – указываются при вызове функции

```
cout << calc(4.0, 5.0);
```

```
double z = calc(3.0, 10.0) * calc(4.0, 6.0);
```

```
calc(100.0, 100500.0);
```

Аргументы по умолчанию

В заголовке, при объявлении, после имени формального параметра можно указать знак = и значение подходящего типа

Тогда, если при вызове фактический аргумент не указывать, то будет использовано это значение

Такие параметры должны быть в конце списка формальных параметров

```
double cost(double price, double count, double discount = 0);
```

```
int main() {  
    cout << cost(100, 5, 25) << endl; //полный список аргументов, скидка 25%  
    cout << cost(100, 5) << endl;    //скидка 0% по умолчанию  
    return 0;  
}
```

```
double cost(double price, double count, double discount) {  
    return price * count * (1 - discount / 100.0);  
}
```

Передача параметров по значению

- в списке формальных параметров указывают только

тип данных имя параметра

```
double calc(double x, double y);
```

- фактическим значением параметра при вызове может быть любое **выражение** подходящего типа

```
double a = -2.5;
```

```
int b = 5;
```

```
double z = calc(3.5, 10) * calc(a, b) * calc(2 * a + 4, b - 14);
```

- при выполнении тела считается что в нем создается локальная переменная с именем параметра и его типом и инициализируется значением выражения, указанного как фактический аргумент при вызове;

в теле функции эта переменная может использоваться, в том числе получать новые значения, но после его завершения она уничтожается, все ее локальные значения теряются

Передача параметров по значению

```
void calc_value(int, int);
```

```
int main()  
{
```

```
    int a = 4;
```

```
    int b = 5;
```

```
    cout << "Before: a = " << a << "\tb=" << b << endl;
```

```
    calc_value(a, b);
```

```
    cout << "After: a = " << a << "\tb=" << b << endl;
```

```
    return 0;
```

```
}
```

```
void calcValue(int a, int b)
```

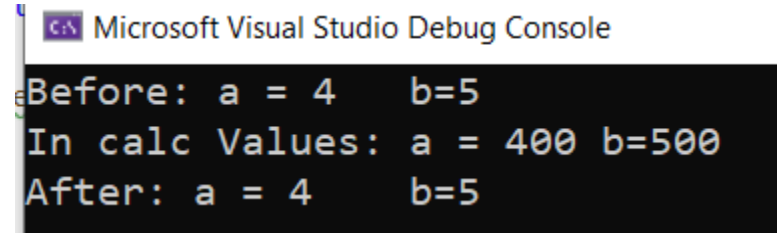
```
{
```

```
    a = 100 * a;
```

```
    b = 100 * b;
```

```
    cout << "In calc Values: a = " << a << "\tb=" << b << endl;
```

```
}
```



Передача параметров по ссылке

- в списке формальных параметров указывают тип данных & имя параметра

```
double calc(double& x, double& y);
```

- фактическим значением параметра при вызове должна быть переменная подходящего типа

```
double a = -2.5;
```

```
int b = 5;
```

```
double z = calc(3.5, 10) * calc(a, b) * calc(2 * a + 4, b - 14);
```

- при вызове функция получает ссылку (прямой доступ) на переменную, указанную как фактический аргумент;
в теле функции эта переменная может использоваться, и **может** получить новое значение;
после завершения работы метода она будет хранить последнее значение, полученное в теле метода и оно может быть использовано дальше, вне метода

Передача параметров по ссылке

```
void calc_reference(int&, int&);
```

```
int main()  
{
```

```
    int a = 4;
```

```
    int b = 5;
```

```
    cout << "Before: a = " << a << "\tb=" << b << endl;
```

```
    calc_reference(a, b);
```

```
    cout << "After: a = " << a << "\tb=" << b << endl;
```

```
    return 0;
```

```
}
```

```
void calc_reference(int& a, int& b)
```

```
{
```

```
    a = 100 * a;
```

```
    b = 100 * b;
```

```
    cout << "In calc References: a = " << a << "\tb=" << b << endl;
```

```
}
```

 Microsoft Visual Studio Debug Console

```
Before: a = 4    b=5
```

```
In calc References: a = 400    b=500
```

```
After: a = 400    b=500
```

Константные параметры

модификатор **const** запрещает изменение параметров в теле функции

```
void calc_const_value(const int a, const int b)
{
    a = 100 * a;
    b = 100 * b;
    cout << "In calc Values: a = " << a << "\tb=" << b << endl;
}
```

```
void calc_const_reference(const int& a, const int& b)
{
    a = 100 * a;
    b = 100 * b;
    cout << "In calc References: a = " << a << "\tb=" << b << endl;
}
```

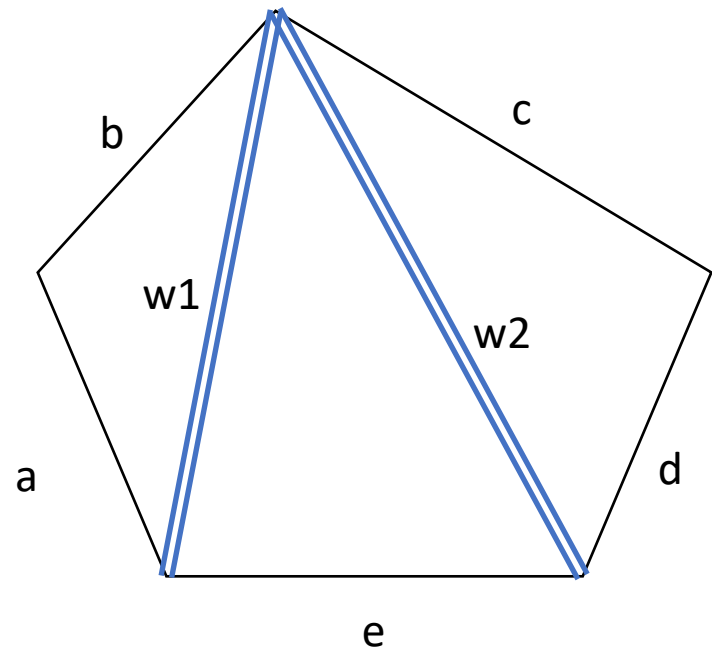
Пример

Для прямоугольного участка земли известны длины его границ a , b , c , d , e .
Через участок проходят две дороги, их длины $w1$ и $w2$.

Хозяину надо заплатить налог, по ставке x рублей за 1 квадратный метр

Хозяину надо построить забор, он стоит y рублей за 1 погонный метр.
Дороги должны остаться проезжими, т.е. забор ставится с двух сторон от дороги.
Шириной дороги можно пренебречь.

Сколько денег надо хозяину?



```
#include <iostream>
using namespace std;
```

```
double area(double a, double b, double c);
```

```
double perimeter(double a, double b, double c);
```

```
bool is_triangle(double a, double b, double c);
```

```
void read_bounds(double& a, double& b, double& c,  
                 double& d, double& e);
```

```
int main() {
    setlocale(0, "");

    double a, b, c, d, e;
    read_bounds(a, b, c, d, e);

    double w1, w2;
    cout << "Введите длины дорог\n";
    cout << "w1 = ";      cin >> w1;
    cout << "w2 = ";      cin >> w2;

    double x, y;
    cout << "Введите ставку земельного налога, руб./1 кв.м \n";
    cin >> x;
    cout << "Введите цену строительства забора, руб./1 м \n";
    cin >> y;

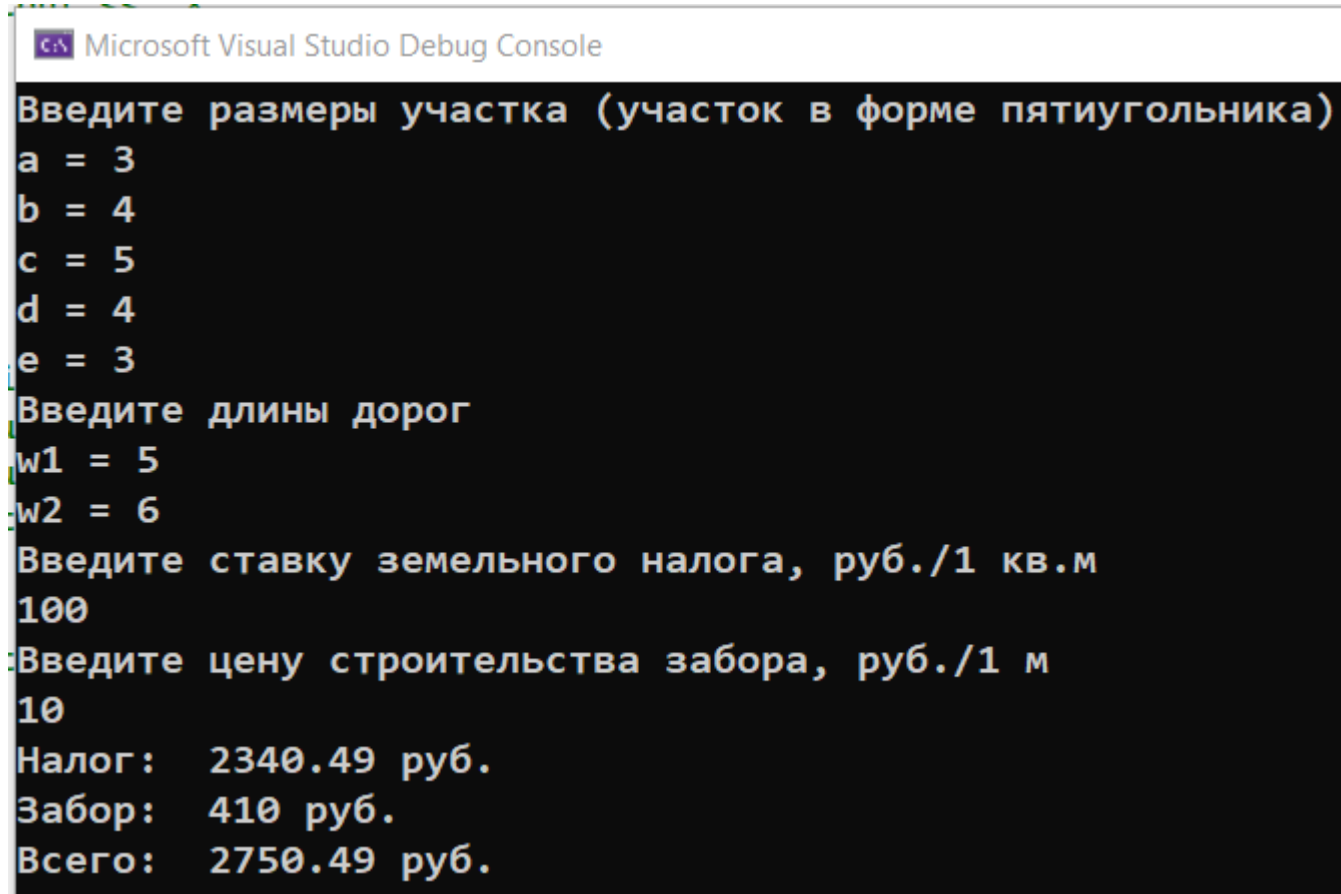
    if (is_triangle(a, b, w1) && is_triangle(c, d, w2) && is_triangle(e, w1, w2))
    {
        double s = area(a, b, w1) + area(c, d, w2) + area(e, w1, w2);
        double p = perimeter(a, b, w1) + perimeter(c, d, w2) + perimeter(e, w1, w2);
        double tax = s * x;
        double fence = p * y;
        cout << "Налог: " << tax << " руб." << endl;
        cout << "Забор: " << fence << " руб." << endl;
        cout << "Всего: " << tax + fence << " руб." << endl;
    }
    else {
        cout << "Расчет невозможен. Вы ввели неверные данные";
    }
    return 0;
}
```

```
double area(double a, double b, double c)
{
    double p = perimeter(a, b, c) / 2.0;
    return sqrt(p * (p - a) * (p - b) * (p - c));
}
```

```
double perimeter(double a, double b, double c)
{
    return a + b + c;
}
```

```
bool is_triangle(double a, double b, double c)
{
    return a > 0 && b > 0 && c > 0 && a + b > c && a + c > b && b + c > a;
}
```

```
void read_bounds(double& a, double& b, double& c, double& d, double& e)
{
    cout << "Введите размеры участка (участок в форме пятиугольника)\n";
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;
    cout << "d = ";
    cin >> d;
    cout << "e = ";
    cin >> e;
}
```



