**ЛЕКЦИЯ 6. ФУНКЦИИ. ПЕРЕГРУЗКА И ШАБЛОНЫ**

Содержание

[1. ФУНКЦИИ, ВОЗВРАЩАЮЩИЕ БОЛЬШЕ ОДНОГО ЗНАЧЕНИЯ 1](#_Toc178946147)

[1.1 Использование параметров функции 1](#_Toc178946148)

[1.2 Использование pair и tuple 2](#_Toc178946149)

[2. СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ В ФУНКЦИИ 3](#_Toc178946150)

[3. ФУНКЦИИ C ПЕРЕМЕННЫМ ЧИСЛОМ ПАРАМЕТРОВ 4](#_Toc178946151)

[3.1 Список параметров с указанием их количества 4](#_Toc178946152)

[3.2 Использование макрокоманд 5](#_Toc178946153)

[3.3 Функции c переменным числом параметров разных типов 6](#_Toc178946154)

[4. УКАЗАТЕЛЬ И МАССИВ УКАЗАТЕЛЕЙ НА ФУНКЦИЮ 7](#_Toc178946155)

[5. ПЕРЕГРУЗКА ФУНКЦИЙ 9](#_Toc178946156)

[5.1 ПАРАМЕТРЫ ПО УМОЛЧАНИЮ 10](#_Toc178946157)

[6. ШАБЛОНЫ ФУНКЦИЙ 11](#_Toc178946158)

[6.1 Понятие и определение шаблона 11](#_Toc178946159)

[6.2 Использование шаблона при обработке массива 12](#_Toc178946160)

# ФУНКЦИИ, ВОЗВРАЩАЮЩИЕ БОЛЬШЕ ОДНОГО ЗНАЧЕНИЯ

## Использование параметров функции

Функция может возвращать несколько значений через параметры функции, передавая их по адресу, используя указатель или ссылку.

Например,

**void add\_dif(int, int, int&, int\*);**

**int main()**

**{**

**int a = 12, b = 5;**

**int add, sub;**

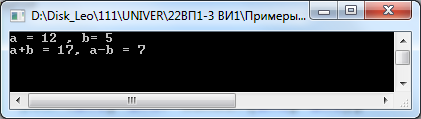
**cout << "a = " << a << " , b= " << b << endl;**

**add\_dif(a, b, add, &sub);**

**cout << "a+b = " << add << ", a-b = " << sub << endl;**

**}**

**void add\_dif(int a, int b, int& add, int\* dif) {**

** add = a + b;**

**\*dif = a - b;**

**}**

## Использование pair и tuple

Начиная с C++ 11:

**#include <iostream>**

**#include <tuple>**

**using namespace std;**

**pair<int, int> sumdif(int, int);**

**tuple<int, int, double> add\_multiply\_divide(int, int);**

**tuple<int, int, int, double> f(int, int);**

**int main() {**

**int a = 12, b = 5;**

**cout << "a = " << a << " , b= " << b << endl;**

**auto p = sumdif(a, b);**

**cout << "pair.a+b = " << p.first << ", pair.a-b = " <<**

**p.second << endl;**

**int add, sub, mult;**

**double div;**

**tie(add, mult, div) = add\_multiply\_divide(a, b);**

**cout << "tie.a+b = " <<add<< " tie.a\*b = " << mult <<**

**" tie.a/b = " << div << endl;**

**tie(add, sub, mult, div) = f(a, b);**

**cout << "tie.a+b = " <<add << " tie.a-b = " << sub <<**

**" tie.a\*b = " << mult << " tie.a/b = " << div << endl;**

**mult = 0;**

**tie(add, sub, ignore, div) = f(a, b);**

**cout <<"tie.a+b = "<< add << " tie.a-b = " << sub <<**

**" ignore = " << mult << " tie.a/b = " << div << endl;**

**return 0;**

**}**

**pair<int, int> sumdif(int a, int b) {**

**return pair<int, int>(a + b, a - b);**

**}**

**tuple<int, int, double> add\_multiply\_divide(int a, int b) {**

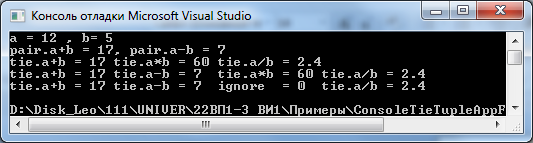
**return make\_tuple(a + b, a \* b, double(a) / double(b));**

