Department of Computing
Rajarata University of Sri Lanka

COM 1407
Computer Programming

LESSON 06 – FUNCTIONS , SCOPE OF VARIABLES AND PARAMETERS, RECURSION

BY : PIYUMI HERATH

Objectives

By the end of this lecture , students should be able to,

- Understand the importance of user-defined functions.
- Define function definitions and declarations with the usability.
- Have the ability to create functions with arguments passing among functions.
- Understanding of the distinction for passing arguments to/from functions.

1. Functions

Definition and declaration

1.1 What is a function ?

- A function is a block of organized, reusable code that is used to perform a single, specific action.
- Functions provide better modularity for your application and a high degree of code reusing.
- You have already seen various functions like **printf()** and **main()**. These are called built-in functions provided by the language itself, but we can write our own functions as well.

1.2 Why we need functions

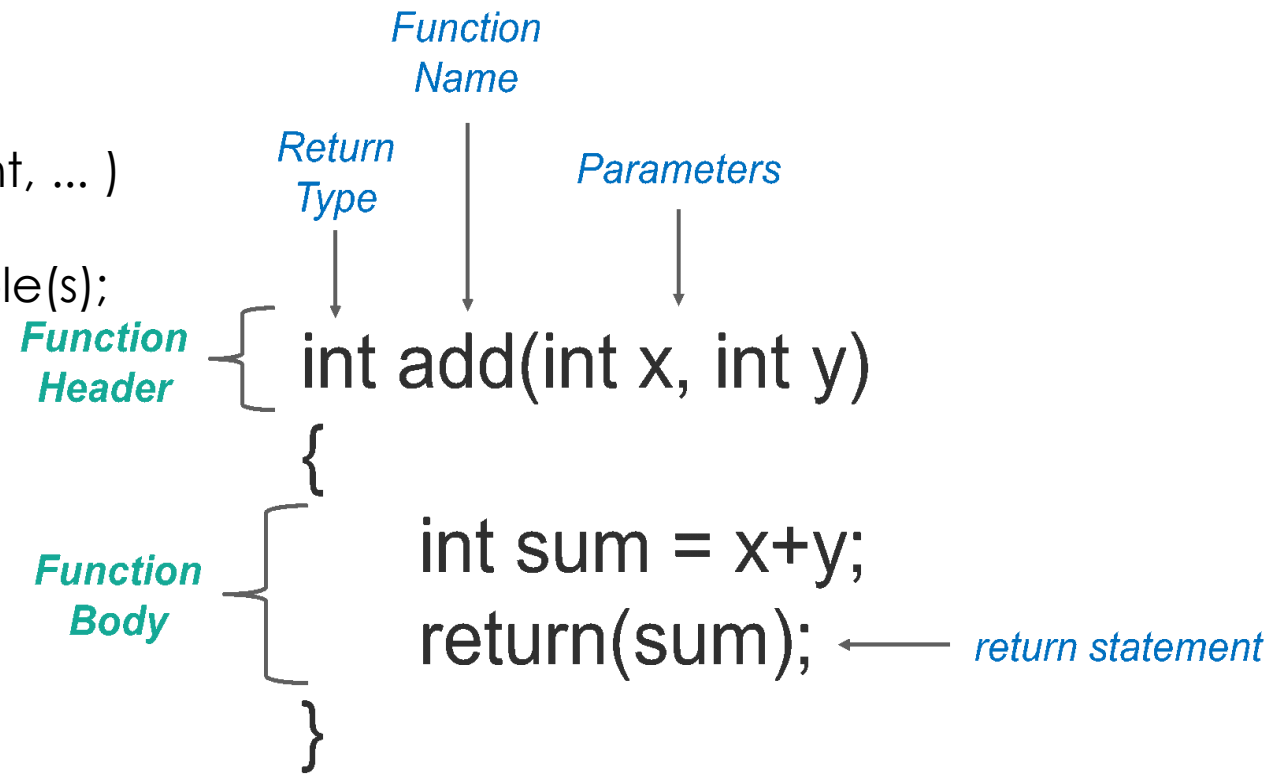
- Very often in computer programs there is some code that must be executed multiple times in different places in the program.
- It is also common to need the same code in multiple different programs.
- Encapsulating frequently-used code into functions makes it easy to re-use the code in different places and/or different programs.
- Functions can be stored in libraries for later re-use.

