First Year Engineer in computer science

# CHAPTER 5:

# functions

2023-2024

Sid Ahmed BERRABAH

# Functions in C



**Definition** : A function is a block of statements that performs a specific task and can be executed repeatedly whenever we need it.

# Functions in C



**Definition** : A function is a block of statements that performs a specific task and can be executed repeatedly whenever we need it.

**Benefits of using function:**

# Functions in C



**Definition** : A function is a block of statements that performs a specific task and can be executed repeatedly whenever we need it.

**Benefits of using function:**

- The function provides modularity.

# Functions in C



**Definition** : A function is a block of statements that performs a specific task and can be executed repeatedly whenever we need it.

**Benefits of using function:**

- The function provides modularity.
- The function provides reusable code.

# Functions in C



**Definition** : A function is a block of statements that performs a specific task and can be executed repeatedly whenever we need it.

**Benefits of using function:**

- The function provides modularity.
- The function provides reusable code.
- In large programs, debugging and editing tasks is easy with the use of functions.

# Functions in C



**Definition** : A function is a block of statements that performs a specific task and can be executed repeatedly whenever we need it.

**Benefits of using function:**

- The function provides modularity.
- The function provides reusable code.
- In large programs, debugging and editing tasks is easy with the use of functions.
- The program can be modularized into smaller parts.

# Functions in C



**Definition** : A function is a block of statements that performs a specific task and can be executed repeatedly whenever we need it.

**Benefits of using function:**

- The function provides modularity.
- The function provides reusable code.
- In large programs, debugging and editing tasks is easy with the use of functions.
- The program can be modularized into smaller parts.
- Separate function independently can be developed according to the needs.

# Functions in C



**There are two types of functions in C**

- Built-in(Library) Functions
  - These functions are provided by the compiler and stored in the library. Therefore it is also called Library Functions.

    **ex.** scanf(), printf(), ...
  - To use these functions, you just need to include the appropriate C header files

    **ex**. stdio.h, stdlib.h, math.h,...

# Functions in C



**There are two types of functions in C**

- Built-in(Library) Functions
  - These functions are provided by the compiler and stored in the library. Therefore it is also called Library Functions.

    **ex.** scanf(), printf(), ...
  - To use these functions, you just need to include the appropriate C header files

    **ex**. stdio.h, stdlib.h, math.h,...

- User Defined Functions: These functions are defined by the programmer (user) to do specific tasks.

# Defining a Function



The general form of a function definition in C language is as follows:

```
return_type function_name( parameter list )
{
    //body of the function
}
```

# Defining a Function



The general form of a function definition in C language is as follows:

```
return_type function_name( parameter list )
{
    //body of the function
}
```

- **return_type**: A function may return a value. The return_type is the data type of the value that the function is expected to return.

# Defining a Function



The general form of a function definition in C language is as follows:

```
return_type function_name( parameter list )
{
    //body of the function
}
```

- **return_type**: A function may return a value. The return_type is the data type of the value that the function is expected to return.
- **function_Name**: The function name (function identifier) should respect the same rules for variable identifier. In C, no two functions can have the same name

# Defining a Function



The general form of a function definition in C language is as follows:

```
return_type function_name( parameter list )
{
    //body of the function
}
```

- **return_type**: A function may return a value. The return_type is the data type of the value that the function is expected to return.
- **function_Name**: The function name (function identifier) should respect the same rules for variable identifier. In C, no two functions can have the same name
- **Parameter list**: Parameters are the data to pass to the function when invoked. The parameter list refers to the type, order, and number of parameters of the function. Parameters are optional; that is, a function may contain no parameters.

# Defining a Function



The general form of a function definition in C language is as follows:

```
return_type function_name( parameter list )
{
    //body of the function
}
```

- **return_type**: A function may return a value. The return_type is the data type of the value that the function is expected to return.
- **function_Name**: The function name (function identifier) should respect the same rules for variable identifier. In C, no two functions can have the same name
- **Parameter list**: Parameters are the data to pass to the function when invoked. The parameter list refers to the type, order, and number of parameters of the function. Parameters are optional; that is, a function may contain no parameters.
- *Function body:* is the block of statements enclosed between { and } that performs the specific task of the function.

# Defining a Function



- Some functions perform the desired operations without returning a value. In this case, the return_type of the function is the keyword **void**.

```
void function_name ( parameter list)
{
        //function body
}
```

This type of function is also called **procedure**.

