# Engineering program development 7

Edited by Péter Vass

# Functions

**Function**
is a separate computational unit which has its own name (identifier).
The objective of a function is solving a well-defined problem.
It often occurs that the solution of a complex problem is implemented by using several functions solving different parts of the whole problem.
The application of functions in our programs significantly facilitates the work of programming.
A function may have input paramaters (or variables) by which data may be got from its environment.
It also may have an output parameter (or variable) by which a value (return value) may be sent to its environment.

# Functions

So, the input and output parameters provide a data connection between the code of a function and the code of its environment.

The detailed description of a function is called function definition. A function definition includes the function header and the body of the function. In fact, the body of the function contains the source code of the function.

The application of a function in a program is called function call.

A function call is always embedded in the definition of another function. Actually, a function calls another function.

For example, the body of the main function always contains one or more function calls.

# Function

There are two main categories of the functions:

- library functions,
- user-defined functions.

Library functions previously compiled are stored in function library files.

We must not define these functions in our source codes.

The standard library functions form a special type of functions which are available for all of us (e.g. printf, scanf, …).

Of course, the header file(s) including the function declarations must be built in the source code by using the pre-processing directive #include.

Contrary to library functions, user-defined functions must always be defined in the source code of a program.

In the case of a single-module C program, the definitions of the user-defined functions are suggested placing after the definition of the main function.

# Functions

A function definition unambiguously specifies how the function works and the form of its call.

A function definition consists of two main parts:

- a function header,
- a function body.

A function header comprises the following components:

- the type of output parameter (on the left side),
- the name of the function (function identifier),
- the list of input parameters enclosed by a pair of round brackets.

If the type of output parameter is int, its indication is not compulsory because the default type of output parameter for functions in C is int.

If the function has not any return value, the type of output parameter must be void.

# Functions

Naming convention concerning the functions are the same as that of variables.
The list of formal input parameters with the type specifications follows the function identifier and it is placed between round brackets.
If a function has not any input parameter, an empty pair of round brackets must be applied.

Examples for function headers:

```
double funct_1(double x, double y, int n)
void funct_2(float a)
int funct_3()
funct_4()
```

# Functions

The function body is located after the function header and it is enclosed by a pair of braces.
A function body usually contains:
the definitions of the local variables (not always necessary),
instructions of the function (statements),
one or more return statements which indicate the exit point or points of the function.
Example:

```
int positive(double x){
    if (x > 0)
        return 1;
    else
        return 0;
}
```

If a function has not any return value, nothing follows the return statement in the function body.

# Functions

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

main(){
    float x;

    printf("Type a number: ");
    if (scanf("%f",&x) != EOF){
        printf("The value of the given number:
%f",x);
        return EXIT_SUCCESS;
    }
    else{
        printf("Input failure");
        return EXIT_FAILURE;
     }
}
```

# Functions

EXIT_SUCCESS and EXIT_FAILURE are symbolic constants defined in the header file stdlib.h .Their values are 0 and 1, respectively. They can be used after a return statement for indicating the successful or unsuccessful execution of a function.

**Function declaration**

It often occurs that a user-defined function is called in the body of the main function.

If the function definition is placed after the definition of main function, a function declaration must be applied before the main function.

A function declaration is a simplified form of the function header which is also called function prototype.

A function declaration contains:

- the type of output parameter,
- the function identifier
- and the types of input parameters arranged in a list enclosed by round brackets.

# Functions

Example:

```
double func_1(double, double, int);
```

The difference between a function header and a function declaration is that the identifiers of the formal parameters are not needed to represent in a function declaration.
Function declarations are important for the compiler to identify the user-defined functions called in the main function.
If the function definition of a user-defined function precedes the main function in the source code, we must not declare the function because the compiler will be able to identify the function by means of its definition.

# Functions

```
#include <stdio.h>
#include <stdlib.h>
int positive(double);    /*function declaration*/
int negative(double);    /*function declaration*/
int main(){
     double x;
     printf("Type a number: ");
     if (scanf("%lf",&x) != EOF){
          if (positive(x))    /*the call of function positive*/
               printf("The number is positive.");
          else if (negative(x))    /*the call of function negative*/
               printf("The number is negative.");
          else
               printf("The number is equal to zero.");
          return EXIT_SUCCESS;
     }
     else{
          printf("Input failure.");
          return EXIT_FAILURE;
     }
}
```

# Functions

```
/*the definition of function positive*/
int positive(double a){
    if (a > 0)
        return 1;
    else
        return 0;
}