1.3 Syntax and components

- Defining a function

```
return_type function_name( type argument, ... )
{
        local_variable_type  local_variable(s);
        executable statement(s);

return return_value;
}
```



1.3.1 Return Type

- The "return type" indicates what kind of data this function will return. In the example above, the function returns an int.
- In C, the functions are not mandatory to return a value. The correct way to indicate that a function does not return a value is to use the return type "void". (This is a way of explicitly saying that the function returns nothing.)

1.3.2 Function Name

- The function name is an identifier by which this function will be known, and obeys the same naming rules as applied to variable names
(Alphanumeric characters, beginning with alpha, maximum 31 significant characters, etc.)

1.3.3 Formal parameter list

- Following the function name are a pair of parentheses containing a list of the formal parameters, (arguments) which receive the data passed to the function.
- The ANSI standard requires that the type of each formal parameter to be listed individually within the parentheses as shown in the example. Even if several parameters are of the same type, each must have its type given explicitly.
- If a function takes no parameters, the parameters may be left empty.

- There is a real, functional difference between a function with an empty parameter list and one with an explicitly void parameter list: It is possible to pass parameters to a function with an empty list; the compiler won't complain.

- That is not the same for a function with a void list. Thus, it is possible, and even easy to invoke empty-list functions incorrectly without knowing it, and thereby introduce bugs that can be very difficult to track down.

In **C**:
•void foo() means "a function foo taking an unspecified number of arguments of unspecified type"
•void foo(void) means "a function foo taking no arguments"

1.3.4 Function body

- The body of the function is enclosed within curly {} braces, just as the "main" function with which we have been dealing so far, and contains the instructions that will be executed when this function is called.

1.3.5 Return Statement

- The **return** statement exits the called function and returns control back to the calling function.
  - Once a return statement is executed, no further instructions within the function are executed.
- A single return value (of the appropriate type) may be returned.
  - Parentheses are allowed but not required around the return value.
  - A function with a void return type will not have a return value after the return statement.
- More than one return statement may appear in a function, but only one will ever be executed by any given function call.
  - (All returns other than the last need to be controlled by logic such as "if" blocks.)

1.4 Function Prototype/declarations

- When a function is called, the compiler will check to see that the correct number and types of data items are being passed to the function, and will automatically generate type conversions as necessary. This is known as type checking.
- Type checking is only possible if the compiler already knows about the function, including what kind of data the function is expecting to receive. Otherwise, the compiler has to make assumptions, which can lead to incorrect and erratic behavior if those assumptions are not correct.
- We can make certain that all functions appear earlier in a file than any calls to them.
- A better approach is to use **function prototypes**. This is a way of declaring to the compiler what data a function will require, without actually providing the function itself.

Example:

```
int add( int a, int b );
```

- Function prototypes end with semicolons, indicating that this is not a function, but merely a prototype of a function to be provided elsewhere.
- Variable names are not a must in the function prototype. Naming the parameters in a function prototype helps identify how they'll be used by the function.
- For clarity it is generally good style to list all functions that will be used by prototypes at the beginning of the file. Then provide main( ) as the first full function definition, followed by each of the other functions in the order in which the prototypes are listed.

1.5 Calling a function

- A function is called by using the function name, followed by a set of parentheses containing the data to be passed to the function.
- The data passed to the function are referred to as the "actual" parameters. The variables within the function which receive the passed data are referred to as the "formal" parameters.
- Any function can be called from any other function
- A function can be called any number of times.
- The order in which the functions are defined in a program and the order in which they get called need not necessarily be same.