# Defining a Function



- Function body : The function body consists of the following three elements:

1. declaration part: variables used in function body.
2. executable part: set of statements or instructions to do specific activity.
3. return : It is a keyword, it is used to return control back to calling function.

If a function is not returning value then the return can be omitted or just write statement: **return;**

If a function is returning value then statement is:

**return value**;

# Defining a Function



- **Parameter list:**
  - The parameter list should be enclosed between parenthesis () and has the form:

    (type1 param1, type2 param2, ... )

  - If a function does not accept parameters the list should be: **(void)** or **()**

# Defining a Function



## Formal Parameters and Actual Parameters

- **Formal Parameters:**
  - The variables defined in the function header in function definition are called formal parameters.
- **Actual Parameters:**
  - The variables that are used when a function is invoked (function call) are called actual parameters.
  - The actual parameters and formal parameters must match in number and type of data.

# Defining a Function



- **Example 1:** function that does not accept any parameters and does not return any result.

```
void display ( void )
{
      printf(" Hello \n");
      printf(" -------- \n");
}
```

The function **display** does not accept any parameters and does not return any result.
Each time the function **display** is called, it prints the following two lines:

```
Hello
--------
```

# Defining a Function



- **Example 2:** function that accepts parameters and returns a result

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

The function **SquareSum** accepts two integer parameters a and b, and return an integer value which is the sum a*a + b*b

# Defining a Function



- The return statement may appear several times in the function body.

**Example :** function to compute the absolute value of a real number:

```
float abs(float x)
{
        if x > 0 return x ;
        else return -x ;
 }
```

After the execution of the return statement, the function is terminated.

# Variables declaration in a function



- The variables declared in the function body and the formal parameters are created (memory allocated) upon entry into the function and destroyed upon exit.

**Example:** variable declaration in function body

```
int SquareSum ( int a , int b )
{
    int result ;

    result = a*a + b*b ;
    return result ;
 }
```

# main function



- Every C program has at least one function, which is **main()**.
- The execution of a C program starts by the **main()** function.
- The **main()** function is of type **int** and finish with the statement **return 0 ;**

```
int main()
{
    // the body of main()
    return 0 ;
}
```

# Function Declaration



- A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
- A function declaration is called also function prototype
- A function declaration has the following parts:

```
return_type function_name( parameter list );
```

**Example** the prototype for the above SquareSum function is :

**int SquareSum(int a, int b);**

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

**int SquareSum(int , int );**

# Function Call



- While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

# Function Call



- While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

- When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function ending closing brace is reached, it returns program control back to the main program.

# Function Call



- While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

- When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function ending closing brace is reached, it returns program control back to the main program.

- To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

# Function Call



**Example :** the call of the function SquareSum in the main function

```c
#include <stdio.h>

float SquareSum( float a, float b)
{
    return a*a + b*b;
}
int main()
{
   float x, y, z;
   printf("give two values x and y : ");
   scanf("%f %f", &x, &y);

   z = SquareSum(x,y);
   printf("the squared sum of %f and %f is %f", x, y, z );
   return 0;
}
```

