

Основы программирования

Лабораторная работа 5

Тема: Особенности работы с функциями

Цель лабораторной работы

Целями данной работы являются получение навыков процедурного программирования на основе создания собственных функции и знаний о методах использования таких функций.

Краткие теоретические сведения

Функция — специальная конструкция, с помощью которой какой-либо фрагмент кода выносится за тело программы (за пределы *main()*).

В языке C/C++ **функция** – это строительный блок всех программ.

Общий вид функции:

```
Возвращаемый_тип Имя_функции(Список_параметров)
{
    Тело функции
}
```

К примеру, функция с двумя формальными параметрами целого типа, возвращающая целое число в качестве результата своей работы:

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

Передача массива в функции

Предположим, что нам нужна функция принимающая массив и выполняющая некоторые действия с ним:

```
int main() {
    int mas[10];
    read_mass(mas);
    return 0;
}
```

Если аргументом функции является *одномерный* массив, то ее формальный параметр можно объявить так:

- как массив фиксированного размера;

```
void read_mass(int m[10]) { // массив фиксированного размера
    ...
}
```

- как массив неопределенного размера.

```
void read_mass(int m[]) { // массив неопределенного размера
    ...
}
```

К примеру, данная функция может быть такой:

```
void read_mass(int m[]){
    for(int i = 0; i < 10; i++)
        cin >> m[i];
}
```

Так как изменения массива внутри функции затрагивают и массив, переданный в качестве аргумента, то с помощью данной функции можно выполнить считывания массива с консоли.

Функция может принимать массивы разных размеров. Для этого при передаче массива необходимо каким-либо образом передавать и **размер массива**. Самым распространенным методом передачи размера массива в функцию является использование дополнительной целочисленной переменной. Для этой цели рекомендуется использовать тип **size_t**:

```
int sum(int arr[], size_t length)
{
    int res = 0;
    for (int i = 0; i < length; i++)
        res += arr[i];
    return res;
}
int main(){
    int mas[10] = {1, 3, 2, 5, 4, 7, 6, 8, 9, 0};
    int s = sum(mas, 10); // вызов функции
    cout << "Sum = " << s << endl;
    return 0;
}
```

КОНСОЛЬ ОТЛА

Sum = 45

Передача двумерного массива

Если в функцию передаётся **двумерный** массив, то описание соответствующего аргумента функции должно содержать количество *столбцов* (вторую размерность).

К примеру, функция, вычисляющая сумму элементов двумерного массива:

```
int summa (int arr[][5], int row, int col)
{
    int res=0;
    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++)
            res +=arr[i][j];
    return res;
}
```

Для корректной работы данной функции нужно передавать двумерный массив со второй размерностью 5:

```
int mas[3][5] = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 0},
    {10,11,12,13,14},
};
int s = summa(mas, 3, 5); // вызов функции
```

```
cout << "Sum = " << s << endl;
```

КОНСОЛЬ ОТЛАДКИ

```
Sum = 105
```

Есть еще один распространенный способ передачи массивов в функции с использованием механизма *указателей*, но его мы рассмотрим позже вместе с этим механизмом.

Типы переменных по области видимости

Глобальные переменные

Глобальные переменные определены в файле программы вне любой из функций и могут использоваться любой функцией. То есть они определены в пределах *глобальной области видимости*. К примеру:

```
void print();
int n = 5; // глобальная переменная
int main()
{
    print(); // n = 5
    n++;
    print(); // n = 6
    return 0;
}
void print()
{
    cout << "n = " << n << endl;
}
```

КОНСОЛЬ

```
n = 5
n = 6
```

Локальные переменные

Переменные, которые создаются внутри блока кода, ограниченного фигурными скобками, называются **локальными**. Такие переменные доступны в пределах только того блока кода, в котором они определены. То есть они существуют в **локальной** области видимости.

```
void print(int);
int main()
{
    int z = 2;
    print(z); // n = 10
    return 0;
}
void print(int x)
{
    int n = 5 * x;
    // z++; так сделать нельзя, так как z определена в функции main
    cout << "n = " << n << endl;
}
```