Passing a value to function as argument

- This is the default way of calling a function in C programming
- When we pass the actual parameters while calling a function then this is known as function call by value. In this case the values of actual parameters are copied to the formal parameters. Thus operations performed on the formal parameters don't reflect in the actual parameters.
- When passing by value, you get a copy of the value. If you change the value in your function, the caller still sees the original value regardless of your changes.

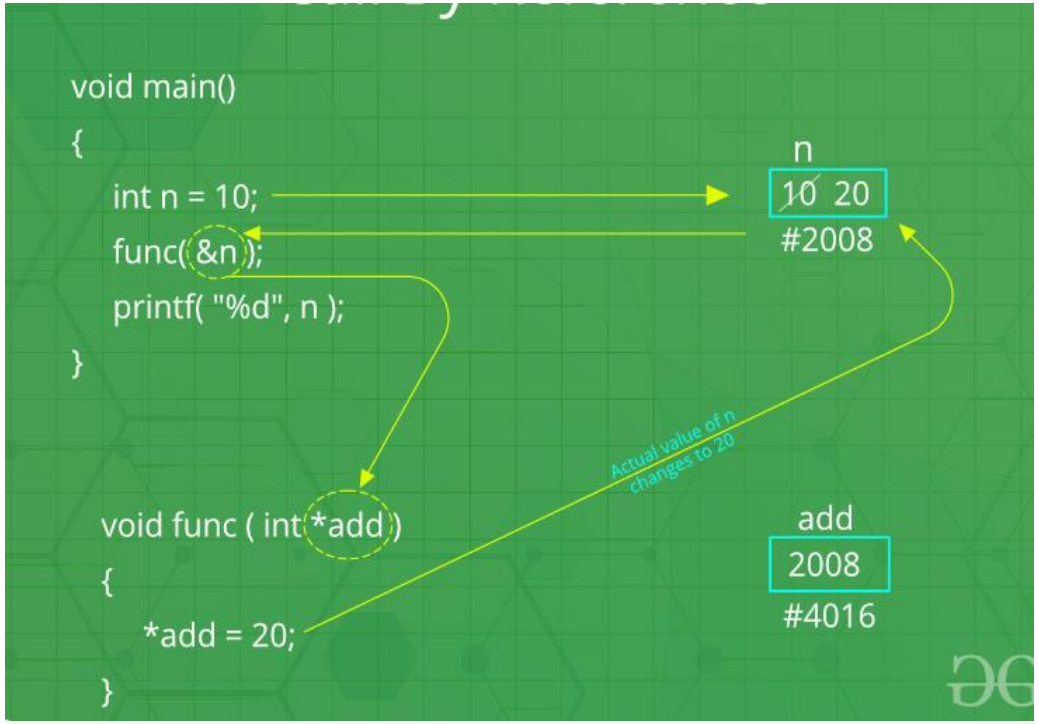
```c
#include <stdio.h>
int sum(int a, int b)
{
     int c=a+b;
     return c;
}

int main(
{
    int var1 =10;
    int var2 = 20;
    int var3 = sum(var1, var2);
    printf("%d", var3);

    return 0;
}
```

Passing a pointer to a function as an argument

- Pointers (pointer variables) are special variables that are used to store addresses rather than values.
- When we pass a pointer as an argument instead of a variable then the address of the variable is passed instead of the value.
- What is passed in is a copy of the pointer, but what it points to is still the same address in memory as the original pointer, so this allows the function to change the value outside the function.



1.6 Returning from a function

- A function may return a single value by means of the return statement.
- Any variable changes made within a function are local to that function.
- A calling function's variables are not affected by the actions of a called function. (When using the normal pass-by-value passing mechanism)
- The value returned by a function may be used in a more complex expression, or it may be assigned to a variable.

```c
#include <stdio.h>

int addNumbers(int a, int b);          // function prototype

int main()
{
    int n1,n2,sum;

    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);          // function call
    printf("sum = %d",sum);

    return 0;
}

int addNumbers(int a, int b)           // function definition
{
    int result;
    result = a+b;
    return result;                     // return statement
}
```

2.Variable scope

A scope is a region of the program, and the scope of variables refers to the area of the program where the variables can be accessed after its declaration.