# Function Call



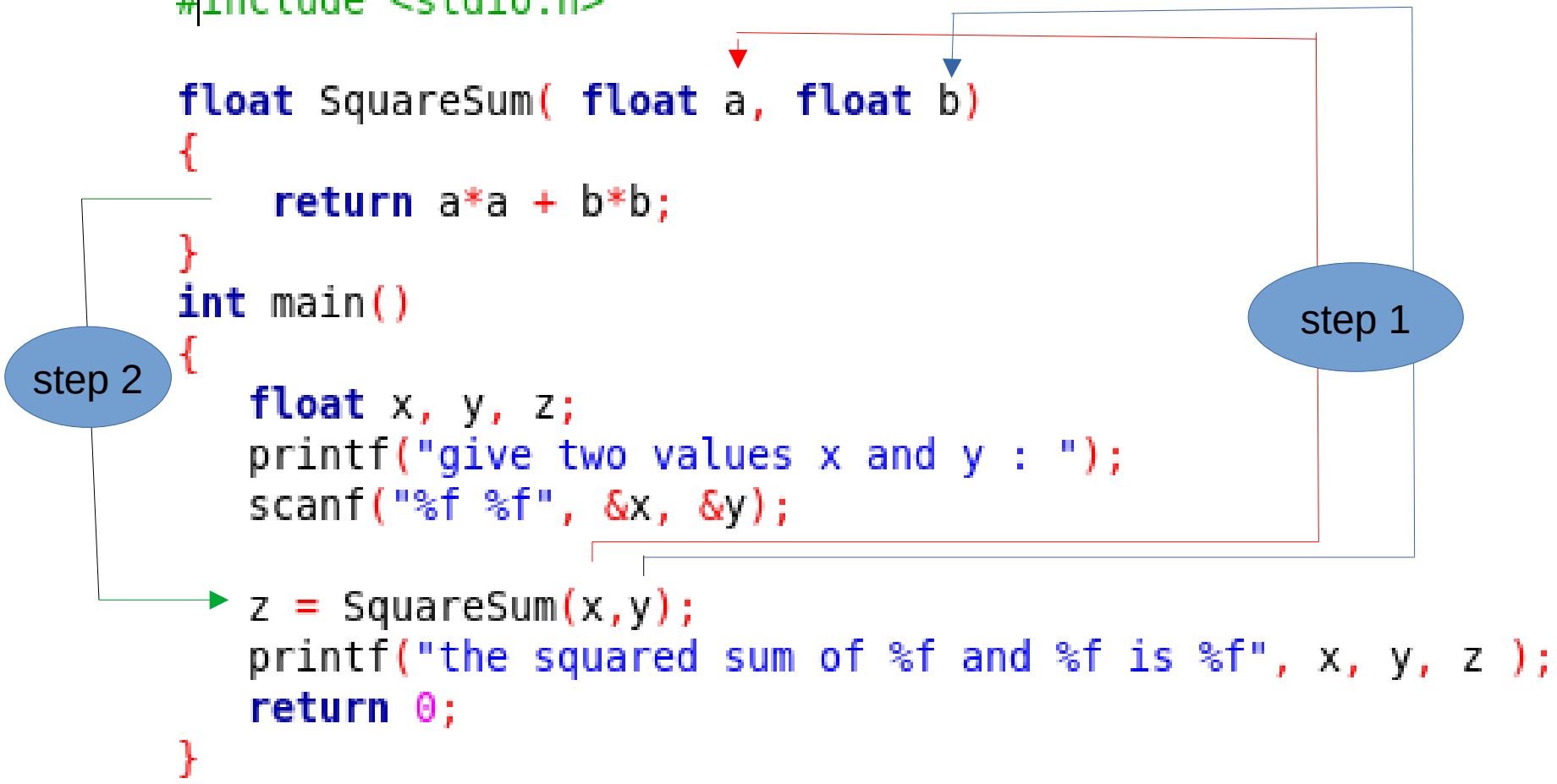
**Example :** the call of the function SquareSum in the main function

```c
#include <stdio.h>

float SquareSum( float a, float b)
{
    return a*a + b*b;
}
int main()
{
   float x, y, z;
   printf("give two values x and y : ");
   scanf("%f %f", &x, &y);

   z = SquareSum(x,y);
   printf("the squared sum of %f and %f is %f", x, y, z );
   return 0;
}
```

# Function Call



**Example :** the call of the function SquareSum in the main function



# Call by value



- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.

- In call by value method, we can not modify the value of the actual parameter by the formal parameter.

- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.

# Call by value



## Example for call by value:

```c
#include <stdio.h>

void modify(int num)
{
   printf("Before adding value inside function num = %d \n",num);
   num = num + 100;
   printf("After adding value inside function num = %d \n", num);
}
int main()
{
   int x = 100;
   printf("Before function call x = %d \n", x);
   modify(x);//passing value in function
   printf("After function call x = %d \n", x);
   return 0;
}
```

# Call by value



## Example for call by value:

```c
#include <stdio.h>

void modify(int num)
{
   printf("Before adding value inside function num = %d \n",num);
   num = num + 100;
   printf("After adding value inside function num = %d \n", num);
}
int main()
{
   int x = 100;
   printf("Before function call x = %d \n", x);
   modify(x);//passing value in function
   printf("After function call x = %d \n", x);
   return 0;
}
```

**Output:**

Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100

# Exercise



1. Write two functions MIN and MAX that return the minimum and maximum of two real numbers given as parameters.

2. Write two other functions MIN_4 and MAX_4 that use the functions MIN and MAX to return the minimum and maximum of four real numbers passed as parameters.