/*the definition of function negative*/
int negative(double a){
    if (a < 0)
        return 1;
    else
        return 0;
}
```

# Functions

When we call a function in the body of another function (e,g. in the main function), we have to exchange the list of formal parameters for the list of actual parameters.

A list of actual parameters contains the identifier of the input variables defined in the calling function.

The types of actual parameters must correspond to the types of formal parameters in order of the input parameters.

A list of actual parameters may not contain type specifications.

Before executing the code of a function, the values of the variables used in the list of actual parameters are copied in the memory space reserved for the function.

That process is called passing input parameters by values.

In such a way, the values of these variables can be used in the statements of the function body.

# Functions

When the execution of the code of function arrives at the return statement, the output value of the function will be the value of the expression following the return statement.

The output value of a function is also called return value and it can be assigned to a variable of the calling function or used in an expression.

The consequence of passing input parameters by their values is that the values of the variables used in the list of actual parameters can not be changed by the code of the function.

The values of the input variables can be used in the computations of the function but the memory spaces of the input variables can not be reached by the function so the stored values can not be modified.

The following example tries to demonstrate that only the copied values of the input parameters can be modified by the function but the actual values of the variables remain the same.

# Functions

```
#include <stdio.h>

void change1(int, int);       /*function declaration*/

main(){
        int a=5, b=10;

        printf("a= %d\tb= %d\n",a,b);
        change1(a,b);           /*function call*/
        printf("a= %d\tb= %d\n",a,b);
}

/*function definition*/
void change1(int x, int y){
        int s;

        s=x;
        x=y;
        y=s;
        printf("x= %d\ty= %d\n",x,y);
}
```

# Functions

In order to actually modify the values of the variables in a function, we must pass the memory addresses of the variables to the function by means of pointers.
The following example shows how we can actually modify the values of variables in a function by using pointers as input parameters of a function.

# Functions

```
#include <stdio.h>

void change2(int *, int *);

main(){
        int a=5, b=10;

        printf("a= %d\tb= %d\n",a,b);
        change2(&a,&b);
        printf("a= %d\tb= %d\n",a,b);
}

void change2(int *x, int *y){
        int s;

        s=*x;
        *x=*y;
        *y=s;
        printf("x= %d\ty= %d\n",*x,*y);
}
```

# Functions

**One-dimensional arrays as input parameters of a function**

An array or rather the elements of an array can not be passed to a function by their values.
But the initial address of the memory space reserved for an array can be passed by using a pointer as an input parameter.
In such a way, the elements of an array can be reached and modified in a function.
Since the initial address of an array does not provide any information about the size of the array, we must apply an additional input parameter for specifying the size.
The following two examples show two equivalent solutions for using one-dimensional arrays as input parameters of a function.
The function called *readvector* asks the user for the coordinates of a vector.
The function called *scalarproduct* computes the scalar product of two vectors of the same size.

# Solution 1

```
#include <stdio.h>

void readvector(float *, int);
float scalarproduct(float *, float *, int);

main(){
   const int n=5;
   float a[n], b[n];

   printf("Type the coordinates of the first vector.\n");
   readvector(a,n);
   printf("Type the coordinates of the second vector.\n ");
   readvector(b,n);
   printf("The scalar product: %f", scalarproduct(a,b,n));
}
```

# Solution 1

```
void readvector(float *v, int size){
    int i;
    for (i=0; i<size; i++){
        printf("the value of the %d. coordinate: ", i+1);
        scanf("%f", v+i);
    }
}

float scalarproduct(float *x, float *y, int size){
    int i;
    float s=0;

    for (i=0; i<size; i++)
        s=s+*(x+i)**(y+i);
    return s;
}
```

# Solution 2

```
#include <stdio.h>

void readvector(float [], int);
float scalarproduct(float [], float [], int);

main(){
    const int n=5;
    float a[n], b[n];

    printf("Type the coordinates of the first vector.\n");
    readvector(a,n);
    printf("Type the coordinates of the second vector.\n ");
    readvector(b,n);
    printf("The scalar product: %f", scalarproduct(a,b,n));
}
```

# Solution 2

```
void readvector(float v[], int size){
     int i;

     for (i=0; i<size; i++){
          printf("the value of the %d. coordinate: ", i+1);
          scanf("%f", &v[i]);
     }
}

float scalarproduct(float x[], float y[], int size){
     int i;
     float s=0;

     for (i=0; i<size; i++)
        s=s+x[i]*y[i];
     return s;
}
```

# File input/output in C

In fact, a file is a series of bytes. The actual information content of a file is encoded by the values of the bytes.

By the type of data stored in a file, two primary groups of the files can be distinguished:

- text files,
- binary files.

In the case of a text file, the elementary piece of information is a character used in writing.

The code of a character (which is a positive integer) is stored by one or more bytes.

There are different code tables by which characters can be corresponded to their numerical values.

One of the most frequently used code table is provided by the character encoding standard called ASCII (*American Standard Code for Information Interchange*).

# File input/output in C

The characters are organized into lines.
The end of a line is indicated by a control character.
The end of a text file (the position after the last character) is also indicated by a control character (EOF).
The content of a text file can be displayed and edited by means of a simple text editor program (e.g. Notepad).

The order of the operations required for the text file input/output in C programs are the following:

- defining a file pointer,
- opening the specified file and assigning the file pointer to the file,
- reading and/or writing characters from and/or into the file,
- closing the file.

# File input/output in C

**Defining a file pointer**

A file pointer points to a data structure in the memory which will contain information about the file after it has been opened.
The definition of this data structure can be found in the header file stdio.h therefore we must include this file in our source code.
We do not need to know the details of this data structure but we must define a file pointer in the main function of our source code.
The definition of a file pointer consists of the type name FILE and the identifier of the pointer with an asterisk.

Example:

```
FILE *fp;
```

If more than one file is used in the program, a file pointer with an individual identifier must be defined for each file.

Example:

```
FILE *fp1, *fp2;
```

# File input/output in C

**Opening the file**

After defining a file pointer, we must open the selected file.
By means of opening a file we specify the name and path of the file, the mode of file access as well as establish a connection between the file pointer and the file.

The standard library function served for opening files is called fopen.
The header of fopen:
FILE * fopen(const char *filename, const char *mod);

The first input parameter of fopen is a character string containing the path and name of the file. If the file and our C program file are located in the same directory, it is enough to provide the name of the file to be opened.
The second input parameter is also a character string and used for specifying the mode of file access.
Example:
"C:\\Users\\Public\\Documents\\datafile.txt"

# File input/output in C

The actual value of the second parameter in a call of fopen indicates how we intend to use the file.

The possible values of the parameter mode:

| | |
|---|---|
| "r" | opening an existing file for reading data, |
| "w" | opening a new file for writing data, if the file already exists, its original content will be lost (overwriting), |
| "a" | opening an existing file for adjoining data, if the file does not exist yet, it will be created by the program, |
| "r+" | opening an existing file for both reading and writing data, |
| "w+" | opening a new file for both reading and writing data, if the file already exists, its original content will be lost (overwriting), |
| "a+" | opening an existing file for reading and adjoining data if the file does not exist yet, it will be created by the program. |

# File input/output in C

We can add a further character to the end of character string which indicates the type of file to be opened. This character has two possible values:

- t means text file,
- b means binary file.

Example for the character string of mode:

"r+t"

If we do not specify the type of file in the character string of mode, the compiler treats the file as a text file.

The function fopen normally returns a file pointer which must be assigned to the previously defined file pointer. Example:

```
fp = fopen("datafile.txt", "wt");
```

If some kind of error occurs, fopen will return NULL.

The most frequent problems resulting in errors are

- trying to read a file that does not exist,
- trying to read a file when we don't have permission.

# File input/output in C

**Writing and reading operations**

After a text file has been opened successfully, we can apply several standard library functions to implement a data stream between the program and the file.

Each of these functions has at least one input parameter by which the opened file must be identified with the proper file pointer.

Some of the functions which may be used for file input/output operations are the following:

Reading a single charcter from a text file

```
int fgetc(FILE *filepointer);
```

The function called getc returns the next character read from a file identified by its file pointer in the input parameter of the function.

It returns EOF for end of file or error.

# File input/output in C

Writing a single charcter to a text file:

```
int fputc(int c, FILE *filepointer);
```

The function called putc writes the character c to the opened file identified by the file pointer and returns the character successfully written, or EOF if an error occurs.

Reading a character string from a tect file

```
char *fgets(char *s, int n, FILE *filepointer);
```

The function called fgets reads the next character string with the maximum length of n from the file identified by the filepointer.
The pointer s will point to the memory adress of the first character in the string.
It returns the memory address stored in s, or NULL if an error or end of file occurs.

# File input/output in C

Writing a character string to a text file:

```
int fputs(const char *s, FILE *filepointer);
```

The function called fputs writes the character string s to the file identified by the file pointer.
It returns a non-negative integer, or EOF if an error occurs.

Formatted input of a text file

```
int fscanf(FILE *filepointer, const char *format
…);
```

The function called fscanf is almost identical to scanf, except that the first argument is a file pointer that specifies the file to be read. The format string is the second argument.
It returns the number of data successfully read, or EOF if an error or end of file occurs.

# File input/output in C

Formatted output of a text file

```
int fprintf(FILE *filepointer, const char *format
…);
```

The function called fprintf is almost identical to printf, except that the first argument is a file pointer that specifies the file to be written. The format string is the second argument.
It returns the number of data successfully written, or a negative integer if an error occurs.

**Closing a file**

After the required input/output operations have been impkemented, the opened file must be closed by means of the function called fclose.

```
fclose(FILE *filepointer);
```

It breaks the connection between the file pointer and the file which was established by the function fopen.

# Writing data to a text file

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

main(){
        FILE *fp;
        int i;
        float t;

        fp = fopen("datafile.txt", "wt");
        if (fp==NULL){
                printf("The file can't be opened.");
                exit(-1);
        }
        for (i=0; i<10; i++){
                t=0.2*i;
                fprintf(fp,"%f\t%f\n", t, sin(t));
        }
        printf("Successful output of the text file.");
        fclose(fp);
}
```

# Reading data from a text file

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

main(){
        FILE *fp;
        float t, sint;

        fp = fopen("adatok.txt", "rt");
        if (fp==NULL){
                printf("The file can't be opened.");
                exit(-1);
        }
        while(!feof(fp)){
                fscanf(fp,"%f\t%f\n",&t,&sint);
                printf("%f\t%f\n",t,sint);
        }
        printf("Successful input of the text file.");
        fclose(fp);
}
```