Microsoft Visual Studio Debug Console

```
Введите размеры участка (участок в форме пятиугольника)
a = 3
b = 4
c = 5
d = 4
e = 3
Введите длины дорог
w1 = 5
w2 = 6
Введите ставку земельного налога, руб./1 кв.м
100
Введите цену строительства забора, руб./1 м
10
Налог: 2340.49 руб.
Забор: 410 руб.
Всего: 2750.49 руб.
```



Указатели и массивы в функциях

Указатели как параметры функций

Для параметров-указателей адреса передаются по значению, функция получает и использует локальную копию указателя.

```
void newPointer(int* pa) {  
    pa = new int;  
    *pa = 1000000;  
}
```

Изменения указателя
действительны только «внутри»
функции




!!! освобождение памяти?

Но эта копия указателя хранит то же адрес, что оригинальный указатель. Поэтому через параметры-указатели можно получить прямой доступ к данным, на которые он указывает и изменить их

```
void newData(int* pa) {  
    *pa = 2 * *pa;  
}
```

Изменения в данных переданы
«наружу» по адресу фактического
аргумента



Указатели как параметры функций

```
void newPointer(int* );  
void newData(int* pa);
```

```
int main()  
{
```

```
    int a = 1;  
    int* pa = &a;  
    newPointer(pa);  
    cout << "After newPoint \tpa = " << pa << "\tdata(*pa) = " << *pa << endl;
```

```
    newData(pa);  
    cout << "After newData \tpa = " << pa << "\tdata(*pa) = " << *pa << endl;  
    return 0;
```

```
}
```

```
void newPointer(int* pa) {  
    cout << "\nIn newPointer \tpa = " << pa << "\tdata(*pa) = " << *pa << " = old\n";  
    pa = new int;  
    *pa = 1000000;  
    cout << "In newPointer \tpa = " << pa << "\tdata(*pa) = " << *pa << " = new\n";  
    delete pa;  
}
```

```
void newData(int* pa) {  
    *pa = 2 * *pa;  
    cout << "\nIn newData \tpa = " << pa << "\tdata(*a) = " << *pa << endl;  
}
```

```
In newPointer    pa = 0000008419EFF524    data(*pa) = 1 = old  
In newPointer    pa = 0000021BEE390850    data(*pa) = 1000000 = new  
After newPoint    pa = 0000008419EFF524    data(*pa) = 1  
  
In newData       pa = 0000008419EFF524    data(*a) = 2  
After newData     pa = 0000008419EFF524    data(*pa) = 2
```

Параметры-указатели и параметры-ссылки

```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
int main()
{
    int a = 5, b = 7;
    swap(&a, &b);
    cout << a << b;
    return 0;
}
```

```
void swap(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
int main()
{
    int a = 5, b = 7;
    swap(a, b);
    cout << a << b;
    return 0;
}
```

Одномерные массивы как параметры функции

Параметры-массивы могут быть использованы только для встроенных массивов

```
void writeA(int a[], int n) {  
  
    //size_t length = sizeof(arr) / sizeof(int); // нет, а приводится к int*  
  
    for (int i = 0; i < n; i++) {  
        cout << a[i] << " ";  
    }  
  
    cout << endl;  
}
```

Одномерные массивы как параметры функции

```
void writeA(int a[], int n) {  
  
    for (int i = 0; i < n; i++) {  
        cout << a[i] << " ";  
    }  
    cout << endl;  
}
```

```
void readA(int a[], int n) {  
    cout << "Введите массив из " << n << " элементов";  
    for (int i = 0; i < n; i++) {  
        cin >> a[i];  
    }  
}
```

```
int main() {  
  
    int a[] { 4, -5, 1, 3, -7, 9 };  
    writeA(a, sizeof(a) / sizeof(int));  
  
    int b[5];  
    readA(b, std::size(b));  
    writeA(b, std::size(b));    //...
```

Параметры-указатели для передачи массивов

Параметры-указатели могут быть использованы как для встроенных, так и для динамических массивов

```
void writeArr(int* a, int n) {  
    for (int i = 0; i < n; i++) {  
        cout << a[i] << " ";  
    }  
    cout << endl;  
}
```

Параметры-указатели для передачи массивов

```
void writeArr(int* arr, int n) {  
    for (int i = 0; i < n; i++) {  
        cout << arr[i] << " ";  
    }  
    cout << endl;  
}
```

```
void readArr(int* arr, int n) {  
    cout << "Введите массив из " << n << " элементов" << endl;  
    for (int i = 0; i < n; i++) {  
        cin >> arr[i];  
    }  
}
```

```
int main() {  
    int a[] { 4, -5, 1, 3, -7, 9 };  
    int n = 4;  
    int* b = new int[n];  
  
    readArr(b, n);  
    writeArr(b, n);  
  
    readArr(a, std::size(a));  
    writeArr(a, std::size(a));    //...    delete[] b;
```


Многомерные встроенные массивы как параметры функции

Для параметров многомерных массивов все размерности кроме первой должны быть указаны явно, в форме константного выражения

```
void writeMultDA(int arr[][3], int n) {  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < 3; j++) {  
            cout << arr[i][j] << " ";  
        }  
        cout << endl;  
    }  
}
```

```
int main() {  
  
    int a[2][3] = { {1, 2, 3}, {4, 5, 6} };  
    int b[4][3] = { {1, 2, 3}, {4, 5, 6}, {1, 2, 3}, {4, 5, 6} };  
    writeMultDA(a, 2);  
    writeMultDA(b, 4);  
  
    int c[2][5] = { {1, 2, 3, -3, -1}, {4, 5, 6, -2, -9} };  
    writeMultDA(c, 2); // тип c не соответствует  
    //...
```

Параметры-указатели для многомерных массивов

Многомерные динамические массивы передаются через соответствующие параметры типа указателей

```
void writeMultDArr(int** arr, int n, int m) {  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < m; j++) {  
            cout << arr[i][j] << " ";  
        }  
        cout << endl;  
    }  
}
```

```
int main() {  
  
    int a[2][3] = { {1, 2, 3}, {4, 5, 6} };  
  
    int n = 3, m = 4;  
    int** d = new int* [n];  
    for (int i = 0; i < n; i++) {  
        d[i] = new int[m];  
        for (int j = 0; j < m; j++) {  
            d[i][j] = rand() % 100;  
        }  
    }  
    writeMultDArr(d, 2, 3);  
  
    writeMultDArr(a, 2, 3); // тип данных для массива a не подходит
```

Перегрузка функций

Перегрузка функций – это создание нескольких функций с одним именем, но с разными параметрами.

Под разными параметрами понимают, что должно быть разным *количество аргументов функции и/или их тип*.

То есть перегрузка функций позволяет определять несколько функций с одним и тем же именем и типом возвращаемого значения.

Перегрузка функций также называется *полиморфизмом функций*.

"Поли" означает много, "морфе" – форма, то есть полиморфическая функция – это функция, отличающаяся многообразием форм.