3. Write a program to test the two functions MIN_4 and MAX_4.

# Scope and Life time of a variable



Scope of a variable is defined as the region or boundary of the program in which the variable is visible. There are two types:

- Global Scope
- Local Scope

# Global variables



- The variables that are defined outside of any block or function have global scope. These variables are called global variables.
- That is any variable defined in global area of a program (in most of the case, on the top of the C program.) is visible (their life time) from its definition until the end of the program.
- Any function can access variables defined within the global scope
- The global variables are initialized to 0 by the compiler when created.

# Global Variables



```
int n;
void function()
{
  n++;
  printf("call number %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
     function();
}
```

The variable n is a global variable (it has a global scope). It is a permanent variable and it is initialized to zero by the compiler.

# Global Variables



```
int n;
void function()
{
  n++;
  printf("call number %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
     function();
}
```

- The variable n is a global variable (it has a global scope). It is a permanent variable and it is initialized to zero by the compiler.
- The program prints out:

```
call number 1
call number 2
call number 3
call number 4
call number 5
```

# Local variables



- The variables that are defined inside a block have local scope. These variables are called **local variables**.
- They exist only (their life time) from the point of their declaration until the end of the block.
- They are not visible outside the block.
- Local variables of a function are created when a function is called, and they are destroyed when the function returns. Their values are not stored between function calls.

# Local Variables



```
int n = 10;

void function()
{
   int n = 0;
   n++;
   printf("call number %d\n",n);
   return;
}

main()
{
   int i;
   for (i = 0; i < 5; i++)
      Function();
   printf("n = %d\n",n);
}
```

This variable n is a global variable

This variable n is a local to function

# Local Variables



```
int n = 10;

void function()
{
   int n = 0;
   n++;
   printf("call number %d\n",n);
   return;
}

main()
{
   int i;
   for (i = 0; i < 5; i++)
      Function();
   printf("n = %d\n",n);
}
```

This variable n is a global variable

This variable n is a local to function

The program prints out

call number 1
call number 1
call number 1
call number 1
call number 1
n = 10

# Static local variables



- Static local variables are a special type of local variable that are created once, when the function in which they are declared is first called. Their values are then preserved and accessible through all subsequent calls to the function.

- A static local variable is created using the keyword **static**

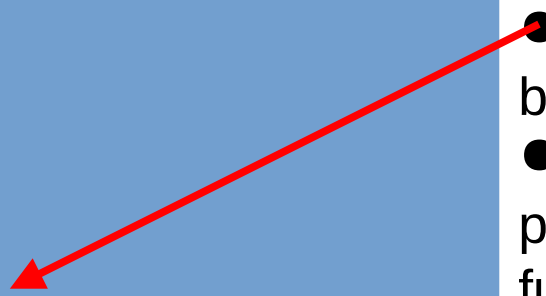
**static** **type** variable_name;

- Static local variables are initialized to 0 by the compiler when created.

# Static local variables



```
int n = 10;

void fct()
{
   static int n;
   n++;
   printf("call number %d\n",n);
}

main()
{
   int i;
   for (i = 0; i < 5; i++)
      fct();
   printf("n = %d\n",n);
}
```

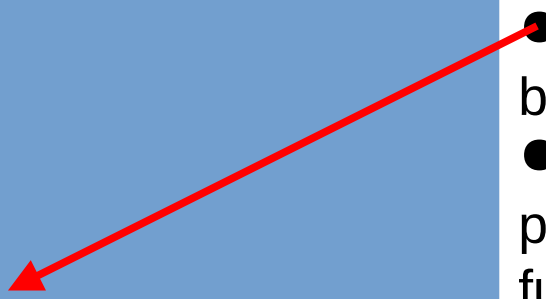


- **n** is a local variable to the function **fct**, but of static class.
- It is initialized to zero, and its value is preserved from one call to the next of the function **fct**.

# Static local variables



```
int n = 10;

void fct()
{
   static int n;
   n++;
   printf("call number %d\n",n);
}

main()
{
   int i;
   for (i = 0; i < 5; i++)
      fct();
   printf("n = %d\n",n);
}
```



- **n** is a local variable to the function **fct**, but of static class.
- It is initialized to zero, and its value is preserved from one call to the next of the function **fct**.

The program prints out:

call number 1
call number 2
call number 3
call number 4
call number 5
n = 10