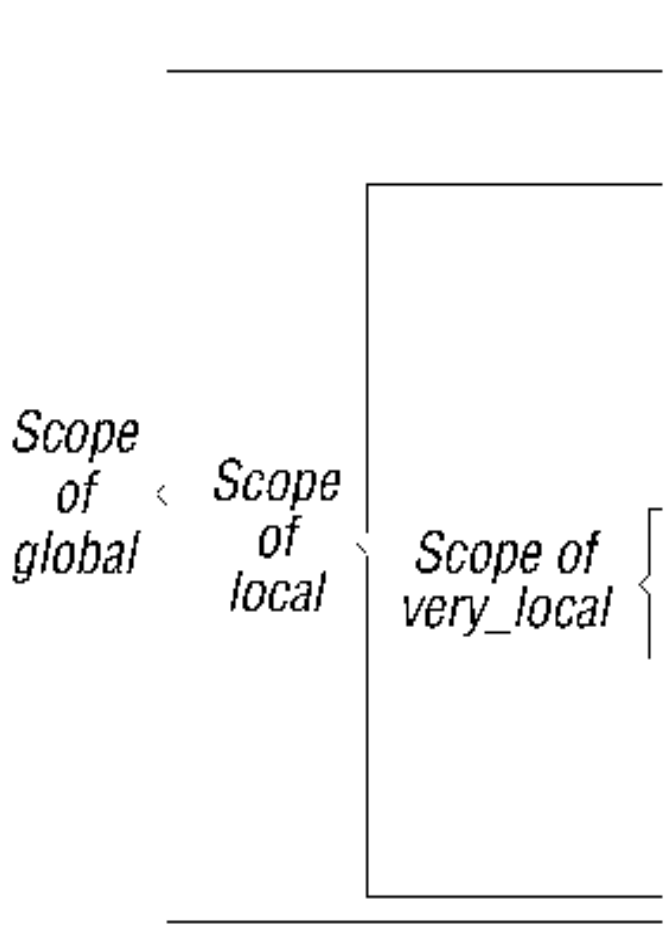
2.1Global Variables

- Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.
- A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

2.2 Local variables

- Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code.
- **The formal parameters are local variables**, which exist during the execution of the function only, and are only known by the called function. They are initialized when the function starts by **copies of** the data passed as actual parameters. This mechanism, known as "pass by value", ensures that the called function can not directly change the values of the calling functions variables.

- Variables defined inside a function are known as *automatic local* variables because they are automatically "created" each time the function is called, and because their values are local to the function.
- The value of a local variable can only be accessed by the function in which the variable is defined. Its value cannot be accessed by any other function.
- If an initial value is given to a variable inside a function, that initial value is assigned to the variable *each* time the function is called.




```
int global;                          /*a global variable*/
main()
{
     int  local;                     /*a local variable*/

     global = 1;                     /*global can be used here*/
     local = 2;                      /*so can local*/

     {                               /*beginning a new block*/
         int  very_local             /*this is local to the block*/

         very_local = global+local;
     }

        /*We just closed the block*/
        /*very_local can not be used*/
}
```