Перегрузка функций

```
#include <iostream>
using namespace std;

int sum(int a, int b);
int sum(int a, int b, int c);

int main() {

    cout << "200 + 801=" << sum(200, 801) << "\n";
    cout << "100 + 201 + 700=" << sum(100, 201, 700) << "\n";

    return 0;
}

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

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

Перегрузка функций

```
#include <string>
#include <iostream>
using namespace std;
```

```
double mult(double a, double b);
double mult(double a, double b, double c);
string mult(string a, int n);
```

```
int main() {
    cout << mult(2, 3) << endl;
    cout << mult(2, 3, 5) << endl;
    cout << mult("Hello! ", 5) << endl;
    return 0;
}
```

```
double mult(double a, double b) {
    return a * b;
}
```

```
double mult(double a, double b, double c) {
    return a * b * c;
}
```

```
string mult(string a, int n) {
    string res;
    res.reserve(a.size() * n);
    while (n--)
        res += a;
    return res;
}
```

Microsoft Visual Studio Debug Console

```
6
30
Hello! Hello! Hello! Hello! Hello!
```

Проблемы перегрузки функций

...

```
int sum(int a, double b);
```

```
int sum(double a, int b);
```

```
int main() {
```

```
    cout << "200+801=" << sum(200, 801) << "\n";
```

```
    cout << "100+201+700=" << sum(100, sum(201, 700)) << "\n";
```

```
    cout << "200+801=" << sum(200, 801.005) << "\n";
```

```
    cout << "100+201+700=" << sum(100.009, 201) << "\n";
```

```
    return 0;
```

```
}
```

```
int sum(int a, double b) {
```

```
    return a + b ;
```

```
}
```

```
int sum(double a, int b) {
```

```
    return a + b;
```

```
}
```

Переменное количество параметров

```
#include <cstdlib>
#include <iostream>
using namespace std;

double avr(string title, int n, ...) // n – количество параметров в ...
{
    double result = 0.0;

    va_list arg;          // указатель на список аргументов
    va_start(arg, n);     // установить указатель на первый аргумент

    for (int i = 0; i < n; i++)
    {
        result += va_arg(arg, int); // получить значение текущего параметра типа int
    }
    va_end(arg); // завершаем обработку параметров

    result /= n;
    cout << title << " " << result;

    return result;
}

int main(void)
{
    cout << avr("Вася", 4, 2, 2, 3, 5) << endl;
    cout << avr("Петя", 7, 2, 2, 3, 4, 5, 1, 5) << endl;
    return 0;
}
```

Переменное количество параметров, тип параметров

```
#include <cstdlib>
#include <iostream>
using namespace std;

double avr(string title, int n, ...) // n - количество параметров в ...
{
    setlocale(0, "");
    double result = 0.0;

    va_list arg;          // указатель на список аргументов
    va_start(arg, n);     // устанавливаем указатель на первый аргумент

    for (int i = 0; i < n; i++)
    {
        result += va_arg(arg, double); // получаем значение текущего параметра типа double
    }
    va_end(arg); // завершаем обработку параметров

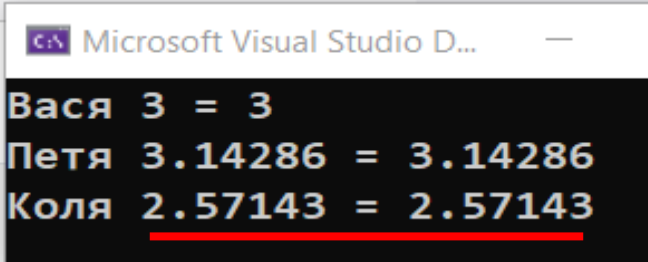
    result /= n;
    cout << title << " " << result << " = ";

    return result;
}

int main(void)
{
    cout << avr("Вася", 4, 2.0, 2.0, 3.0, 5.0) << endl;

    cout << avr("Петя", 7, 2.0, 2.0, 3.0, 4.0, 5.0, 1.0, 5.0) << endl;

    // неверно, размещены аргументы типа int;
    // будет неправильно интерпретироваться как double
    cout << avr("Коля", 7, 2, 2, 3, 4, 5, 1, 5) << endl;
    return 0;
}
```



```
Microsoft Visual Studio D...
Вася 3 = 3
Петя 3.14286 = 3.14286
Коля 2.57143 = 2.57143
```


Стек вызовов

Пример:

```
void f1() {  
    int x = 5;  
    cout << "f1 : x = " << x << endl;  
}
```

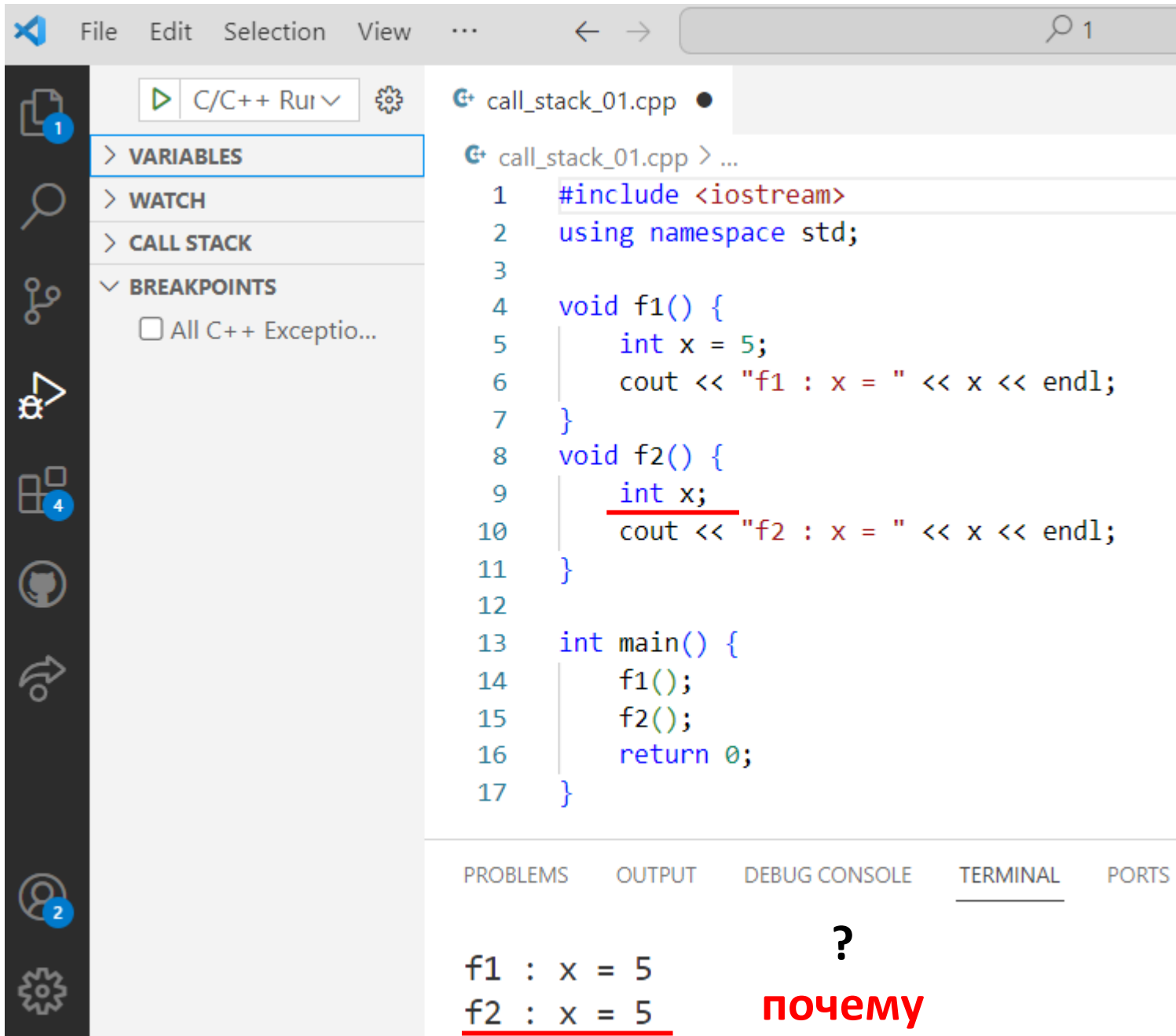
```
void f2() {  
    int x;  
    cout << "f2 : x = " << x << endl;  
}
```

