# FUNCTIONS

Sercan Külcü | 21.06.2023

# Contents

# 1 Function declaration, definition, and calling

In this chapter, we will dive into the fundamental concept of function declaration, definition, and calling. Functions play a crucial role in organizing and structuring your code, allowing you to break down complex tasks into smaller, manageable parts. So, let's get started!

A function is a self-contained block of code that performs a specific task. It acts as a reusable module, enabling you to write code once and call it multiple times from different parts of your program. Functions provide modularity, improve code readability, and facilitate code maintenance.

Before you can use a function, you need to declare it. Function declaration tells the compiler about the existence, name, return type, and parameter types of the function. It acts as a forward declaration, allowing the compiler to perform type checking and ensure proper usage of the function.

A function declaration follows this general syntax:

return_type function_name(parameter_type1 parameter1, parameter_type2 parameter2, ...);

The return_type specifies the type of value the function returns, which can be void (no return value) or any other data type in C. The function_name is a unique identifier for the function, and the parameter types and names define the input arguments required by the function.

Once you have declared a function, you need to define its implementation. The function definition contains the actual code that gets executed when the function is called. It consists of a function header and a function body.

The function header includes the return type, function name, and parameter list, which must match the declaration. The function body contains the statements enclosed within curly braces { }.

Here's an example of a function definition:

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

   int sum = a + b;

   return sum;
}
```

In this example, we define a function called "add" that takes two integer parameters (a and b). Inside the function body, we calculate the sum of the two parameters and return the result as an integer.

Once a function is declared and defined, you can call it from other parts of your program. Function calling allows you to execute the code within the function and obtain the desired output or perform a specific task.

To call a function, you use its name followed by parentheses. If the function has any required arguments, you provide them within the parentheses. Here's an example of a function call to our "add" function:

```
int result = add(5, 3);
```

In this example, we call the "add" function with arguments 5 and 3. The function executes its code and returns the sum, which is then assigned to the variable "result."

# 2 Function parameters and return values

In this chapter, we will delve into the fascinating realm of function parameters and return values. Understanding how to pass values into functions and retrieve results is vital for building dynamic and interactive programs. So, let's dive right in!

Function parameters are the inputs that you provide to a function when you call it. They allow you to pass values from the calling code to the function, enabling the function to work with specific data. Parameters help make functions flexible and adaptable to different situations.

In C programming, function parameters can have various types, such as integers, floating-point numbers, characters, arrays, pointers, and even structures. The parameter types should match the data types declared in the function declaration and definition.

1. Passing Parameters by Value

By default, parameters in C are passed by value. This means that when you call a function, a copy of the parameter's value is made and used within the function. Any modifications made to the parameter's value within the function do not affect the original value in the calling code.

Let's take a look at an example:

```
void increment(int num) {

   num += 1;

   printf("Inside the function: %d\n", num);

}


int main() {

   int value = 5;

   increment(value);

   printf("Outside the function: %d\n", value);

   return 0;
}
```

In this example, we have a function called "increment" that takes an integer parameter. Within the function, we increment the value of the parameter by 1. However, notice that the change does not affect the original value of "value" in the calling code. This is because the parameter was passed by value, and any modifications were made on a local copy.

2. Passing Parameters by Reference

Sometimes, you may need to modify the original value of a parameter within a function. To achieve this, you can pass the parameter by reference using pointers. By passing the address of a variable, you give the function direct access to the original data, allowing it to modify the value.

Here's an example:

```
void changeValue(int* ptr) {
    *ptr = 10;
}


int main() {
    int value = 5;
    changeValue(&value);
    printf("Modified value: %d\n", value);
    return 0;
}
```

In this example, we define a function called "changeValue" that takes a pointer to an integer as a parameter. Within the function, we dereference the pointer using the * operator and assign a new value to the memory location pointed to by the pointer. As a result, the original value of "value" in the calling code is modified.

Functions not only accept parameters but can also produce return values. A return value is the result or output that a function provides back to the calling code. It allows functions to communicate information or computed results to the rest of the program.

To specify a return value, you declare the return type in the function declaration and definition. A function can have any valid C data type as its return type, including void for functions that do not return a value.

Here's an example:

