Teaching Functions in C Programming Through a Single Simple Example: An Educational Survey of Introductory Computer Science Students

Bojken Shehu

Faculty of Information Technology, Tirana Polytechnic University, Tirana, Albania,
bshehu@fti.edu.al

Abstract: Functions represent a fundamental concept in the C programming language and play an essential role in the development of modular, structured, and reusable programs. For introductory computer science students, understanding how functions operate is a critical step in learning programming, as it supports the development of clear program structure and efficient problem-solving strategies. However, many beginners experience difficulties when first encountering concepts such as function declaration, function definition, parameters, return values, and function calls. This paper examines the teaching of functions in C programming using a simplified instructional approach based on a single illustrative example. Instead of presenting multiple unrelated examples, the proposed approach focuses on one coherent example designed to demonstrate the main concepts associated with functions, including the interaction between user-defined functions and the main() function. This method aims to reduce conceptual complexity and support students in understanding the logical structure of function-based programming. To evaluate the effectiveness of this approach, an educational survey was conducted with 200 first-year computer science students from the Faculty of Information Technology at the Polytechnic University of Tirana. The study combined a conceptual explanation of functions with intentionally simple programming examples and collected student feedback regarding their understanding of the topic. The results suggest that presenting function concepts through a clear and simple example can significantly support students' conceptual understanding and improve the learning experience in introductory programming courses.

Keywords: C Programming Languages, Functions, Modular Programming, Programming Education, Introductory Programming, Computer Science Education.

1 Introduction

The C programming language remains one of the foundational languages in computer science education and software development. Its structured approach and efficiency make it an ideal language for understanding core programming concepts **[3]**. Among these concepts, functions play a crucial role by enabling modular programming, which allows large programs to be divided into smaller and more manageable components **[4]**. Modular programming through functions improves code organization, enhances code reusability, and simplifies debugging and maintenance **[5]**.

These advantages are particularly significant for first-year computer science students who are introduced to fundamental programming principles. Learning how to design and use functions effectively helps students develop structured thinking and problem-solving skills that are essential for advanced software development.

Teaching introductory programming concepts is widely recognized as a challenging task in computer science education [2], [10]. Many novice programmers experience difficulties when learning fundamental concepts such as functions because these topics require understanding both program logic and programming syntax simultaneously.

This article presents a survey of functions in C programming, examining their types, structure, and practical significance. The objective of this study is not only to present the fundamental concepts of functions in the C programming language, but also to examine how simplified programming examples can support the learning process of introductory computer science students. In particular, this study investigates whether presenting function concepts through a **single simple example** can improve student's conceptual understanding and reduce learning difficulties commonly encountered in introductory programming courses.

Based on the educational objectives of this study and the challenges faced by introductory computer science students when learning functions in C programming, the following research questions guide this work:

RQ1: How effectively do first-year computer science students understand the fundamental concepts of functions in C programming, including function declaration, function call, and function definition?

RQ2: Does the use of a single simple programming example help students better understand the interaction between user-defined functions and the `main()` function?

RQ3: How do simplified examples influence students' conceptual understanding of modular programming in introductory programming courses?

The main contribution of this study is the investigation of a simplified instructional approach for teaching functions in C programming using a single coherent programming example. By combining conceptual explanation with an educational survey conducted among first-year computer science students, this study provides empirical insight into how simplified programming examples can support the understanding of fundamental programming concepts.

2 Methodology

The methodology of this study is designed to address the research questions presented above by examining how simple programming examples can support the understanding of functions among first-year students. This study follows a descriptive and educational approach to analyse the role of functions in C programming. The analysis focuses on the fundamental concepts taught in introductory programming courses [5].

To illustrate the discussed concepts, several simple programs were developed and executed using the DevCpp programming environment. The programs presented in this paper are deliberately simple so that the fundamental concept of functions in C can be understood more easily [3]. This approach allows first-year informatics students to focus on the structure of functions, including function declarations, function calls, function definitions, parameters, and return values, without being distracted by complex program logic.

In addition to the conceptual explanation, an educational survey was conducted with 200 first-year computer science students. The survey included a short conceptual questionnaire administered before the lesson (pre-test) and after the lesson (post-test) in order to evaluate students' understanding of function-related concepts. The collected responses were analysed to examine the effectiveness of the proposed instructional approach.

The examples used in this study aim to support beginner programmers in understanding the practical use of functions in program development and to enhance their comprehension of modular programming principles [4].

3 Conceptual Overview of Functions in C Programming

This section provides a conceptual overview of functions in the C programming language. In order to maintain consistency throughout the paper, the same simple programming example is used and gradually extended to illustrate different aspects

of functions, including function declaration, function calls, function definitions, parameter passing, and recursion. This approach allows students to focus on the structural role of functions without being distracted by changes in program logic.