```
int main() {  
    f1();  
    f2();  
    return 0;  
}
```

?

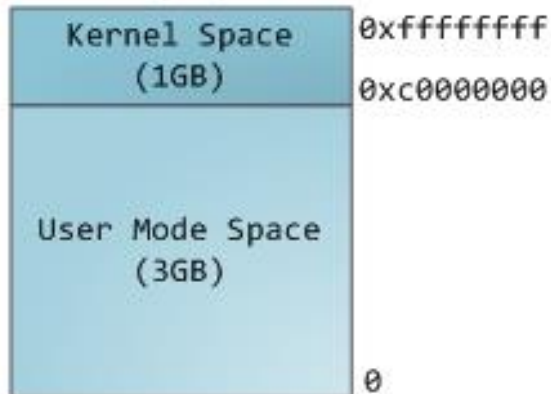
что будет выведено

Стек вызовов

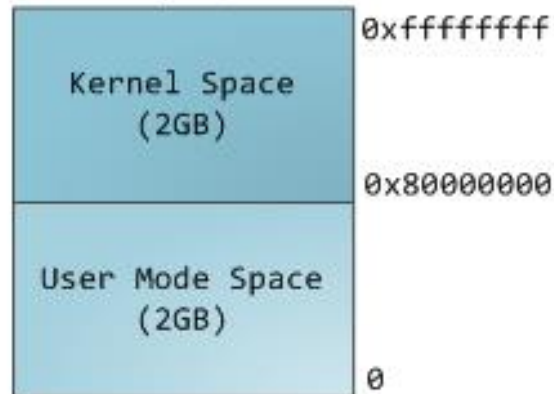


Память процесса, виртуальное адресное пространство

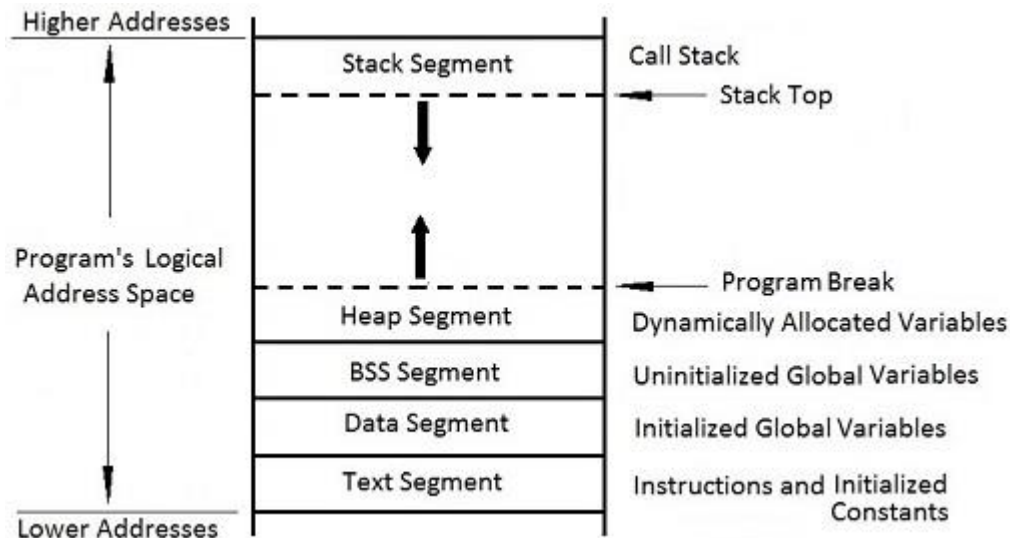
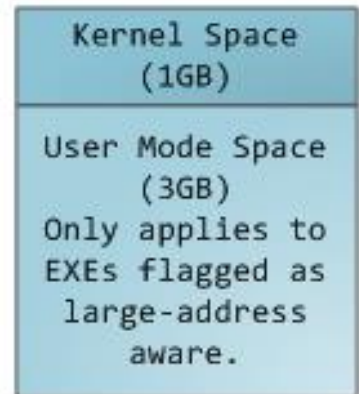
Linux User/Kernel
Memory Split



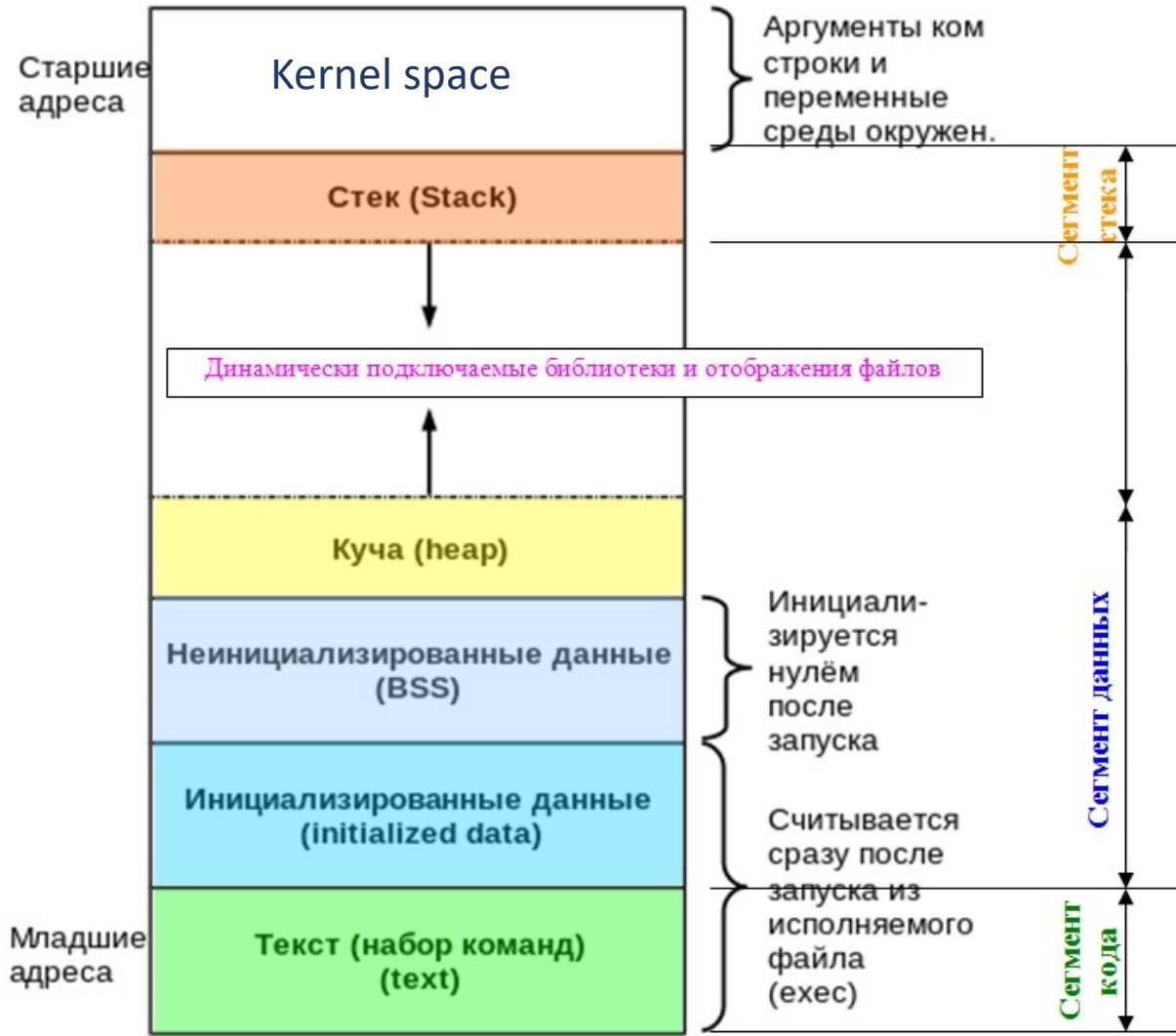
Windows, default
memory split



Windows booted
with /3GB switch

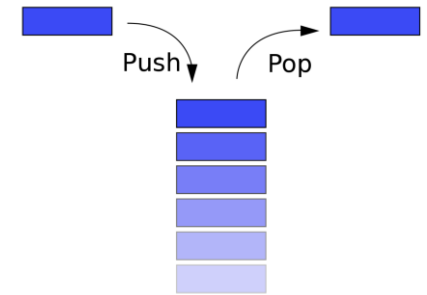
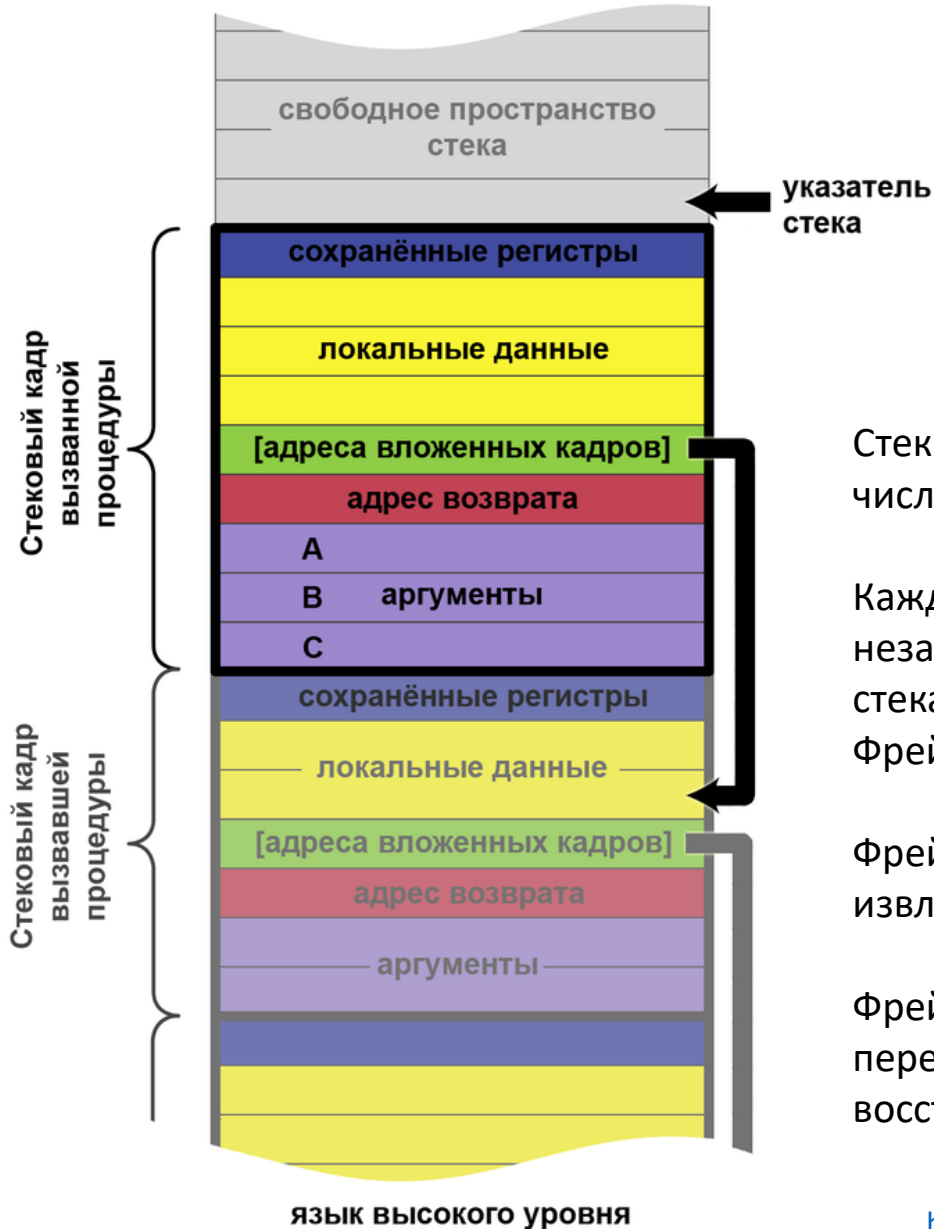


Память процесса, виртуальное адресное пространство



Сегмент – область памяти с определенным назначением, внутри которой поддерживается линейная адресация