```
int multiply(int a, int b) {

    int result = a * b;

    return result;

}


int main() {

    int product = multiply(5, 3);

    printf("Product: %d\n", product);

    return 0;

}
```

In this example, we define a function called "multiply" that takes two integers as parameters and returns their product. The return statement inside the function specifies the value to be returned. In the calling code, we assign the returned value to a variable called "product" and display it.

# 3 Function overloading

In this chapter, we'll explore the fascinating concept of function overloading. Function overloading allows you to define multiple functions with the same name but different parameter lists. This powerful feature enhances code organization, readability, and flexibility. So, let's dive right in and uncover the wonders of function overloading!

Function overloading is a technique where you can define multiple functions with the same name but with different parameter types, numbers, or orders. The compiler differentiates between these overloaded functions based on their unique parameter signatures. This allows you to perform similar operations on different data types or handle various cases with a single function name.

One way to overload functions is by using different parameter types. Let's consider a simple example:

```
int sum(int a, int b) {

   return a + b;

}



float sum(float a, float b) {

   return a + b;

}
```

In this example, we have two functions named "sum." The first function takes two integers as parameters and returns their sum as an integer. The second function takes two floats as parameters and returns their sum as a float. By providing different parameter types, we can create multiple versions of the "sum" function to handle different data types.

Another way to overload functions is by varying the number of parameters. Consider the following example:

```
int multiply(int a, int b) {

   return a * b;
}


int multiply(int a, int b, int c) {
   return a * b * c;
}
```

In this example, we define two functions named "multiply." The first function multiplies two integers, while the second function multiplies three integers. By changing the number of parameters, we can create overloaded functions that cater to specific scenarios.

C does not provide direct support for overloading based on parameter order. However, you can achieve similar functionality by using default arguments or function wrappers. Default arguments allow you to specify default values for parameters, while function wrappers act as intermediaries to redirect calls to the appropriate functions.

When you call an overloaded function, the compiler determines the appropriate version of the function to execute based on the function name and the arguments provided. It matches the function call with the closest matching function based on the parameter types, number of parameters, and order of parameters.

If the compiler cannot find an exact match or encounters ambiguous function calls, it generates a compilation error. Therefore, it is crucial to ensure that your overloaded functions have distinct parameter signatures to avoid any ambiguity.

# 4 Handling multiple return values

In this chapter, we will delve into the exciting topic of handling multiple return values in C. While functions typically return a single value, there are situations where you might need to retrieve multiple results from a function. Fear not, for we shall explore various techniques and data structures that enable us to handle multiple return values efficiently. So, let's embark on this enlightening journey!

In C, a function is traditionally designed to return a single value. However, in certain scenarios, you might encounter situations where you need to obtain multiple results from a function. For instance, you may want to calculate the sum and average of a series of numbers or retrieve both the real and imaginary parts of a complex number.

One approach to handling multiple return values is by utilizing pointers. Pointers allow you to indirectly access and modify variables in the calling code, thus enabling a function to return multiple values.

Let's consider an example where we want to calculate the sum and average of a series of numbers:

```
void calculateSumAndAverage(int* numbers, int length, int* sum, float* average) {

   *sum = 0;

   for (int i = 0; i < length; i++) {

      *sum += numbers[i];

   }

   *average = (float)*sum / length;

}
```

In this example, we define a function called "calculateSumAndAverage" that takes an array of numbers, its length, and two pointers as parameters. Within the function, we iterate over the array, calculating the sum and storing it in the memory location pointed to by the "sum" pointer. Similarly, we compute the average and store it in the memory location pointed to by the "average" pointer.

By passing the addresses of variables as arguments to this function, we can retrieve both the sum and average in the calling code.

Another elegant approach to handling multiple return values is by utilizing structures. Structures allow you to group related variables together, making it convenient to return multiple values as a single entity.

Let's take a look at an example:

```
typedef struct {
    int sum;
    float average;
} Result;


