Laboratory 3: “Functions”

Leszek Ciopiński, MSc
Arkadiusz Chrobot, PhD
Grzegorz Łukawski, PhD

September 4, 2026

1 Introduction

The second section of this laboratory manual explains how functions are created and used in programs written in C. The third section is about local variables. The last section describes function parameters and arguments. This manual ends with a set of programming exercises for solving during the laboratory classes.

2 Functions

C is a language that supports the procedural paradigm of programming. A part of this paradigm is the concept of *subroutines* that is realized in the C language as *functions*.

2.1 Prototype And Function Definition

The function definition in `C` has two parts: a *header* and a *body*.

The header starts with a data type of the value returned by the function (*the return type*). Some functions return no value. In that case their return type is `void`. The next element of the header is the *function name*. The name is followed by parentheses that enclose a *parameter list* i.e. declaration of *parameters* separated by commas. If the function accepts no arguments then its parameter list should be empty. It means that it should contain only the `void` keyword. If a programmer leaves the parameter list literally empty then the compiler will interpret it as information that the function accepts an unspecified number of arguments.

The function body (i.e. declarations of local variables, definitions of other local elements, statements to be performed) is enclosed by curly braces.

In C functions can also be declared by defining their prototype, i.e. their headers ended by a semicolon. Declared function must be defined, but its definition doesn't have to immediately follow its declaration. The definition of this function can be even located behind the definition of `main()`. It is allowed to skip the names of parameters in the function prototype, but their data types must be given. The C language doesn't allow the function definition to be nested in other function definition.

2.2 Return Statement

If the function returned type is other than `void` then the `return` keyword must be used in the function body, followed by an expression that evaluates to a value of a data type compatible with the return type of the function. The expression can be enclosed by parentheses, but it is not mandatory. It can be also as simple as a constant or literal, but usually it is more complex. Apart from returning, the `return` keyword causes the function to terminate. Thus, it is important that every possible way of terminating the function should end with this keyword. Otherwise, the function may return an unspecified value.

The `return` keyword may be used in the body of a function that returns no value (i.e. its return type is `void`), but in this case the keyword has to be immediately followed by the semicolon. The only work performed by the `return` statement in such a function is terminating it.

2.3 Function Call

If the parameter list of a given function is not empty, then *arguments* have to be assigned to the parameters in the *function call statement*. If the function has no parameters then its is called (or invoked) by using its name followed by empty parentheses and the semicolon. In this context the parentheses are the *function call operator*. The value returned by function can be assigned to a variable of a compatible data type or it can be used in an expression. It can also be ignored. In the latter case we are interested not in the value but in the side–effect of the function. To inform the compiler that the result of the function is really ignored and it is not a mistake made by the programmer, we can cast the returned value to `void`. However it is not necessary. Usually not assigning the returned value is sufficient. Calling a function that returns no value in an expression or assigning its value to a variable is an error.

The program in the listing 1 demonstrates several ways of creating and calling of functions that have no parameters. Functions with parameters are described 5 section of this manual.

```c
#include<stdio.h>

int f1(void)
{
        puts("I'm the f1() function and I return 5.");
        return(5);
}

int f2()
{
        puts("I'm the f2() function and I return 4.");
        return 4;
}

void f3(void)
{
        puts("I'm the f3() function and I return nothing.");
}

void f4()
{
        puts("I'm the f4() function and I also return nothing.");
}

void f5(void)
{
        puts("I'm the f5() function.");
        return;
        puts("I return nothing and I don't print this message.");
}

void f6(void); //Function prototype

int main(void)
{
        int a=f1(); //The  f1() function is called. The value returned by
                    //the function is assigned to the "a" variable.
        printf("The value of the \"a\" variable: %d\n",a);
        (void)f1(); //The f1() is called and its return value is ignored.
        f2();       //The f2() is called and its return value is ignored.
        //a=f3();   Calling f3() in such a way is an error!
        f3();
        f4();
        f5();
        f6();
        return 0;
}

void f6(void)
{
        puts("I'm the f6() function. I'm defined after the main().");
}
```