Стек, стек вызовов



Стек хранит информацию о каждом вызове, в том числе и вызовы вложенных функций.

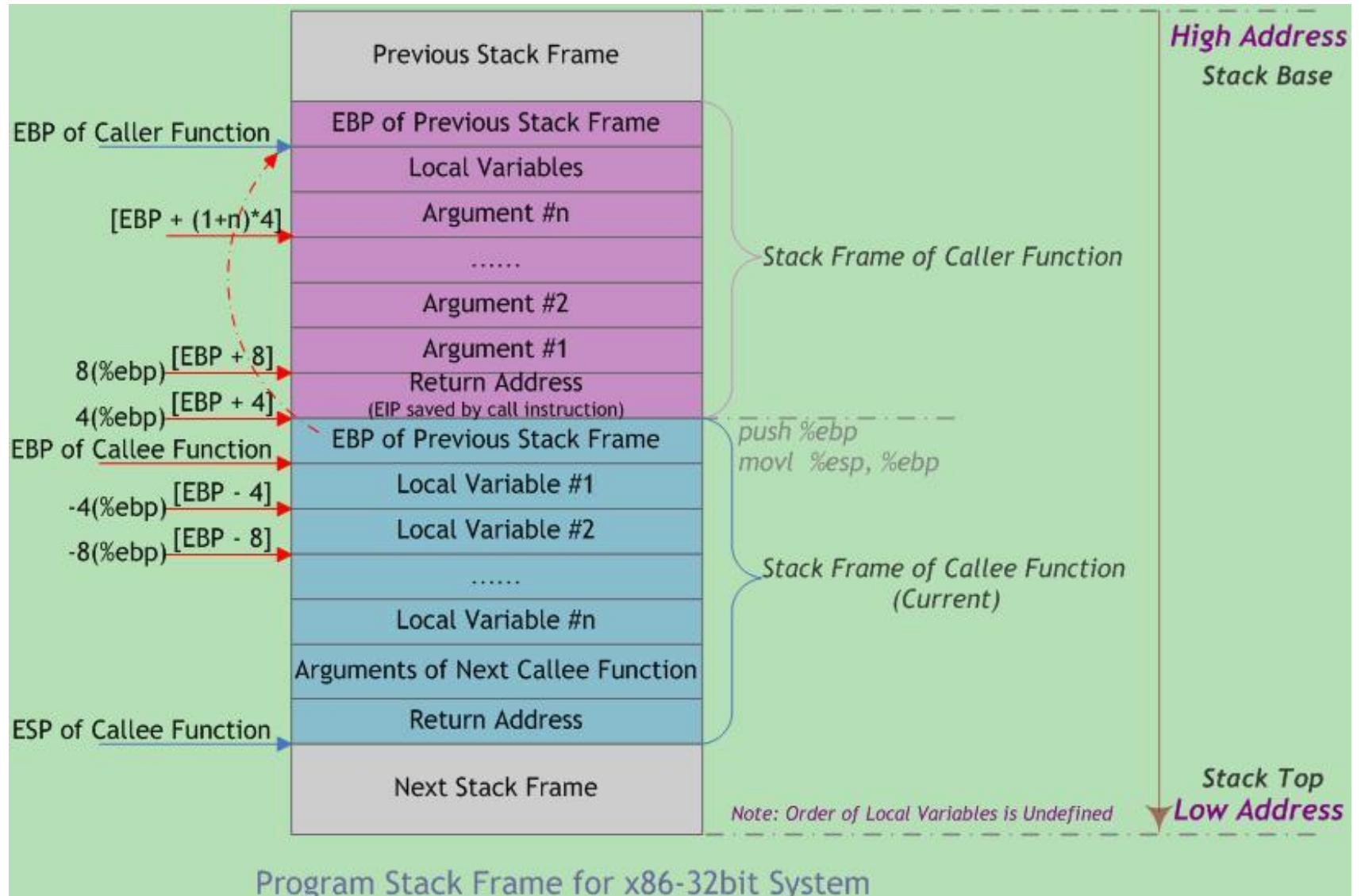
Каждая незавершенная функция занимает независимую непрерывную область – **фрейм (кадр)** стека.

Фрейм стека - это логический фрагмент стека.

Фрейм помещается в стек при вызове функции и извлекается из стека при возврате функции.

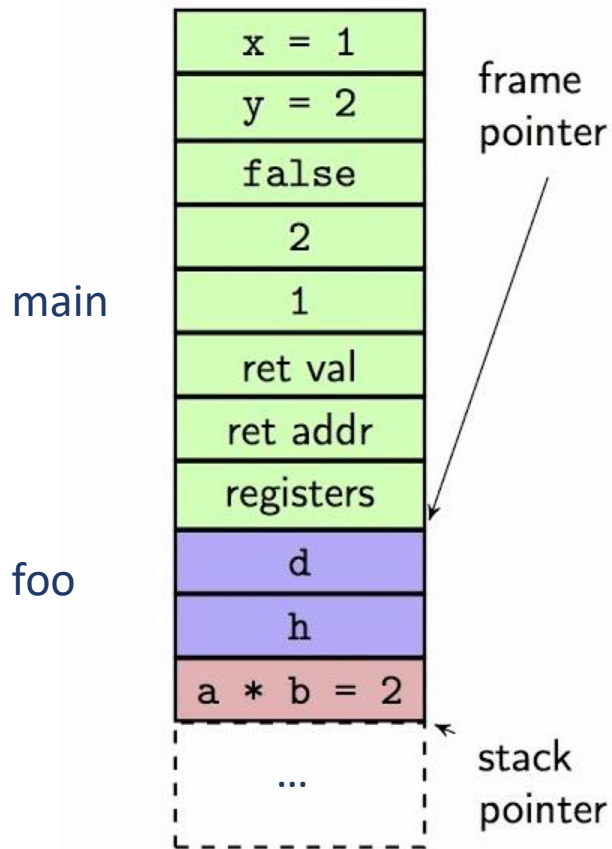
Фрейм хранит параметры функции, локальные переменные и данные, необходимые для восстановления предыдущего кадра стека.

Стек вызовов, структура фрейма



Стек, стек вызовов

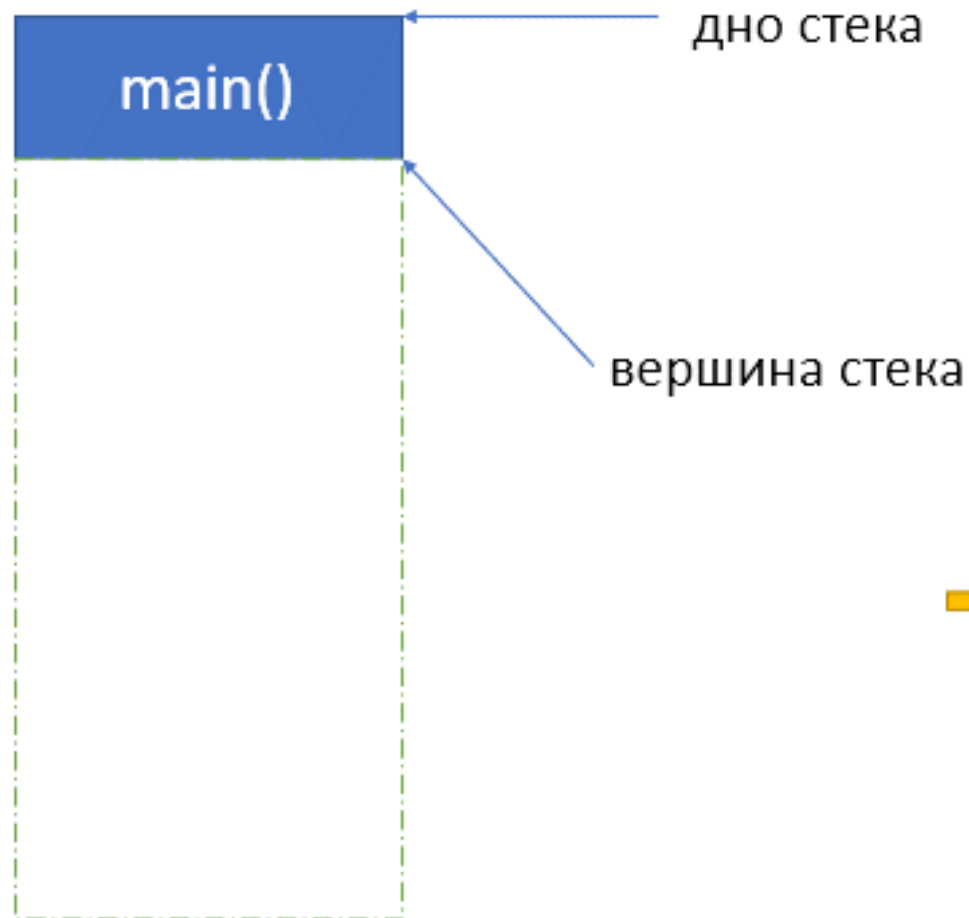
Вызов функции