**}**

**tuple<int, int, int, double> f(int a, int b) {**

**return { a + b, a - b, a \* b, static\_cast<double>(a) / b };**

**}**



# СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ В ФУНКЦИИ

Пусть создана следующая программа, в которой функция использует **статические переменные total и count.**

**#include <iostream>**

**using namespace std;**

**float average(float);**

**int main(){**

**for(int i=1; i<4; ++i)**

**cout << average(static\_cast<float>(i))<<" ";**

**cout << endl;**

**return 0;**

**}**

**float average(float newdata) {**

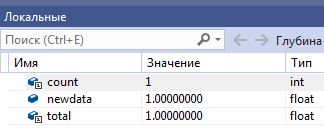
**static float total = 0;**

**static int count = 0;**

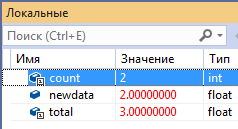
**count++;**

**total += newdata;**

**return total / count;**

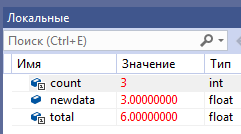
**}**

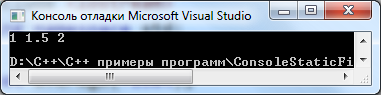
Поставим точку останова

на оператор **return total/count**

и рассмотрим, как меняются переменные

**total и count** при вызове функции.





**Результат:**

# ФУНКЦИИ C ПЕРЕМЕННЫМ ЧИСЛОМ ПАРАМЕТРОВ

## Список параметров с указанием их количества

Если по каким-то причинам предполагается создать функцию с неизвестным заранее числом аргументов, то в объявлении и определении такой функции переменное число аргументов задается многоточием в конце списка формальных параметров или списка типов аргументов

Для того, чтобы получить доступ ко всем параметрам, принимаемых функцией, нужно знать имя и тип параметров первого параметра.

**#include <iostream>**

**#include <stdlib.h> // для system("cls")**

**using namespace std;**

**int sum(int n, ...) //Так объявляется функция с заранее**

**//неизвестнымчислом параметров**

**{**

**int result = 0; //Инициализация значения суммы**

**for(int\* ptr = &n; n > 0; n--)//Пока встречаются параметры**

**{**

**result += \*(++ptr); //Добавляем к сумме значения**

**cout << \*ptr << " "; //отладка**

**}**

**cout << endl;**

**return result;**

**}**

**void main()**

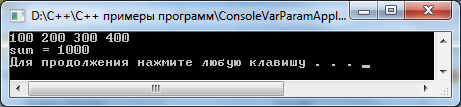
**{**

**//system("cls"); //требует stdlib.h**

**// В функцию sum передается значения данных: 100, 200, 300, 400**

**cout <<"sum = "<< sum(4, 100, 200, 300, 400)<< endl;**

**}**



**ВНИМАНИЕ!!!**

**Все параметры должны иметь тип первого параметра!**

## Использование макрокоманд

При определении функций с переменным количеством параметров, рекомендуется использовать специальный набор макроопределений, которые становятся доступными при включении в программу заголовочного файла **stdarg.h**:

* специальный тип **va\_list** используется для представления списков параметров неизвестной длины и состава;
* **va\_start** вызывается непосредственно перед началом работы с неименованными параметрами;
* **ар** – параметр, который является условным обозначением и относится к макрокомандам, а не функциям. **ар** инициализируется указателем на последний именованный параметр в списке с переменным числом параметров – **lastarg**.

После вызова макроопределения **va\_start**, каждый вызов **va\_arg** возвращает значение заказанного типа t**ype** себе. Надо заранее указать тип желаемого параметра.

Работа со списком параметров завершается вызовом макроопределения **va\_end: void va\_end(ap);** Это макроопределение обеспечивает корректный возврат из функции и вызывается перед выходом из функции.

**#include <stdarg.h>**

**#include <iostream>**

**using namespace std;**

**void test(char p, ...);**

**void main() {**

**test('1','2', '3', '4', '5', '6', '7', '\0');**

**cout << endl;**

**}**

**void test(char p, ...) {**

**char cval = p;**

**va\_list ap; //Указатель на список параметров**

**va\_start(ap, p); //Настроились на список параметров**

**do{**

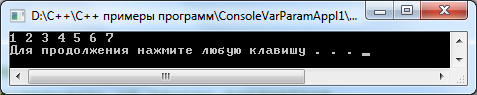
**cout << cval << " ";**

**cval = va\_arg(ap, char);**

**} while (cval != '\0');**

**va\_end(ap); //Завершаем работу с макрокомандами**

**}**



## Функции c переменным числом параметров разных типов

Вообще в функцию можно передавать произвольное число параметров с переменным типом.