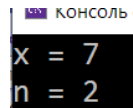
КОНСОЛЬ ОТ

```
n = 10
```

*Параметры функции также являются **локальными переменными**.*

Используя только фигурные скобки можно определить **вложенные** области видимости:

```
int n = 2;
{
    int x = 5 + n;    // x существует только в этом блоке кода
    cout << "x = " << x << endl;
}
cout << "n = " << n << endl;
```



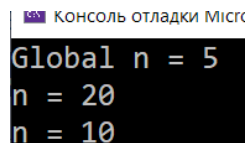
КОНСОЛЬ

x = 7
n = 2

Соккрытие переменных

Если несколько переменных или констант из нескольких пересекающихся областей видимости имеют одно и то же имя, то при обращении к переменной по этому имени происходит обращение к той переменной, которая «ближе» к этой области видимости. Этот эффект называется **сокрытием**. К примеру:

```
int n = 5;
int main()
{
    cout << "Global n = " << n << endl; // n = 5
    int n = 10; // локальная n скрывает глобальную
    {
        int n = 20;
        cout << "n = " << n << endl; // n=20
    }
    cout << "n = " << n << endl; // n=10
    return 0;
}
```



КОНСОЛЬ ОТЛАДКИ MSVC

Global n = 5
n = 20
n = 10

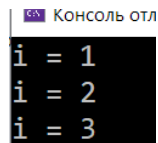
Статические переменные

Иногда нам необходимо сохранять значение локальных переменных функции между ее вызовами. Для этой цели мы можем использовать **статические переменные**. Такие переменные не уничтожаются при выходе за пределы области видимости.

Они определяются на уровне функций с помощью ключевого слова **static**. К примеру:

```
void display();
int main()
{
    display(); // 1
    display(); // 2
    display(); // 3
    return 0;
}
```

```
void display()
{
    static int i = 0;
    i++;
    cout << "i = " << i << endl;
}
```



```
Консоль отл
i = 1
i = 2
i = 3
```

К переменной *i* был добавлено ключевое слово *static*, поэтому при завершении работы функции *display* переменная не уничтожается, а сохраняется в памяти. При последующих вызовах функции *display* значение переменной *i* постепенно увеличивается на 1.

Рекурсивные функции

Функция, которая вызывает саму себя, называется **рекурсивной**.

Простым примером рекурсивной функции является рекурсивное **вычисление факториала**. Для написания рекурсивной функции вычисления факториала, нужно определить из каких составляющих она должна состоять.

Рекурсивная функция состоит из трех основных составляющих:

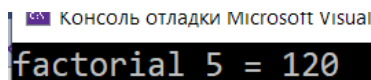
- базис рекурсии – основное тело вычисления;
- шаг рекурсии – изменение аргумента для рекурсивного вызова;
- условие выхода из рекурсии – обязательно должно достигаться на каком-либо шаге рекурсии.

Теперь представим рекурсивное выражение факториала в виде функции:

```
int fact(int N)
{
    if (N < 1) return 0; // на случай 0 или отрицательного числа
    if (N == 1) return 1; // факториал 1 равен 1
    return N * fact(N-1); // вызов себя через формулу N! = N * (N-1)!
}
```

Данная функция корректно работает. К примеру, вызов функции с аргументом 5:

```
int f = fact(5);
cout << "factorial 5 = " << f << endl;
```



```
Консоль отладки Microsoft Visual
factorial 5 = 120
```

Задание

Разработать программу для каждой задачи из **общего** и **индивидуального** задания.

Для решения задач необходимо использовать собственные функции. Функции должны использоваться в основном коде (внутри функции *main()*), чтобы можно было проверить их работоспособность.

Продумайте, какие аргументы должны быть у разрабатываемых функций. Внутри функции **не должно быть ввода с клавиатуры**. Ввод должен осуществляться в функции *main()*, а полученные значения переданы в функцию в качестве аргументов.