```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```


Стек, стек вызовов



```
void bar () {  
    int c;  
}  
void foo () {  
    int b = 3;  
    bar();  
}  
→ int main () {  
    int a = 3;  
    foo();  
    bar();  
    return 0;  
}
```

Рекурсивные функции

```
#include <iostream>
using namespace std;
```

```
void f();
```

```
int main() {
```

```
    f(); // не рекурсивный вызов
    return 0;
```

```
}
```

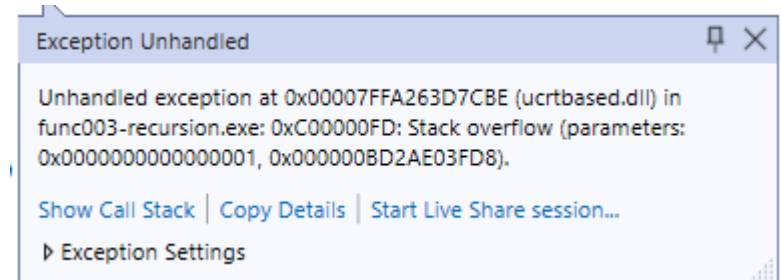
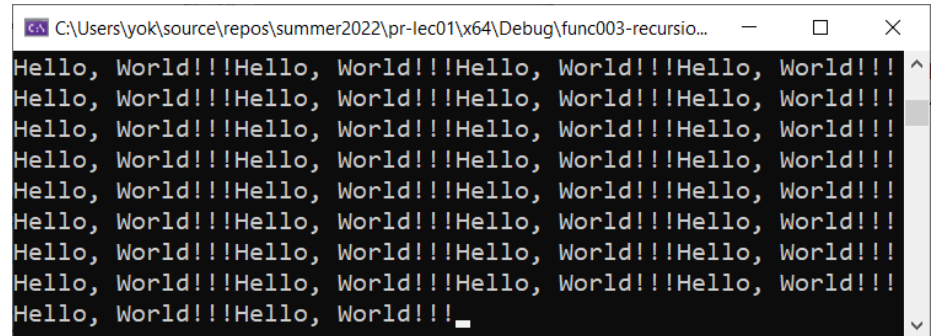
```
void f() {
```

```
    cout << "Hello, World!!!";
```

```
//...
```

```
    f(); // рекурсивный вызов
```

```
}
```



Рекурсивные алгоритмы и рекурсивные функции

Рекурсия в программировании – это пошаговое разбиение задачи на подзадачи, подобные исходной

Функция называется **рекурсивной**, если в своем теле она содержит обращение к самой себе явно или не явно.

Такое обращение называется **рекурсивным вызовом**

При этом количество обращений должно быть конечно

Бесконечная рекурсия – последовательность рекурсивных вызовов, неограниченное число раз выполняющая новые рекурсивные вызовы; на практике **не допускается**

Обычный прием обеспечения конечности – изменять аргументы в каждом рекурсивном вызове, так чтобы решение сводилось к базовому случаю, когда ответ очевиден и рекурсивный вызов не нужен.

Базис рекурсии – условие выхода из блока рекурсивных вызовов;
– базисное решение задачи, когда нет необходимости вызывать рекурсию

Терминальная ветвь – ветвь кода рекурсивной функции, завершающая его без дальнейших рекурсивных вызовов

Шаг рекурсии – вызов функцией самой себя при изменении параметров

```
#include <iostream>
using namespace std;

void f(int);

int main() {
    f(5); // не рекурсивный вызов

    return 0;
}
```

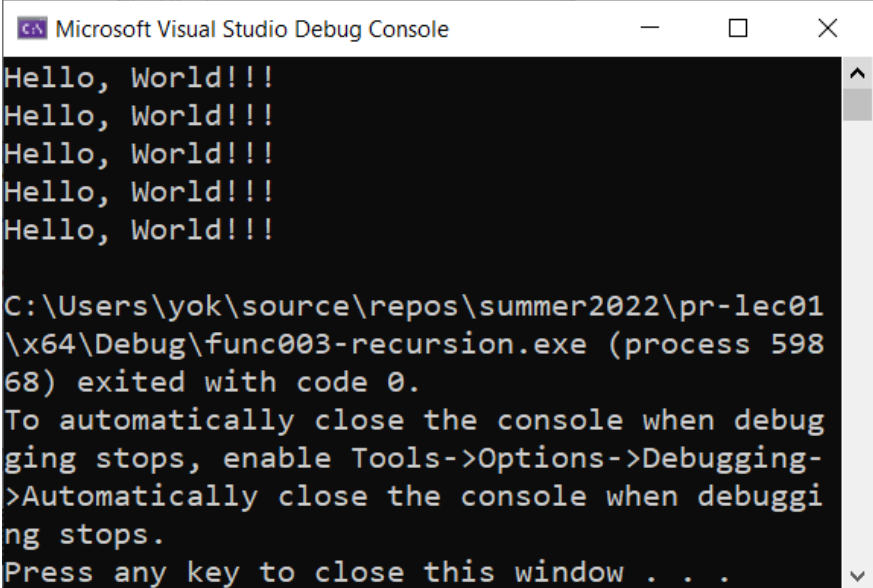
```
void f(int n) {

    if (n <= 0) return; // база рекурсии

    cout << "Hello, World!!!\n";

    f(n-1); // рекурсивный вызов

}
```



The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio Debug Console". The console output displays five lines of "Hello, World!!!". Below the output, the path to the executable is shown: "C:\Users\yok\source\repos\summer2022\pr-1ec01\x64\Debug\func003-recursion.exe (process 59868) exited with code 0." A message follows: "To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops." The window ends with the prompt "Press any key to close this window . . .".

Рекурсивное вычисление факториала

```
#include <iostream>
using namespace std;

int factorial(int);

int main() {
    int num = 5;
    cout << factorial(num);
    return 0;
}

int factorial(int n) {

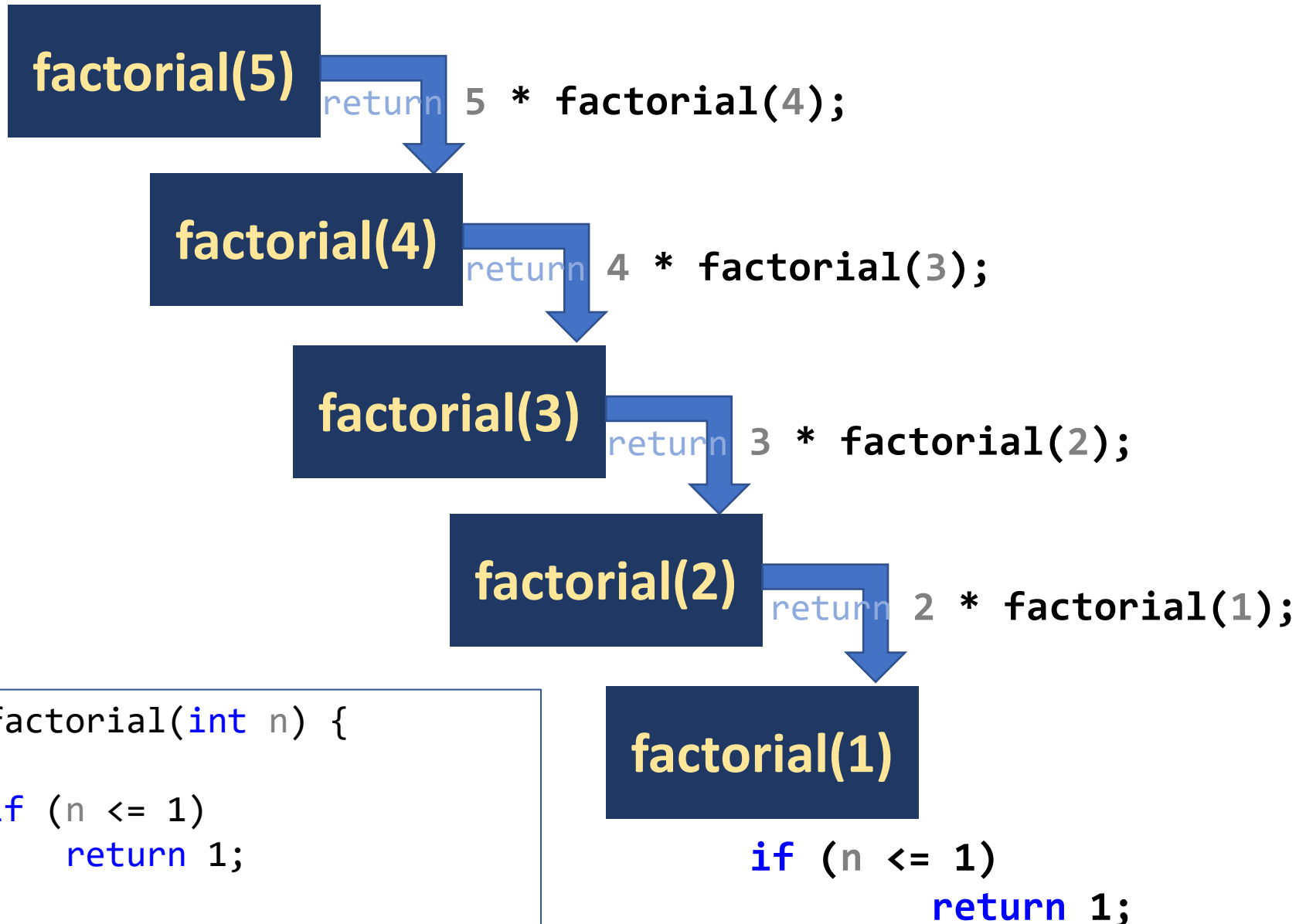
    if (n <= 1)
        return 1;

    return n * factorial(n-1);

}
```