**#include <stdarg.h>**

**#include <iostream>**

**using namespace std;**

**void OK(char\* format, ...);**

**void main(){**

**OK(const\_cast<char\*>("%d%f%с"), 8, 9.555, "ABC");**

**}**

**void OK(char\* format, ...) {//Функция OK с произвольным числом**

**//параметров**

**va\_list ap; //Указатель на список параметров**

**va\_start(ap, format); //Настроились на список параметров**

**for (char\* p = format; \*p; p++) {**

**if (\*p == '%') {//Если встретится символ %**

**switch (\*++p) {//То анализируем следующий символ**

**case 'd': {int ival = va\_arg(ap, int); //Если символ d**

**cout << ival << " "; break; //выводим int**

**}**

**case 'f': {double dval = va\_arg(ap, double); //Если f**

**cout << dval << " "; break; //выводим double**

**}**

**case 'с': {char\* cval = va\_arg(ap, char\*); //Если с**

**cout << cval << " "; break; //выводим строку**

**}**

**}**

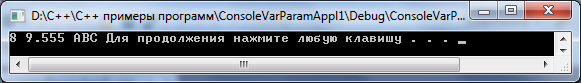
**}**

**else cout << \*p << " ";**

**}**

**va\_end(ap); //Завершаем работу с макрокомандами**

**}**



# УКАЗАТЕЛЬ И МАССИВ УКАЗАТЕЛЕЙ НА ФУНКЦИЮ

Объявление указателя на функцию имеет вид:

**тип результата (\*)(параметры)**

Например, **bool (\*)(int, int)** – указатель на функцию, которая имеет два параметра **int** и возвращает **bool.**

Определение массива таких функций задается следующим образом

**bool (\*имя\_массива[размер массива])(int, int)**

рассмотрим пример сортировки массива по возрастанию и убыванию с использованием указателя на функцию

**//**

**//Сортировка массивов по возрастанию или по убыванию**

**#include <iostream>**

**using namespace std;**

**void bubble(int[], const int, bool (\*)(int, int));//сортировка массива**

**bool ascending(int, int); //сравнение на меньше**

**bool descending(int, int);//сравнение на больше**

**int main() {**

**const int Size = 10;**

**int a[Size] = { 2, 5, 6, 4, 8, 10, 12, 9, 3, 1 };**

**int b[Size] = { 2, 5, 6, 4, 8, 10, 12, 9, 3, 1 };**

**bubble(a, Size, ascending);**

**cout << "Massiv a:" << endl;**

**for (int i = 0; i < Size; ++i)**

**cout << a[i] << " ";**

**cout << endl;**

**bubble(b, Size, descending);**

**cout << "Massiv b: " << endl;**

**for (int i = 0; i < Size; ++i)**

**cout << b[i] << " ";**

**cout << endl << endl;**

**//Массив указателей на функции**

**bool (\*f[2])(int, int) = { ascending, descending };**

**cout << "ascending(10, 5) = " << (\*f[0])(10, 5) << endl;**

**cout << "descending(10, 5)= " << (\*f[1])(10, 5) << endl;**

**return 0;**

**}**

**void bubble(int x[], const int n, bool (\*compare)(int, int)) {**

**void swap(int&, int&); //прототип функции swap**

**for (int i = 1; i < n; ++i)**

**for (int j = 0; j < n - 1; ++j)**

**if ((\*compare)(x[j], x[j + 1]))**

**swap(x[j], x[j + 1]);**

**}**

**void swap(int& elem1, int& elem2) {**

**int tmp = elem1;**

**elem1 = elem2;**

**elem2 = tmp;**

**}**

**bool ascending(int c, int d) {**

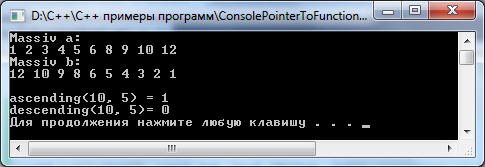
**return d < c;**

**}**

**bool descending(int c, int d) {**

**return d > c;**

**}**



# ПЕРЕГРУЗКА ФУНКЦИЙ

Правила описания перегруженных функций:

* перегруженные функции должны находиться в одной области видимости, иначе произойдет сокрытие аналогично одинаковым именам переменных во вложенных блоках;
* перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию;
* **функции не могут быть перегружены**, если описание их параметров отличается только модификатором **const** или использованием ссылки (например, **int** и **const int** или **int** и **int&**).