Все пользовательские функции должны **только выполнять вычисление и возвращать результат** в главную функцию (если в задаче на сказано иного). Все результаты работы программ необходимо выводить на экран в функции *main()*.

Методические указания

Выполните **общие** и **индивидуальные** задания.

При выполнении **общих** заданий не требуется отчет, но все еще нужна защита с объяснением кода программы.

При выполнении **индивидуальных** заданий необходимо выполнить следующие этапы **для каждой задачи отдельно**:

- 1) словесная постановка задачи;
- 2) анализ задачи и формальная постановка задачи;
- 3) проектирование (разработка алгоритма) – лучше всего в графической форме (блок-схема), но можно и в словесной форме или псевдокод;
- 4) реализация (кодирование, отладка);
- 5) тестирование.

Результаты выполнения индивидуальных заданий оформить в виде отчета.

Общее задание

Напишите две функции:

- Функция должна принимать одномерный целочисленный массив и заполнять его **случайными числами** в диапазоне от 0 до 100.
- Функция должна принимать одномерный целочисленный массив и возвращать значение **максимального** элемента массива.

В функции *main()* создайте одномерный целочисленный массив. После этого заполните его с помощью *первой функции*. После этого выведите массив на экран консоли. Далее вызовите *вторую функцию* для данного массива и выведите результат также на экран.

Варианты индивидуальных заданий

Вариант 1

Напишите функцию, которая имитирует игру «21» на шестигранных кубиках. В функции должна быть **статическая** переменная *sum*, которая представляет собой текущую сумму очков. Изначально она равна 0.

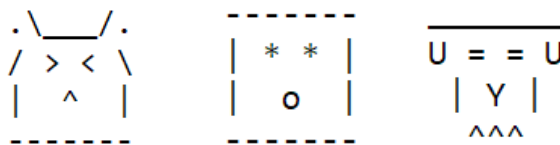
При вызове функции генерируются 2 случайных числа от 1 до 6 (имитация бросков 2-х шестигранных кубиков). Их сумма складывается с переменной *sum* и сохраняется в ней. Если *sum* превышает 21, то она становится равна 0. Функция должна возвращать текущее значение *sum*.

Пользователь должен иметь возможность бесконечно запрашивать вызов функции. После каждого вызова выводите на консоль текущую сумму бросков.

Вариант 2

Напишите функцию, в которой должна быть **статическая переменная** (изначально равная 0). При каждом вызове переменная должна увеличиваться на 1, пока не достигнет значения 3. В этом случае она снова должна сброситься до 0.

В зависимости от текущего значения *статической* переменной на консоль должно выводиться одно из трех изображений. К примеру:



Вариант 3

Напишите функцию, которая принимает массив и целое число. Функция заполняет данный массив переданным числом. При этом переданное число должно сохраняться в **статической переменной** в функции (изначально она равна 0).

Если в функцию в качестве числа для заполнения будет передано 0, то вместо, того чтобы заполнить массив 0, массив должен быть заполнен числом из статической переменной (то есть предыдущем переданным в функцию числом).

Протестируйте работу функции на нескольких массивах и с разными переданными числами (в том числе и 0). Выводите все заполненные массивы для демонстрации работы.

Вариант 4

Напишите функцию, которая внутри себя содержит **статический массив** целых чисел и **статическую переменную**, обозначающую количество заполненных элементов в массиве. Изначально считается, что массив пустой, то есть количество заполненных элементов равно 0.

Функция принимает как аргумент одно число и добавляет его в массив (при первом вызове число запишется в первый элемент массива). После этого выводит массив в консоль. Если

массив будет полностью заполнен, то новое число не добавляется.

Пользователь должен иметь возможность бесконечно вводить числа в консоль. Вводимые числа должны передаваться в функцию.

Вариант 5

Написать функцию, которая принимает в качестве аргумента число N , которое должно быть больше 0. Если это так, то функция генерирует случайное число от 0 до N и возвращает его как результат. Сгенерированное число запоминается в **статической** переменной S . Вначале число $S = 0$.