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0. \end{cases}$$

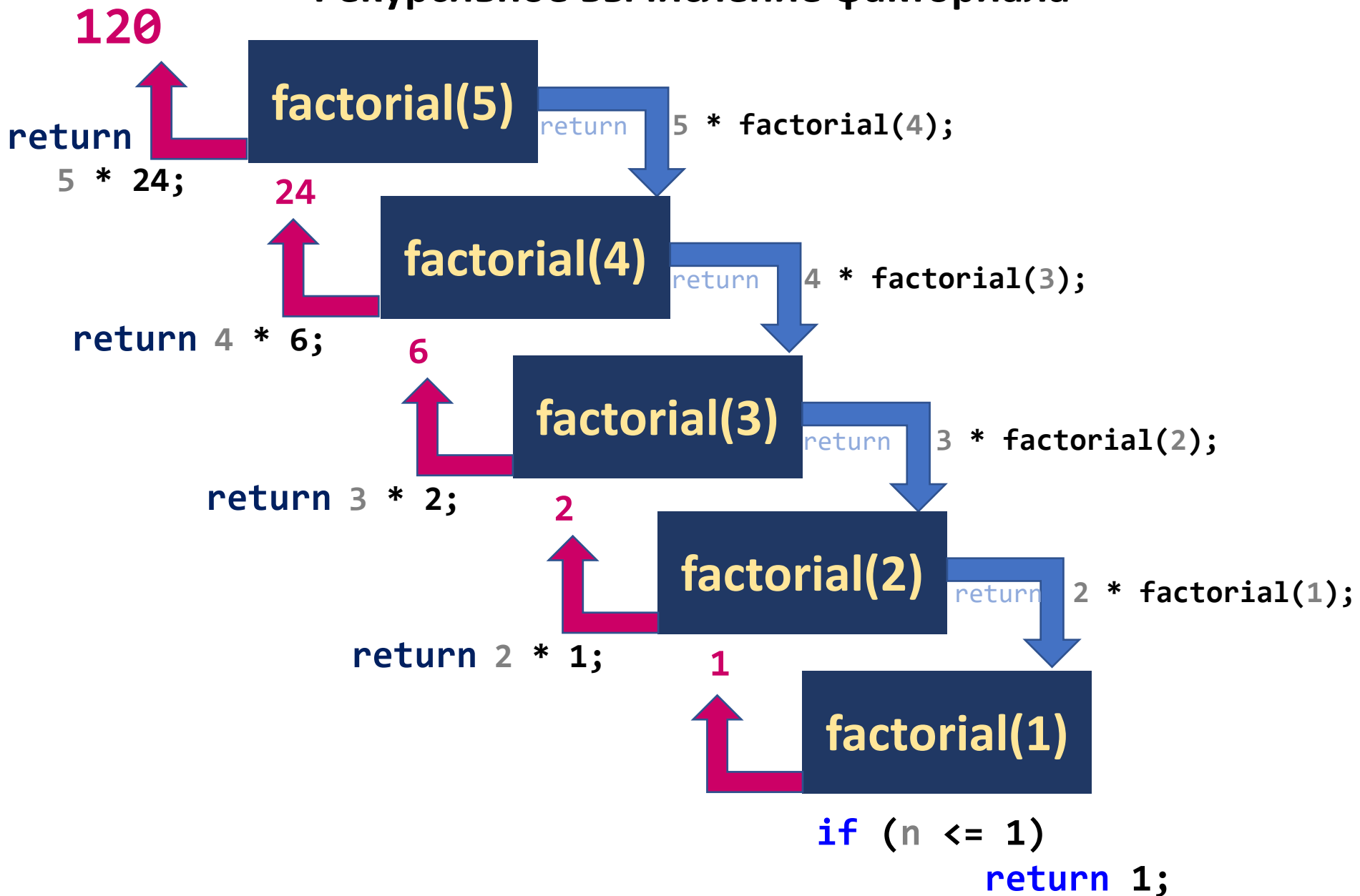
Рекурсивное вычисление факториала



```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    return n * factorial(n-1);  
}
```

```
if (n <= 1)  
    return 1;
```

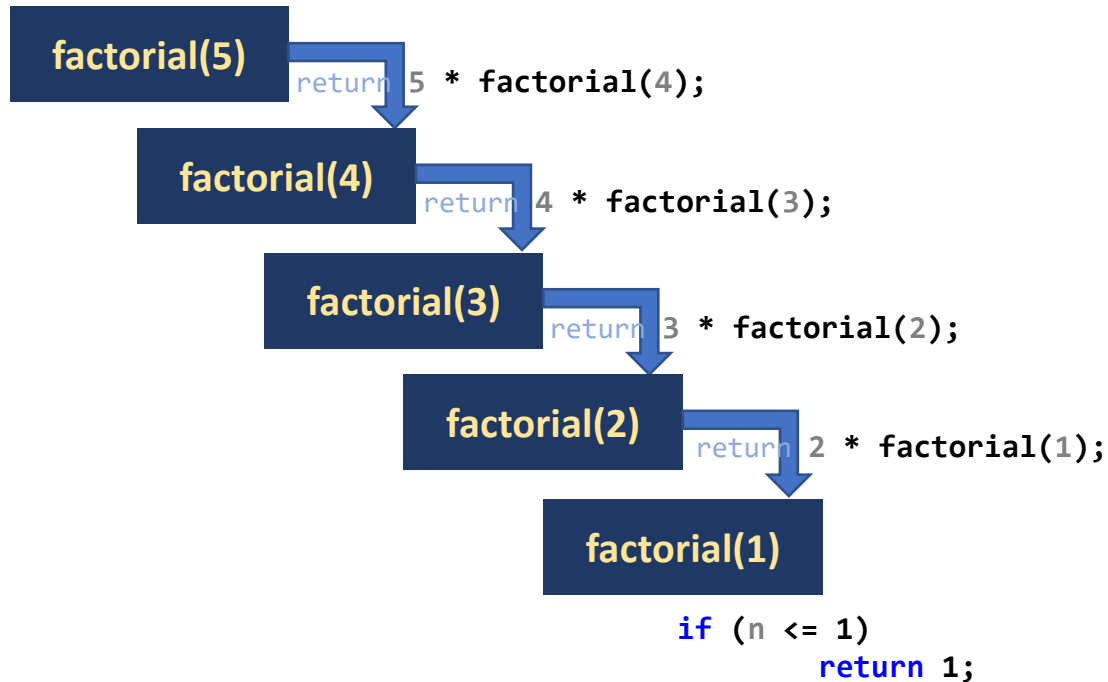
Рекурсивное вычисление факториала



Рекурсивное вычисление факториала –

СТЕК ВЫЗОВОВ

```
int main() {  
    int num = 5;  
    cout << factorial(num);  
    return 0;  
}  
  
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    return n * factorial(n-1);  
}
```



main()
...
num = 5
...
factorial(5)
...
n = 5
...
factorial(4)
...
n = 4
...
factorial(3)
...
n = 3
...
factorial(2)
...
n = 2
...
factorial(1)
...
n = 1
...

Виды рекурсии

- Прямая рекурсия (direct recursion)
 - головная (не хвостовая) рекурсия
 - концевая (хвостовая) рекурсия
 - параллельная рекурсия (рекурсия по дереву)
- Косвенная рекурсия (indirect recursion)

Головная (не хвостовая) рекурсия

```
void f(int n) {  
    if (n <= 0) return; // база рекурсии  
  
    cout << "Hello, World!!!\n";  
  
    f(n-1); // рекурсивный вызов  
  
    cout << "Hello, ВАСЯ!!!\n";  
}
```

После рекурсивного
вызова в функции
выполняются еще
какие-нибудь действия

Концевая (хвостовая) рекурсия

```
void f(int n) {  
  
    if (n <= 0) return; // база рекурсии  
  
    cout << "Hello, World!!!\n";  
  
    f(n-1); // рекурсивный вызов  
  
}
```

Рекурсивный
вызов - последняя
инструкция в
функции

Головная (не хвостовая) рекурсия

```
void f(int n) {  
    if (n <= 0) return; // база рекурсии  
  
    cout << "Hello, World!!!\n";  
  
    f(n-1); // рекурсивный вызов  
  
    cout << "Hello, ВАСЯ!!!\n";  
}
```

После рекурсивного
вызова в функции
выполняются еще
какие-нибудь действия

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    return n * factorial(n-1);  
}
```

После рекурсивного
вызова в функции
выполняются еще
какие-нибудь действия

Концевая (хвостовая) рекурсия

Оптимизация хвостовой рекурсии, или *оптимизация хвостового вызова* — преобразование транслятором хвостового вызова функции в линейный (циклический) код

```
void print_tail(int n)
{
    if (n < 0) return;
    cout << " " << n;

    print_tail(n - 1);
}
```

Рекурсивный вызов
- последняя
инструкция в
функции

```
void print_tail(int n)
{
    start:
    if (n < 0) return;
    cout << " " << n;
    n--;
    goto start;
}
```

Легко оптимизируется
преобразованием в цикл;
меньше места в стеке,
быстрее

Рекурсивное вычисление чисел Фибоначчи

0 1 1 2 3 5 8 13 21 34 55 89 144 ...

