STEP computer ACADEMY

PROGRAMMING **C**

Lesson No. 8

Programming C

Contents

1. Introduction into the world of functions

Necessity of use. Declaration. Request.

As a rule, there is a situation when code segments within our program repeat several times and we have to type within the same code segment many times in a raw. Such situation has it disadvantages. First, process of the program creation becomes boring and long; second, the volume of an end file increases. What to do then?! Is there a mechanism allowing automatizing programmer's actions and contract a program code?! Yes, there is, — we will answer. This mechanism is called a **function**. So:

Function is a special construction using which a code fragment that repeats within a program two or more times is carried out of the program body. This fragment receives a name in future, and in order to use this carried out code we should indicate its name.

Declaration

There are two ways of function declaration:

- A function is declared before the function main.
- A function is declared using a prototype, and after the function main a body of a declared function is described.

First way: before function main.

General function declaration syntax:

```
return_value function_name (parameters)
{
        Block of_a repeated_code (body);
}
```

1. Function name subdues the same rules as a variable name and naturally is chosen by a programmer.

2. Parameters are input data a function needs for working with the code. Common variables are used in the capacity of parameters indicating the data type for each parameter. If a function does not need input data the parenthesis should remain empty. Second name of the parameters is arguments.

3. Return value is a result of function operation. Any of basic types is placed to the place of a return value. This is the data type that function will put to the place of its request within a program. If function does not return anything, then void(empty)is put to the place of the return value. Overall, «reasonable» return value is indicated in case if the result of function operation is needed for the further calculations.

Note: we cannot create one function inside the other one.

Note: cannot request a function before its declaration.

Function request syntax.

To use a function within a particular code section, we should request it directly within this section. Request of a function is a prescription for an operating system to start executing a code fragment that is contained within the function body. Upon termination of the function execution, a program should continue its operation within a main code from the same place where a function was requested. Function request consists of function name indication, transfer of arguments (if any) and obtaining a return value if there is a necessity to receive it):

```
variable_name=function_name(parameter1, parameter2,....,
parameterN);
```

1. Types of these values should correspond to the argument types upon function defining. There are exceptions though, in particular when a transmitted value can be easily transformed into the type in question.
2. Upon function request we should always indicate a number of parameters that was defined upon its declaration.
3. Variable type, into which a return value will be written, should coincide with the type that the function actually returns. It is not obligatory but recommended.

Key word return

To return the Value from the function into the program, specifically to the place from where the function was requested using an operator return. Return syntax is the following:

```
return value;
```

If the function does not return any values then operator return can be used simply for terminating the function; with this purpose we write:

```
return; // in this case return completes for the function just like break
for the loop.
```

Remember about important aspects of working with return:

1. There are may be several return operators (depending on a situation), but only one will complete its work.
2. If return was initiated (regardless the form), everything within the function below it will not be completed.
3. If the type returned by function is not void, then we need ALWAYS use the form: **return value**;

Now when we are familiar with a theoretical part, we can proceed to the next chapter — use cases on creating functions.

2. Use cases for creating and requesting functions

Function that does not accept any parameters and does not return any value.

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

// creating a function
void Hello(){
        // displaying a text line on a screen
        cout<<"Hello, World!!!\n\n";
}

void main(){

        Hello(); // request
        Hello(); // request
        Hello(); // request
}
```

Result — three times a phrase — Hello, World!!! — on a screen. **Function that accepts one parameter, but does not return any value.**

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

// draws a line of asterisks with the length count
void Star(int count){
        for(int i=0;i<count;i++)
                cout<<'*';
        cout<<"\n\n";
}

void main(){

        Star(3); // display of a line out of three asterisks
        Star(5); // display of a line out of three asterisks
}
```

Function that accepts two parameters, but still does not return any value.

```
#include <iostream>
using namespace std;
// draws a symbol line - symb, with the length count
 void AnyLine(char symb, int count){
        for(int i=0;i<count;i++)
                cout<<symb;
        cout<<"\n\n";
 }
 void main(){

        AnyLine('+',3); // display of a line out of three pluses
        AnyLine('=',5); // display of a line out of five equals

 }
```

Function that accepts two parameters and returns a value.

```
#include <iostream>
using namespace std;
// calculates the power (Pow) of a number (Digit)
int MyPow(int Digit, int Pow){
        int key=1;
        for(int i=0;i<Pow;i++)
                key*=Digit;
        return key;
}
void main(){
         // record of a returned result into the variable res
        int res=MyPow(5,3);
        cout<<"Res = "<<res<<"\n\n";

}
```

3. Arguments transfer. Function prototypes

Arguments transfer by value.

Let's talk about what is going on within RAM. Arguments that are indicated upon defining a function are called **formal**. It relates to the fact that they are created at the moment of function request within RAM. Upon exit from the function such parameters are destroyed. Therefore, if there are parameters with the same name within another function there won't be any conflict. Let's study one of arguments transfer methods:

Example of formal parameters operating upon data transfer by value.