Listing 1: Function calls

3 Local Variables

In C local variables are also called automatic variables. They can be declared almost anywhere within the definition of a function, but they do not have any specific default value. It is the programmer responsibility to initialize them before using. Usually local variables are created in the program's memory area called a *call stack*, but thanks to the `register` keyword they can be placed in one of the CPU registers. It can increase the performance of a program, however using that keyword is not recommended because it makes applying the address operator (discussed later in the manual) impossible and modern C compilers are often better at recognizing which variables should be kept in registers than humans.

The local variables are destroyed immediately after the function terminates. If we need a local variable with a lifespan of a global one then we can use the `static` keyword in its declaration. Such a variable is not destroyed between subsequent calls of the function, moreover it keeps the last value assigned to it. Its default value is zero. Basically, the static local variable is a global variable, that can be accessed only inside the function where it is declared. The program in the listing 2 demonstrates how local variables can be declared and used.

```c
#include<stdio.h>

//It is compiled with warnings!
void f1(void)
{
        int a;
        printf("I wonder what is the vale of \"a\": %d\n",a);
        register int b;
        printf("... and now the \"b\": %d\n",b);
        static int c;
        printf("This function was called %d times.\n",c);
        c++;
        a++;
        b=5;
}

int main(void)
{
        int i;
        for(i=0;i<3;i++)
                f1();
        return 0;
}
```

Listing 2: Local variables

4 Pointers

Pointers are special variables that can be used as parameters in functions. For better understanding how they work we have to make a short digression about the internal organization of the RAM, particularly about how the memory cells are addressed.

For all variables that the program uses the computer allocates a place in the Random Access Memory. Although RAM can store data of different data types, its internal structure is quite simple. From the programmer point of view it is build from memory cells. Each of them has its own unique address and it can store one byte (8 bits) of information. Hence, the variable of the `char` type occupies only one memory cell, while the `long int` variable takes 8 consecutive cells (in PC computers). When the program is compiled the name of the variable is replaced by an address in memory where the variable should be allocated. Actually this process is a lot more complicated than that, but the details won't be

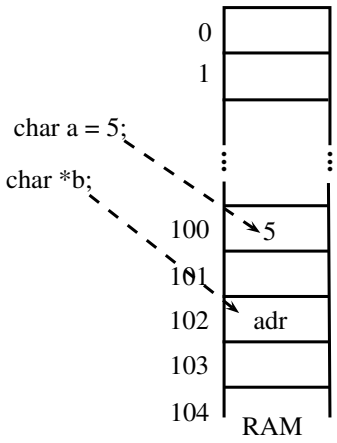


Figure 1: An exemplary mapping of variables into the RAM — the addresses for the variables are chosen arbitrary

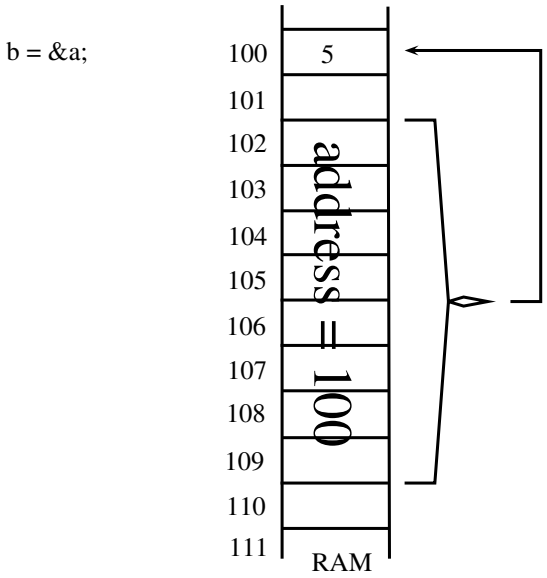


Figure 2: Getting the address of the `a` variable — the result is then stored in cells which addresses range from 102 to 109.

necessary to understand the rest of this topic.