Если переданное число (аргумент) равен или меньше 0 или же вообще не указан (используйте значение по умолчанию для $N = 0$), то вместо генерации нового числа, должно возвращаться значение из S (предыдущее сгенерированное число).

Пользователь должен иметь возможность бесконечно вводить числа в консоль. Вводимые числа должны передаваться в функцию. Результат функции должен выводиться в консоль.

Вариант 6

Напишите функцию, которая принимает массив целых чисел. Она должна считать, сколько в массиве отрицательных элементов. Это количество нужно складывать в **статическую переменную** в функции (изначально равная 0). То есть эта переменная считает, сколько всего отрицательных элементов во всех переданных в данную функцию массивах.

Функция должна возвращать текущее значение **статической переменной**.

Протестируйте работу функции на нескольких массивах. Выводите результаты, возвращенные из функции.

Вариант 7

Напишите функцию, которая принимает целое число. В функции должна быть **статическая** переменная, в которой сохраняется **максимальное** из переданных в нее чисел.

При первом вызове в *статическую* переменную записывается переданное число независимо от его значения. При последующих вызовах, переданное число сравнивается со статической переменной. Если оно больше, то записывается в *статическую* переменную.

После этого функция возвращает текущее значение *статической* переменной.

Пользователь должен иметь возможность бесконечно вводить числа в консоль. Вводимые числа должны передаваться в функцию. После чего пользователю должен выводиться результат, возвращенный из функции.

Вариант 8

Напишите функцию, которая принимает целое число. В функции должен быть **статический массив**, изначально заполненный нулями.

Переданное число помещается в *случайный нулевой элемент* массива (нужно генерировать

случайное число заново, если попался *ненулевой* элемент). После этого массив выводится в консоль. Если массив будет полностью заполнен, то новое число не добавляется.

Пользователь должен иметь возможность бесконечно вводить числа в консоль. Вводимые числа должны передаваться в функцию.

Вариант 9

Напишите функцию, которая принимает массив целых чисел. Она должна определять является ли массив **возрастающим**, то есть все элементы в нем расположены по возрастанию. Если является, то **статическая переменная** в функции (изначально равная 0) должна увеличиваться на 1. То есть эта переменная считает, сколько раз в функцию был передан возрастающий массив.

Функция должна возвращать текущее значение **статической переменной**.

Протестируйте работу функции на нескольких массивах. Выводите результаты, возвращенные из функции.

Вариант 10

Напишите функцию, которая внутри себя содержит **статический массив** целых чисел, изначально заполненный нулями.

Функция принимает как аргумент одно число. Далее *статический* массив просматривается с начала, пока не будет найдено число, меньшее, чем переданное как аргумент. На его место в массиве записывается переданное число. Если такое число не будет найдено, то массив остается без изменений. После этого массив выводится в консоль.

Пользователь должен иметь возможность бесконечно вводить числа в консоль. Вводимые числа должны передаваться в функцию.

Вариант 11

Напишите функцию, которая принимает массив целых чисел и целое число (по умолчанию значение должно быть 0). В функции должна быть **статическая переменная** (изначально равная 1). Если переданное в функцию число не равно 0, то оно записывается в *статическую* переменную.

Если статическая переменная содержит *положительное* число, то массив нужно отсортировать по возрастанию. Если *отрицательное* – по убыванию.

Протестируйте работу функции на нескольких массивах с разными переданными числами. Выведите отсортированные массивы.

Вариант 12

Напишите функцию, которая принимает целое число. В функции должен быть **статический массив**, изначально заполненный нулями.

Переданное число помещается в *случайный нулевой элемент* массива (нужно генерировать

случайное число заново, если попался *ненулевой* элемент). После этого массив выводится в консоль. Если массив будет полностью заполнен, то новое число не добавляется.

Пользователь должен иметь возможность бесконечно вводить числа в консоль. Вводимые числа должны передаваться в функцию.