**// overload – перегрузка функций**

**#include <iostream>**

**using namespace std;**

**void repchar(); // прототипы**

**void repchar(char);**

**void repchar(char, int);**

**int main() {**

**repchar();**

**repchar('=');**

**repchar('+', 30);**

**return 0;**

**}**

**//функция repchar() - выводит на экран 15 символов '\*'**

**void repchar() {**

**for (int j = 0; j < 15; j++)**

**cout << '\*'; // вывод символа '\*'**

**cout << endl;**

**}**

**//функция repchar() - выводит 15 заданных символов**

**void repchar(char ch) {**

**for (int j = 0; j < 15; j++)**

**cout << ch; // вывод заданного символа**

**cout << endl;**

**}**

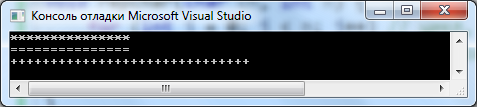
**// функция repchar() - выводит заданный символ заданное число раз**

**void repchar(char ch, int n) {**

**for (int j = 0; j < n; j++) // цикл, выполняющийся n раз**

**cout << ch; // вывод заданного символа**

**cout << endl;**

**}**

В программе содержатся три функции с одинаковым именем. Каждой из функций соответствует свое объявление, определение и вызов. Каким же образом компилятор различает между собой эти три функции? Здесь на помощь приходит сигнатура функции, которая позволяет различать между собой функции по количеству аргументов и их типам.

## ПАРАМЕТРЫ ПО УМОЛЧАНИЮ

Задание значений параметров по умолчанию полезно в тех случаях, когда параметры функции часто принимают какое-то одно значение. Кроме того, значения по умолчанию могут использоваться, если программист хочет модифицировать уже написанную функцию путем добавления в нее нового параметра. В этом случае не нужно будет изменять вызовы функции, поскольку значение нового параметра будет задано по умолчанию.

**#include <iostream>**

**using namespace std;**

**//прототип**

**void repchar(char = '\*', int = 15);**

**int main() {**

**repchar();**

**repchar('=');**

**repchar('+', 30);**

**return 0;**

**}**

**// функция repchar() - выводит заданный символ заданное число раз**

**void repchar(char ch, int n) {**

**for (int j = 0; j < n; j++) // цикл, выполняющийся n раз**

**cout << ch; // вывод заданного символа**

**cout << endl;**

**}**

Результат аналогичен предыдущей версии программы

# ШАБЛОНЫ ФУНКЦИЙ

## Понятие и определение шаблона

С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тип данных передается функции в виде параметра на этапе компиляции. Компилятор автоматически генерирует правильный код, соответствующий переданному типу.

Формат простейшей функции-шаблона:

**template <class Туре> заголовок{//вместо class можно typename**

**/\* тело функции \*/**

**}**

Вместо слова **Туре** может использоваться произвольное имя.

**template < class T1, class T2, class T3>**

**T1 f(T2, T3);**

Вычисления без использования шаблона:

**#include <iostream>**

**using namespace std;**

**int min\_(int, int);**

**double min\_(double, double);**

**int main() {**

**cout << min\_(5, 10 ) << endl;**

**cout << min\_(1.5, 5.2) << endl;**

**return 0;**

**}**

**int min\_(int a, int b) {**

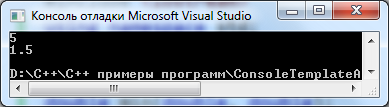
**return (a < b) ? a : b;**

**}**

**double min\_(double a, double b) {**

**return (a < b) ? a : b;**

**}**



Как перейти к шаблону:

берем реализацию функции для какого-то типа;

* приписываем заголовок **template <class Type> или template <typename Type>;**
* В реализации функции имя типа заменяем на **Type.**

**#include <iostream>**

**using namespace std;**

**template <class Type> //можно так: template <typename Type>**

**Type min\_(Type a, Type b);**

**int main()**

**{**

**cout << min\_(5, 10 ) << endl;**

**cout << min\_(1.5, 5.2) << endl;**

**return 0;**

**}**

**template <class Type> //можно так: template <typename Type>**

**Type min\_(Type a, Type b) {**

**return (a < b) ? a : b;**

**}**

Пока нет вызова функции **min\_** при компиляции она в бинарном коде не создается (не *инстанцируется*). При вызове функции с определенными параметрами компилятор создает на основе шаблона функцию с указанным типом параметров.