The pointer is a variable that stores address of other variable. In case of 32–bit computers the size of the pointer is 4 bytes (which is exactly 32 bits), and in case of the 64–bit computer systems its size is 8 bytes (also 64 bits). Regardless of the data type of the variable that the pointer *points to*, its size is always the same. In this context the "points to" phrase means that the pointer stores an address of the variable. To declare a pointer the asterisk (`*`), casually called "star", symbol is used.

An example of mapping a "regular" variable and a pointer into RAM is shown in the Figure 1. The `a` variable is allocated in the RAM under the address 100. Assuming a 64–bit computer system, the cells which addresses range from 102 up to 109 are allocated for the pointer.

The default value of a pointer is unspecified if it is declared as a local variable or equal `NULL` if it is declared as a global variable[1].

We can get the address of the variable `a` by using the *address operator* which is represented by the symbol (`&`). The result of this operator can be assigned to the pointer as shown in the Figure 2. This figure also shows (figuratively) how the address of `a` is stored in the pointer `b`.

In the listing 3 the usage of pointers is demonstrated. The `b=&a;` statement assigns to the `b` pointer

[1] In newer standards of C the `NULL` can be replaced by 0.

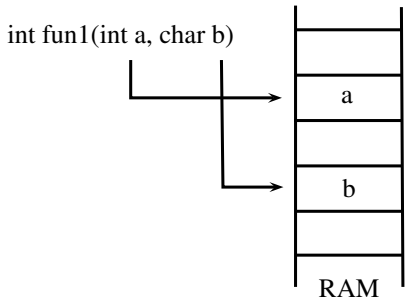


Figure 3: An example of mapping function parameters into the RAM

the address of the `a` variable. That's why the second invocation of `printf()` displays the address of this variable. It may change each time the program is run. To get the value of the variable pointed by `b` we have to use the *dereference operator* represented by the asterisk (`*`) symbol — the same that we use for declaring a pointer. The third invocation of `printf()` shows how to use it. Because the pointer **is not** the copy of the variable, the modification of the value of `a` results in displaying 7 by the last invocation of the `printf()` function.

```c
#include<stdio.h>

int main(void)
{
    char a = 5;
    char *b;

    b=&a;
    printf("%d\n", a);   //Displays 5
    printf("%ul\n", b);  //Displays the address of "a"
    printf("%d\n", *b);  //Displays 5

    a=7;
    printf("%d\n", *b);  //Displays 7

    return 0;
}
```

Listing 3: An example of using pointers.

5 Parameters

Parameters are special local variables that allow the function to exchange data with other functions. They cannot have the same names as local variables declared immediately in the body of the function, but they can have the same identifiers as global variables. The basic way of passing information to a function in C is passing by value[2]. When the function is invoked the values of arguments are copied to the parameters, as shown in the Figure 3. That's why the values of the parameters can be changed inside the function, but the values of arguments stay untouched. If we want to pass a read only data then the declaration of the parameter should start with the `const` keyword.

If we want to change inside the function the value of a variable that we use as an argument of this function then we need to declare the parameter of the function, which is passing the argument, as a pointer. We have to put the `*` symbol after the data type and before the identifier of the variable. This

[2]In some textbooks it is also referred as "calling by value".

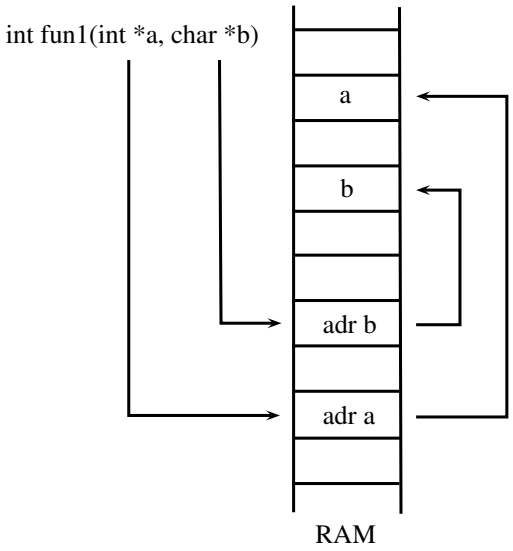


Figure 4: Modification of the `a` and `b` will persist after the function terminates

time the address of the argument will be stored in the parameter, which means that using it the function may change the value of the argument and the modification will be persistent — the variable will still store the changed value even when the function terminates. The Figure 4 shows how this passing of *argument by an address* works.