```
#include <iostream>
using namespace std;
```

```
int fibonacci(int);
```

```
int main() {
```

```
    cout << fibonacci(8);
    return 0;
```

```
}
```

```
int fibonacci(int n)
```

```
{
```

```
    if (n == 0)
        return 0; // базовый случай
```

```
    if (n == 1)
        return 1; // базовый случай
```

```
    return fibonacci(n - 1) + fibonacci(n - 2);
```

```
}
```

$$F(n) = \begin{cases} 0 & \text{если } n = 0 \\ 1 & \text{если } n = 1 \\ f(n-1) + f(n-2) & \text{если } n > 1 \end{cases}$$

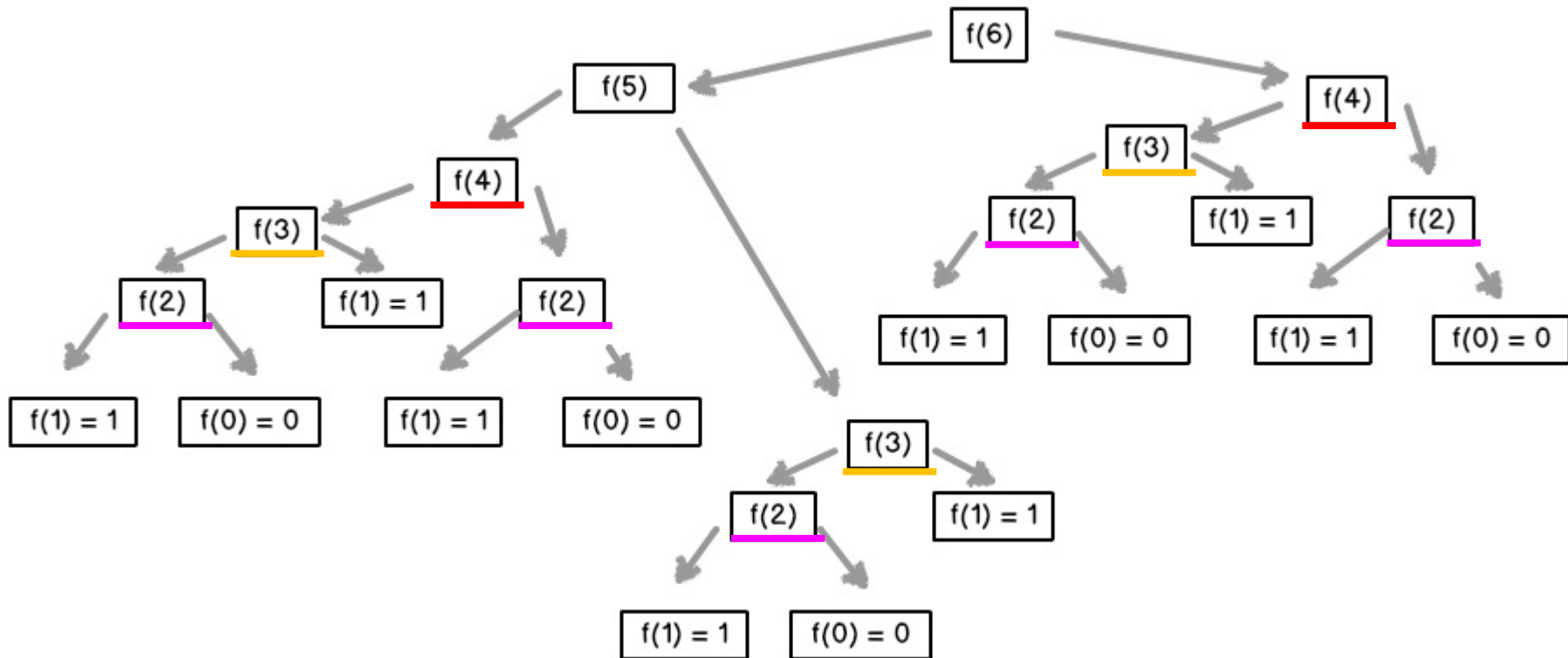
Рекурсивное вычисление чисел Фибоначчи

```
int f(int n)
{
    if (n == 0)
        return 0; // базовый случай
    if (n <= 1)
        return n; // базовый случай

    return f(n - 1) + f(n - 2);
}
```

0 1 1 2 3 5 8 13 21 34 55 89 144 ...

Проблема:
многократное
повторение одних и тех
же вычислений



Косвенная рекурсия

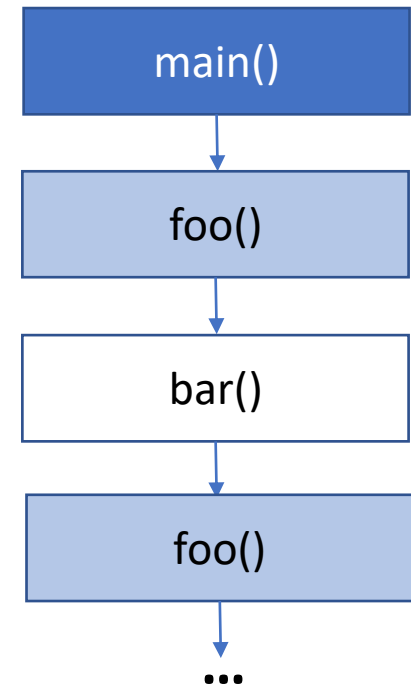
```
#include <iostream>
using namespace std;

void foo();
void bar();

int main() {
    foo();
    return 0;
}

void foo() {
    cout << "Hello from foo \n";
    bar();
}

void bar() {
    cout << "Hello from bar \n";
    foo();
}
```



```
C:\_d_08_2...
Hello from foo
Hello from bar
Hello from foo
Hello from bar
Hello from foo
```

Exception Unhandled

Unhandled exception at 0x00007FF95E8B5308 (KernelBase.dll) in Call_stack_01.exe: 0xC00000FD: Stack overflow (parameters: 0x0000000000000001, 0x00000004092273FF8).

[Show Call Stack](#) | [Copy Details](#) | [Start Live Share session...](#)

► [Exception Settings](#)

Косвенная рекурсия

```
#include <iostream>
#include <conio.h>
using namespace std;

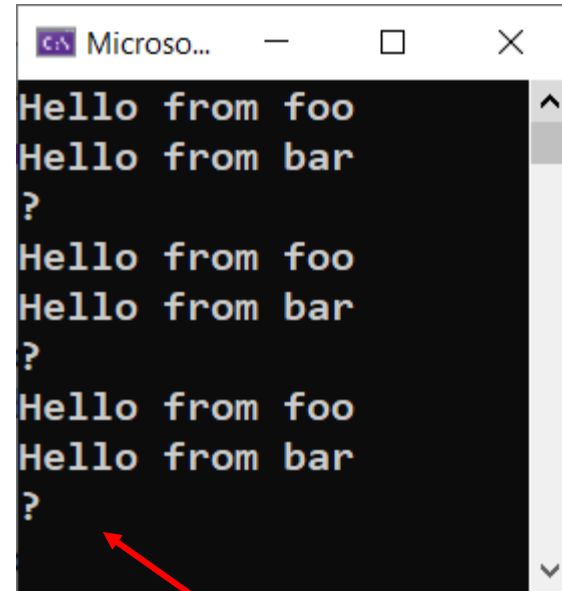
void foo();
void bar();

int main() {
    foo();
    return 0;
}

void foo() {
    cout << "Hello from foo \n";
    bar();
}

void bar() {
    cout << "Hello from bar \n";

    cout << "\?\n";
    if (getch() == 27) return; // выход при нажатии
    foo();
}
```



```
Microso...
Hello from foo
Hello from bar
?
Hello from foo
Hello from bar
?
Hello from foo
Hello from bar
?
```

Press ESC

Рекурсивные функции для массивов

Линейный поиск в одномерном массиве, итеративная реализация

```
int linear_search(int key, int* a, int n) {  
    for (int i = 0; i < n; i++)  
    {  
        if (key == a[i])  
            return i;  
    }  
    return -1;  
}
```

вернуть индекс элемента key в массиве a;
или -1, если key в массиве отсутствует

```
int main() {  
    int a[5]{ 2, 3, 7, 12, 25 };  
    int res_index = linear_search(12, a, size(a));  
    if (res_index > 0) {  
        cout << "Число 12 есть в массиве a под индексом " << res_index << endl;  
    } else {  
        cout << "Нет такого элемента в массиве a \n";  
    }  
    //...
```

Линейный поиск в одномерном массиве, рекурсивная реализация

```
int linear_search_req(int key, int* a, int n) {
```

```
    if (n < 1) {  
        return -1;  
    }
```

```
    if (key == a[n - 1]) {  
        return n;  
    }
```

пытаемся найти значение key в последнем из доступных сейчас элементов массива a

```
    return linear_search_req(key, a, n - 1);  
}
```

в следующий раз будет рассматриваться тот же массив, но без еще одного элемента в его «хвосте»

```
int main() {
```

```
    int a[5]{ 2, 3, 7, 12, 25 };
```

```
    int res_index = linear_search_req(12, a, size(a));
```

```
    if (res_index > 0) {
```

```
        cout << "Число 12 есть в массиве a под индексом " << res_index << endl;
```

```
    } else {
```

```
        cout << "Нет такого элемента в массиве a \n";
```

```
    }
```

```
//
```

Бинарный поиск в одномерном отсортированном массиве, рекурсивная реализация

```
int main() {  
    //!!! отсортировано  
    int a[] { 2, 5, 10, 12, 15, 20, 25, 31, 40 };  
  
    cout << binarySearch(12, a, 0, size(a) - 1);  
  
    return 0;  
}
```

Получить индекс
ищмого элемента
в массиве
или -1, если его в массиве нет

if, (number to search == middle), then we found the element.

2	5	10	12	15	20	25	31	40
0	1	2	3	4	5	6	7	8

if, (number to search < middle), then number to search will be on left sub array.

2	5	10	12	15	20	25	31	40
0	1	2	3	4	5	6	7	8

←

if, (number to search > middle), then number to search will be on Right sub array.

2	5	10	12	15	20	25	31	40
0	1	2	3	4	5	6	7	8

→

Бинарный поиск в одномерном отсортированном массиве, рекурсивная реализация

```
int binary_search(int key, int* a, int left, int right) {  
  
    if (left > right) return -1;  
  
    int middle = left + (right - left) / 2;  
  
    if (key == a[middle]) {  
        return middle;  
    }  
  
    if (key < a[middle]) {  
        return binary_search(key, a, left, middle - 1);  
    }  
    else {  
        return binary_search(key, a, middle + 1, right);  
    }  
}
```

if, (number to search == middle), then we found the element.

2	5	10	12	15	20	25	31	40
0	1	2	3	4	5	6	7	8

if, (number to search < middle), then number to search will be on left sub array.

2	5	10	12	15	20	25	31	40
0	1	2	3	4	5	6	7	8

←

if, (number to search > middle), then number to search will be on Right sub array.

2	5	10	12	15	20	25	31	40
0	1	2	3	4	5	6	7	8

→