Result calculateSumAndAverage(int* numbers, int length) {
    Result result;
    result.sum = 0;
    for (int i = 0; i < length; i++) {
        result.sum += numbers[i];
    }
    result.average = (float)result.sum / length;
    return result;
}
```

In this example, we define a structure called "Result" that contains two members: "sum" and "average." The function "calculateSumAndAverage" takes an array of numbers and its length as parameters. Inside the function, we perform the necessary calculations and store the results in a "Result" structure. Finally, we return the structure, providing access to both the sum and average values.

By using structures, we can encapsulate multiple return values and retrieve them as a cohesive unit in the calling code.

In some cases, you may need to return a collection of values as an array. In C, arrays are passed and returned by reference, meaning you can return an array from a function and use it directly in the calling code.

Consider the following example, where we generate the Fibonacci sequence:

```
int* generateFibonacciSequence(int length) {

   int* sequence = (int*)malloc(length * sizeof(int));

   sequence[0] = 0;

   sequence[1] = 1;

   for (int i = 2; i < length; i++) {

      sequence[i] = sequence[i - 1] + sequence[i - 2];

   }

   return sequence;

}
```

In this example, we allocate dynamic memory for an array of integers using the "malloc" function. We then populate the array with the Fibonacci sequence. Finally, we return the array, allowing us to utilize the generated sequence in the calling code.

Remember to free the dynamically allocated memory once you are done using the array to prevent memory leaks.

# 5 Recursive functions

In this chapter, we will embark on a fascinating exploration of recursive functions. Recursive programming is a powerful technique that allows a function to call itself, enabling elegant and efficient solutions to complex problems. So, fasten your seatbelts and get ready to dive into the captivating world of recursion!

Recursion is a concept where a function solves a problem by breaking it down into smaller, similar subproblems. Each subproblem is then solved by invoking the same function, eventually leading to a base case that terminates the recursion. This recursive process allows for concise and elegant solutions to problems that exhibit repetitive structures.

Recursive functions consist of two essential components: the base case and the recursive case. The base case defines the condition that terminates the recursion, while the recursive case describes how the function calls itself to solve smaller subproblems.

Let's consider a classic example: calculating the factorial of a number.

```
int factorial(int n) {

   // Base case: factorial of 0 or 1 is 1

   if (n == 0 || n == 1) {

      return 1;

   }

   // Recursive case: call the function with a smaller subproblem

   else {

      return n * factorial(n - 1);

   }

}
```

In this example, the base case is when n equals 0 or 1, where the factorial is known to be 1. The recursive case occurs when n is greater than 1. In the recursive case, the function calls itself with a smaller subproblem (n - 1) and multiplies the result by n. This process continues until the base case is reached.

Understanding how recursive functions work can sometimes be challenging. To visualize their execution, it's helpful to imagine a stack of function calls known as the "call stack." Each function call is added to the top of the stack, and as the base cases are reached, the function calls are removed from the stack.

Let's visualize the execution of the factorial function for factorial(4):

```
factorial(4)
  factorial(3)
    factorial(2)
      factorial(1)
        factorial(0)
        return 1
      return 1 * 1 = 1
    return 2 * 1 = 2
  return 3 * 2 = 6
return 4 * 6 = 24
```

As you can see, each recursive call builds upon the previous result until the base case is reached. Then, the results are multiplied together to obtain the final factorial value.

Recursion is a powerful technique that allows us to solve complex problems by breaking them down into smaller, manageable parts. When working with recursive functions, it's important to think recursively and identify the base case and the recursive case. Properly defining these cases is crucial to ensure the termination of the recursion and prevent infinite loops.

Recursion and iteration are two common programming techniques for solving repetitive problems. Both have their strengths and weaknesses, and the choice between them depends on the specific problem and the programming language.

Recursion excels in solving problems with a recursive structure, such as tree traversals or mathematical calculations. It offers concise and elegant solutions, but recursive functions can be memory-intensive and may encounter performance issues for large inputs.

Iteration, on the other hand, uses loops to repeatedly execute a block of code. It is often more efficient in terms of memory and performance. Iterative solutions are suitable for problems that can be solved by sequential processing and do not require repetitive function calls.