*Таким образом, шаблон определяет* ***автогенерацию*** *кода функции с определенными параметрами.*

## Использование шаблона при обработке массива

**Пример 1. Шаблон функции сортировки массива**

**#include <iostream>**

**using namespace std;**

**template <typename Т>**

**void sort(Т [], int); //или void sort(T\* b. int n)**

**int main() {**

**const int N = 5; //массив целых чисел**

**int a[N] = { 2, 5, 3, 4, 1 };**

**sort(a, N);**

**for (int i = 0; i < N; ++i)**

**cout << a[i] << " ";**

**cout << endl;**

**char c[] = "dcbaf";//массив символов**

**sort(c, strlen(c));**

**for (int i = 0; i < N; ++i)**

**cout << c[i] << " ";**

**cout << endl;**

**return 0;**

**}**

**template <typename Т>**

**void sort(Т b[], int n) {**

**Т t; //буферная переменная для обмена элементов**

**for (int i = 0; i < n - 1; i++) {**

**int imin = i;**

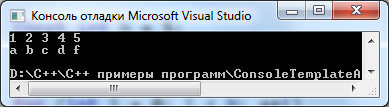
**for (int j = i + 1; j < n; j++)**

**if (b[j] < b[imin]) imin = j;**

**t = b[i]; b[i] = b[imin]; b[imin] = t;**

**}**

**}**



Можно «вручную» задать вариант шаблона функции для работы с целыми числами:

**void sort<int>(int b[], int n){**

**... // Тело специализированного варианта функции**

**}**

**Пример 2. Шаблон функции вычисления максимального и среднего арифметического значения массива**

**#include <iostream>**

**using namespace std;**

**//требуется указание размера массива в вызове функции**

**template <typename T> const T\* max(const T[], unsigned);**

**//не требуется указание его размера массива в вызове функции:**

**//массив передается по ссылке**

**//[std::size\_t](https://en.cppreference.com/w/cpp/types/size_t) – беззнаковый тип данных (минимум 16 бит)**

**template <typename T, size\_t N> T average(const T(&)[N]);**

**int main() {**

**int numbers1[]{ 1, 2, 5, 4, 3 };**

**for (auto& i : numbers1)**

**cout<<i<<"\t";**

**cout << endl;**

**cout << \*max(numbers1, size(numbers1)) << endl; //5**

**cout << average(numbers1) << endl; // 3**

**double numbers2[]{ 1.1, 3.2, 4.3, 5.4, 6.5, 2.6 };**

**for (auto& i : numbers2)**

**cout<<i<<"\t";**

**cout << endl;**

**cout << \*max(numbers2, size(numbers2)) << endl; //6.5**

**cout << average(numbers2) << endl; // 3.85**

**}**

**template <typename T>**

**const T\* max(const T data[], unsigned size)**

**{**

**const T\* result{&data[0]};**

**for (unsigned i =1; i < size; i++) {**

**if ( data[i] > \*result)**

**result = &data[i];**

**}**

**return result;**

**}**

**template <typename T, size\_t N> T average(const T(&data)[N])**

**{**

**T result{};**

**for (unsigned i{}; i < N; i++) {**

**result += data[i];**

**}**

**return result / N;**

**}**

**Пример 3. Шаблон функции «сглаживания» массива**

**#include <iostream>**

**using namespace std;**

**template <class Type>**

**void newMas(Type[], int, Type b[]);**

**int main()**

**{**

**setlocale(LC\_ALL, "Russian");**

**//int d[]{ 2, 3, 3, 6, 5, 8, 7, 1 };**

**double d[]{ 2, 3, 3, 6, 5, 8, 7, 1 };**

**cout << "d: " << endl;**

**for (auto m : d)**

**cout << m << "\t";**

**cout << endl;**

**//int b[size(d)]{};**

**double b[size(d)]{};**

**newMas(d, size(d), b);**

**cout<<"b: "<< endl;**

**for (auto m : b)**

**cout << m << "\t";**

**cout << endl;**

**}**

**template <class Type>**

**void newMas(Type a[], int n, Type b[]) {**

**auto sum3{ [](Type x, Type y, Type z) { return (x + y + z) / 3; } };**

**for (int i{ 1 }; i < n - 1; ++i)**

**b[i] = sum3(a[i - 1], a[i], a[i + 1]);**

**};**

Результат