3. Recursive functions

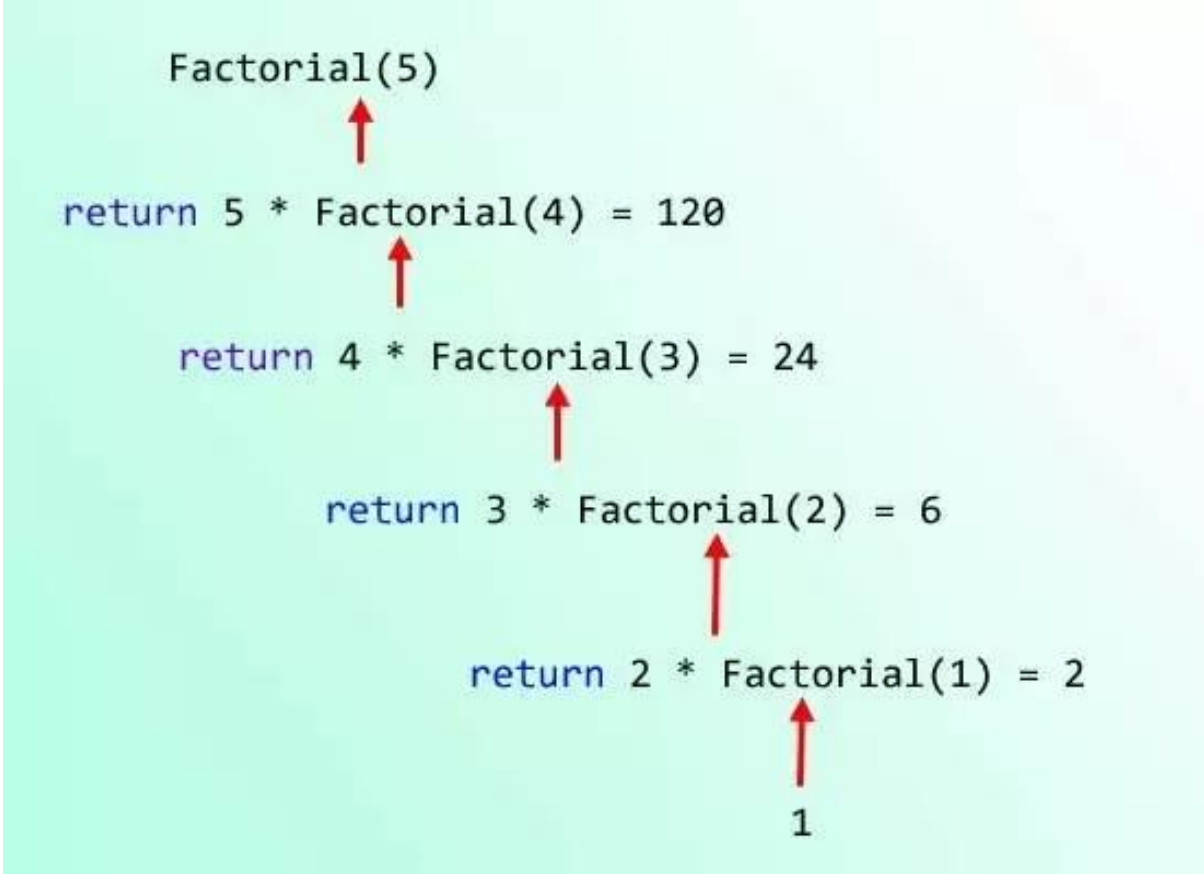
- A function that calls itself is known as a recursive function. And, this technique is known as recursion. A recursive function is a function that calls itself during its execution
- They are commonly used in applications in which the solution to a problem can be expressed in terms of successively applying the same solution to subsets of the problem.
- The recursion continues until some condition is met to prevent it.
- To prevent infinite recursion, if – else statements (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

- Generally, iterative solutions are more efficient than recursion since function call is always overhead.
- Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example ,Fibonacci series, factorial finding, etc.

Example :
Recursive function to find factorial

```
int factorial (int n)
{
    if (n>1)
        return n*factorial(n-1);
    else
        return 1;
}
```

Example


Factorial(5)
return 5 * Factorial(4) = 120
return 4 * Factorial(3) = 24
return 3 * Factorial(2) = 6
return 2 * Factorial(1) = 2
1

References

- Chapter 08 (Pages 119- 162) -Programming in C, 3rdEdition, Stephen G. Kochan

Thank you.

Next lesson : Arrays