Introduction to Programming Language (C)

(EEF 110E)

Dr. Emre DİNCEL

LECTURE NOTES

(WEEK 4)

❖ Functions in C

A function is a set of statements which take inputs, perform some specific computation and then produces output. Instead of repeating the same code again and again we can use functions. Functions allow,

- → Generic/parametric solutions to problems
- → Division of big problems into smaller ones
- → Reuse of program segments in different problems

We frequently use functions in mathematics. Examples include trigonometric functions $\left(\sin(x), \cos(x)\right)$, exponential functions $(\mathrm{e}^x, 10^x)$, rational functions such as,

$$f(x,y) = \frac{x^2 + y^2 + 2xy}{y - 2}$$

However, a function in C is *not* restricted with math functions and any piece of code that can be called again and again can be defined as a function. Every function has a set of <u>arguments</u> (as input parameters) such as x and y in the above examples and an <u>output</u> (that is the result of the function).

In order for us not to reinvent the wheel, there are many functions ready for use in C. Some of these functions, such as the **sizeof** function, come with the core of the language. However, a great deal of functions is provided by the standard libraries.

For example, we have already seen that **printf**, **getc**, **getchar**, **putc**, **putchar** and **scanf** functions are defined in *stdio* library. So, in order to be able to use these functions we have to include the header file stdio.h that holds the declarations in our C program. There are other standard libraries such as *stdlib* and *math,* which we will be talking about.

- Math Functions (math.h):

Some of the mathematical functions and constants are defined in math library. For instance,

- Trigonometric functions $(\sin(x), \cos(x), \tan(x), \mathrm{asin}(x), \mathrm{acos}(x), \mathrm{atan}(x))$
- Power and square root $\left(\mathrm{pow}(x,y), \mathrm{sqrt}(x)\right)$
- Exponential, natural log, log in base 10 $\left(\exp(x), \log(x), \mathrm{log10}(x)\right)$
- Rounding function $\left(ceil(x), round(x), floor(x)\right)$

In these functions, x can be a real (double) valued expression and the output of the function is a real (double) number. So these are used anywhere in the program where a real expression can be used. For example,

```
num = 2*sin(0.5)/log(x+4.0);
```

means

$$\text{num} = \frac{2\sin(0.5)}{\ln(x+4)}$$

Consider the following program written in C:

```
#include <stdio.h>
#include <math.h>

int main()
{
   double x;

   for (x=0.0; x<=90.0; x+=15.0)
      printf("sin(%lf) = %lf", x, sin(x * M_PI / 180.0));

   return 0;
}
```

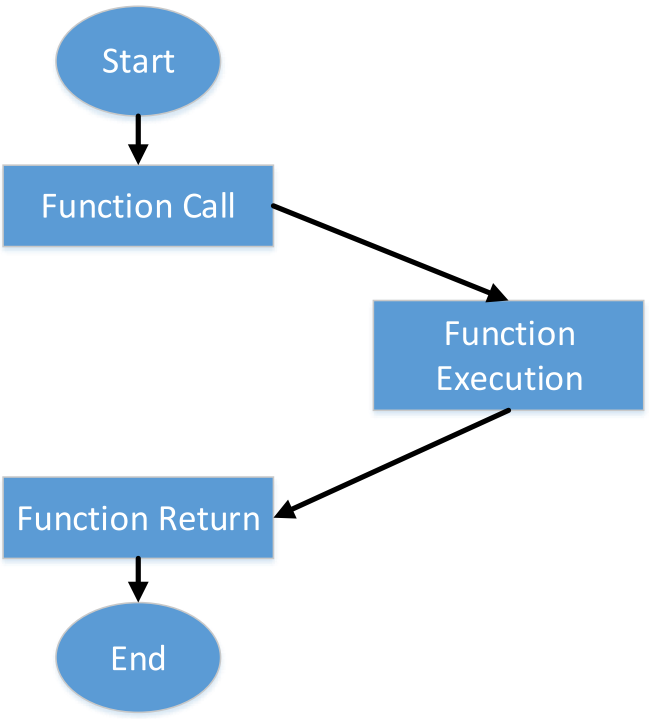
In the above code, we calculate the value of **sin(x)** function between 0 and 90 degrees with 15 degrees intervals. Since such trigonometric functions use radians, we perform a conversion between degrees and radians. Note that **M_PI** is a constant defined in *math.h* library. Also we use the format specifier "%lf" for double types. It is also possible to use it as "%.3lf" to show only 3 digits after "." to improve the readability on the screen.