Inside the function body we can access the value pointed by the pointer by using the dereference operator (`*`) before the pointer name. In the function call statement the parameter has to be substituted by an expression that gets an address of a variable by using the address operator (`&`). Several examples of using arguments and parameters in functions are given in the listing 4.

We should be careful when using such operators as post–increment and post–decrement in argument of a function, because the outcomes may be unexpected. The listing 5 contains a short program that passes five expressions of the form `i++` to the `printf()` function. Some compilers, like the GCC adds values of arguments in reversed order to the call stack, which means that they will be displayed also in reversed order. Others like `clang` add the values of arguments to the call stack in the same order as the arguments occur in the code and they are not reversed when displayed on the screen.

```c
#include<stdio.h>
//Compiles with warnings!

int main(void)
{
        int i=1;
        //Displays: 5 4 3 2 1, instead of expected: 1 2 3 4 5.
        printf("%d %d %d %d %d\n",i++,i++,i++,i++,i++);
        return 0;
}
```

Listing 5: A program which result depends of the compiled used for translating its code

```c
#include<stdio.h>

int f1(int a, int b)
{
        return a+b;
}

void f2(int a)
{
        printf("The f2() function got the vale: %d",a);
        a=10;
        printf(" and changed it into: %d.\n",a);
}

void f3(int *a)
{
        printf("'The f3() function got the value: %d",*a);
        *a=10;
        printf(" and changed it into: %d.\n",*a);
}

void f4(const int a)
{
        printf("The f4() function got value %d that cannot change.\n",a);
}

int main(void) {
        //A function that returns value may be called inside an expression.
        printf("The result of f1() function: %d\n",f1(5,3));
        int a=3;
        printf("The value of \"a\" before calling f2(): %d\n",a);
        f2(a);
        printf("The value of \"a\" after calling f2(): %d\n",a);
        printf("The value of \"a\" before calling f3(): %d\n",a);
        f3(&a);
        printf("The value of \"a\" after calling f3(): %d\n",a);
        printf("The value of \"a\" before calling f4(): %d\n",a);
        f4(a);
        return 0;
}
```

Listing 4: Parameters

6 Exercises

All Your programs have to have functions with parameters!

1. Write a program that allows the user to enter three integers and finds and displays the greatest of them.
2. Write a function that gets a number of a day in a week by a parameter and prints its name. Use the function in a program.
3. Write a program that finds the sum of arithmetic or geometric progression, depending on the user choice.
4. Write a program that finds the surface area of a cuboid. The length, width and height of the cuboid should be provided by the user.
5. Write a program that finds the surface area of a regular pyramid. The base of this pyramid is a square. The high of the pyramid and the length of an edge of the base should be provided by the user.
6. Write a program that calculates an aproximated value of π number using the Leibniz Series. The accuracy of the approximation should be given by the user.
7. Write a program that calclates an approximaed value of the square root of a natural number using the Newton's Method. The number and the accuracy should be given by the user.
8. Write a program with a function that calculates the remainder after division of one nonzero natural number by another. In the definition of the function You cannot use the remainder operator, only substarction and relational operators. The numbers should be provided by the user.
9. Write a program that gets from the user a sequence of max. 10 characters build with the use of only two letters: "a" and "b" and entered character by character. The program should terminate if it detects a subsequence "abba" in the entered sequence or when user enters the 10^{th} letter. In the former case the program should display a message informing the user that it detected the subsequence.