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

// Should change the variable values' places
void Change(int One, int Two){
        cout<<One<<" "<<Two<<"\n\n";// 1 2
        int temp=One;
        One=Two;
        Two=temp;
        cout<<One<<" "<<Two<<"\n\n";// 2 1
}
void main(){

        int a=1,b=2;
        cout<<a<<" "<<b<<"\n\n"; // 1 2
        // transfer by value
        Change(a,b);
        cout<<a<<" "<<b<<"\n\n"; // 1 2
}
```

1. 1. it is not a and b are transferred to the function, but their exact copies.

2. All changes happen to the copies (One and Two), wherein true a and b remain unchanged.

3. Upon exit from function variable copies are deleted.

Proceeding from the above said — so far be attentive upon processing the values inside the function. Later we will learn how to resolve this problem.

Note: by the way, nothing like that happens to arrays. All changes of an array within a function are remained after exit from it.

Something about arrays...

There is some peculiarity about using the arrays in the capacity of arguments. This peculiarity is that an array name is transformed to an indicator directed onto its first element, i.e. when transferring an array an indicator is transferred as well. That is why a requested function cannot distinguish, whether a transferred indicator is directed to the array beginning or to the only object. When transferring one-dimension array it is enough to simply indicate empty square brackets:

```
int summa (int array[ ], int size){
     int res=0;
     for (int i = 0; i < size; i++)
        res += array[i];
     return res;
}
```

If a two-dimension array is transferred into the function, description of the relevant function argument should contain a number of columns; number of lines is not essential, for as we have discussed before, a pointer is directly transferred into the function.

```
int summa (int array[ ][5], int size_row, int size_col){
     int res=0;
     for (int i = 0; i < size_row; i++)
         for (int j = 0; j < size_col; j++)
                res += array[i][j];
     return res;
}
```

Function prototypes or second declaration method.

Second method of function declaration implies that we should inform a compiler that there is a function. With this purpose we should put a function name, its arguments and type of a return value before the «main». This construction is also called a function prototype. When a compiler meets a function prototype, it knows exactly that there is a function within the program after the «main» — and it should be exactly there.

```
libraries
return_value function_name(arguments);
void main(){
        тело main;
}
return_value function_name(arguments){
        function bodie;
}
```

```
#include <iostream>
using namespace std;
// прототипы
void MyFunc();
void MyFuncNext();

void main(){
        MyFunc();       //MyFunc
        MyFuncNext();   //MyFuncNext
}
//описания
void MyFunc(){
        cout<<"MyFunc\n";
}
void MyFuncNext(){
        cout<<"MyFuncNext\n";
}
```

It is considered that such function declaration is the most attractive and correct.

4. Visibility scope. Global and local variables

Visibility scope.

Any curve brackets within the program code form a so called visibility scope. It means that variables declared within these brackets are visible only inside of the brackets. In other words, if we address the variable created within the function, loop or if or any other from the other place of a program, there will an error occur, as outside of curve brackets this variable is to be destroyed.

```
int a=5;
if(a==5){
        int b=3;
}
cout<<b; // error! b does not exist
```

Global and local variables

According to visibility scope rules variables are divided into two types: local and global.

Local variables are created inside a code section, and we already know what it means for a program.

Global variables are created without any visibility scopes. Mainly before the function main(). Such variable is visible in any place within a program. Global variables in comparison with local ones are initialized by 0 by default. And what is more important, any changes to global variable that occur within the function are saved after we exit it.

Note: remember — if there is, for example, a global variable a, and inside the function the same variable with the same name is defined, then inside the function we use a variable declared within it. That is why it is better to avoid using variable names, which are invisibly used within external visibility scopes. We can reach that by completely avoiding using the same identifiers within the program.

```
int a=23; // global a
void main(){
        int a=7;// local a
        cout<<a; // 7, local is used
}
```

5. Default arguments (parameters)

Formal function parameter can be assigned with the default argument. It means that this argument may be not transferred upon request. In this case default value will be used.

General syntax for realization of such an approach looks as follows:

```
type_of return_value function_name(type_arg name_arg=default_value)
```

Here **default_value** is the value which is assigned to an argument if it was omitted upon request. Definitely, there can be several default arguments:

```
type_of return_value function_name(apr1=value, apr2=value)
```

Default arguments are the arguments starting from the right end of the list of the function parameters and further intermittently in sequence from the right to the left. For example:

```
void foot (int i, int j = 7) ;      //admissible
void foot (int i, int j = 2, int k) ;        //not admissible
void foot (int i, int j = 3, int k = 7) ;    //admissible
void foot (int i = 1, int j = 2, int k = 3); //admissible
void foot (int i=- 3, int j);                //not admissible
```

Let's review an example of handling default parameters.

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

// draws a line out of asterisks with the length count
void Star(int count=20){
        for(int i=0;i<count;i++)
                cout<<'*';
        cout<<"\n\n";
}

void main(){

        Star(); // display of a line out of 20 asterisks
        Star(5); // display of a line out of 5 asterisks

}
```

That's it for today. And now it's your turn to show the job!!! No test for today, only home assignment for you. Good luck!!!

6. Home assignment

1. Write a function that receives in the capacity of arguments an integer positive number and enumerating system into which this number is to be converted (number systems from 2 to 36). For example, when we convert the number 27 into a number system 16 we should get 1B; 13 into the 5th — 23; 35 into 18th — 1H.

2. Game «Dice». Condition: there are two dices with the values from 1 to 6. The games goes with the computer, the dices are thrown in turn. Wins the one who has a bigger amount of fallen out points after five throws. You should foresee the possibility of who gets the first move: a person or a computer. Dices are displayed by means of symbols. In the end of the game we should display an average amount according to throws for both participants.