- Calling Functions:

When running if a program comes to a function the execution continues from the function. This is named as *calling the function*. Before this, however, all argument expressions of a function are evaluated and the results of these evaluations are passed to the function. The statements in the function are then executed and the result of the function is returned to the main program. The main program then acts as if the function statement is replaced by the result of the function. For example,

```
x = 30.0
numResult = sin(x * M_PI / 180.0)
```

Here, first of all, the expression “x * M_PI / 180.0” is evaluated (that results ≅ 0.523) and then the function **sin(x)** is called. The function call statement is finally replaced by the result of the function, hence, the value of the variable “numResult” becomes 0.5 at the end.



- Defining Your Own Functions:

Functions defined in standard libraries are generally not enough for writing large programs. In order to solve larger programming problems, you often need to define your own functions and call them from your program. The syntax for defining a function is as follows:

```
return_type functionName(type_1 arg1, type_2 arg2, ...)
{
   body of the function
}
```

Here, “arg1”, “arg2”, ... are called as the arguments (inputs) of the function. We shall see that arguments can also be used to provide additional outputs to the function.

The *return_type* is the data type of the value that the function returns. Some functions perform the desired operation but does not return any value. In such cases, the *return_type* of the function is set as **void**. Check the following function:

```
void printWarning(void)
{
   printf("This is a warning!\n");
}
```

As it is seen, **void** can also be used to state that the function has no arguments.

Another important thing about the functions is that functions must be declared before they are used. A function declaration is the *interface* part (the part without the main body) of the function. However, definition of a function (implementation details) can be given later.

Please examine the following C program:

```c
#include <stdio.h>

// Declaration of the function
double mySum(double, double);

int main()
{
   double x = 2.5, y = 3.1;
   double result = mySum(x, y);
   printf("%lf + %lf = %lf", x, y, result);
   return 0;
}

// Definition of the function
double mySum(double a, double b)
{
   double result;
   result = a + b;
   return result;
}
```

In the above code, we wrote a function named "mySum" which takes two arguments (inputs with type of *double*) and returns a *double* type output. Inside the function, we performed a summation of the arguments and returned the resulting value.

The main() function consists of only one statement (printf) so this is executed. However, for printf to be executed first mySum() function is called. When the function is called, a space for two real (double) arguments are reserved in memory with labels "a" and "b", and then initialized to the actual argument values which are 2.5 and 3.1, respectively. After that a space for a double variable is reserved in memory with label "result" and the sum operation is executed. Function then returns the value (5.6 in our case) and the return value of the function is assigned to "result" defined in main function. Finally, we see the result of the function on the screen via printf() function.

- Local, Global and Static Variables:

All variables defined in functions are called local variables. Normally, memory corresponding to these variables is reserved dynamically when the function is called, and is marked as 'empty' when the function terminates. It means that when function terminates all local variables are removed from memory. Also, other functions cannot reach these variables by just using their names.

Note that, in the above example, the variable "result" in **main()** function is different from the variable "result" defined in **mySum()** function.

It is possible to define variables outside of functions. Such variables are called as global variables. The memory corresponding to these variables are reserved when the program starts and marked as empty when the program terminates. All functions defined after the definition of these variables can reach and change them. Check the below example:

```
#include <stdio.h>

double myFnc(double);

double multiply = 5.0; // A global variable

int main()
{
   double x = myFnc(2.0);

   printf("%lf * %lf = %lf", 2.0, multiply, x);

   return 0;
}

double myFnc(double a)
{
   return a * multiply;
}
```

Here, the variable "multiply" is defined globally; therefore, it is possible to reach this variable from any part of the program. Nevertheless, it is not recommended to use global variables as such variables can cause serious problems if their content is changed somewhere else by mistake.

As stated earlier, normally local variables are erased when the function returns to the calling function. If a local variable is defined with the static specifier, however, its value is saved. Examine the below program example:

```
#include <stdio.h>

void count(void);

int main()
{
   count();   // Values of the k and j are: 1 1
   printf("\n");
   count();   // Values of the k and j are: 2 1

   return 0;
}

void count(void)
{
   static int k = 1;
   int j = 1;
   printf("Values of the k and j are: %d %d", k, j);
   k++; j++;
}
```

- Recursive Functions:

If a function calls itself either directly or through a chain of function calls the function is called a recursive function.

It is programmer's responsibility to make sure that the recursion (calling itself) does not continue infinitely and there exists a segment of the function where it is possible to exit from the function without recursion.

When planned carefully recursive functions are easy to implement. However, they might slow down the program and can use excessive amount of memory unnecessarily. Therefore, should be avoided when possible. In the below recursive function, factorial of a number (n!) is calculated recursively:

```
long int factorial(int n)
{
   if (n < 2)
      return 1; // 1! = 1
   else
      return n * factorial(n - 1); // Since n! = n*(n-1)!
}
```