Figure 1 illustrates the conceptual structure of functions in C programming used in this study. The figure summarises the main topics presented to students, including the importance of functions, the main concepts of functions, types of functions, methods of passing parameters, and categories of function definitions. The visual representation was designed to support beginner programmers in understanding the relationships between theoretical concepts and their practical implementation in C programming.

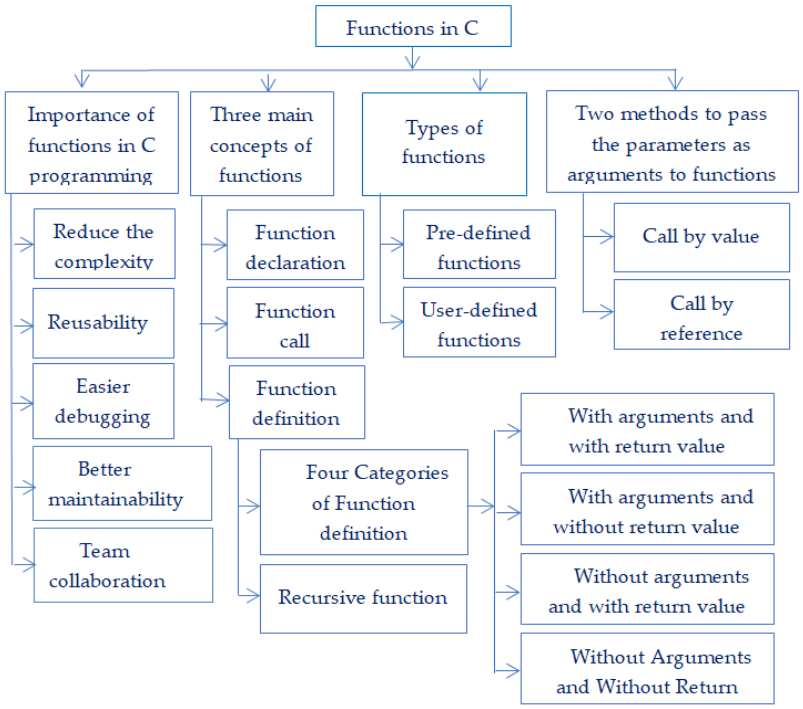

Figure 1
Conceptual overview of functions in C Programming [1]

3.1 What is the Importance of Functions in C Language

Functions play an important role in the C programming language because they help developers organize and manage complex programs. Developers divide the main problem into smaller subprograms in order to reduce program complexity, improve code reusability, and make the debugging process easier. Modularity allows

programs to be divided into manageable components, which enhances software structure and maintainability [6].

Dividing programs into functions reduces complexity, improves code reusability, simplifies debugging, and enhances maintainability, allowing developers to organize programs into manageable components [7]. These subprograms or sub-problems are called functions or modules, and this type of programming is known as modular programming.

3.2 Types of Functions in C Programming Language

There are two types of functions: user-defined functions and predefined functions.

3.2.1 Predefined Functions

Predefined functions are functions whose logic is already provided by the compiler or the standard library. Developers do not need to write the source code for these functions; they simply use them in their programs. For example, the function printf() is used directly without rewriting its implementation. To use predefined functions, the corresponding header files must be included in the program. For instance, for the printf() and scanf() functions, the stdio.h header file must be included, since the logic and declarations of these functions are defined within it. If the developer fails to include the stdio.h header file, these predefined functions will not work properly in the program.

Predefined functions are utilized to improve programming efficiency and reduce development complexity by providing ready-made solutions for common computational tasks in C. These functions are thoroughly tested and widely documented, ensuring reliability and software stability [6]. The use of predefined functions enhances code readability and maintainability, contributing to structured and efficient software development, concepts emphasized in foundational programming and software engineering literature [7].

3.2.2 User-defined Functions

User-defined functions in the C programming language are functions or subprograms that are created by the developer to perform a specific task. Unlike predefined functions (which are provided by the standard library), user-defined functions are written by programmers to implement custom logic. The developer must create the complete logic of the function. User-defined functions must also have a specific name given by the developer. In the C programming language, user-defined functions are usually defined outside the main() function and can be called from within main() or other functions.

Every user-defined function must have a specified data type. The execution of a C program always starts from the main() function, and initially only the main block is executed. In order to execute user-defined functions, developers must provide a reference to them inside the main() function. This reference is called a function call. Like variables, user-defined functions must be declared before they are used in the source code.

The main() function is also a user-defined function, but it has a special role because the compiler only recognizes the `main()` function as the entry point of the program. User-defined functions help to:

a. Reduce code complexity.
b. Improve code reusability.
c. Make debugging easier.
d. Organize the program into smaller, manageable parts [6].

3.3 Three Main Concepts of User-defined Functions in C Programming Language

A function in C Programming Language generally consists of:

3.3.1 Function Declaration

A function declaration (prototype) informs the compiler about the function's name, return type, and parameters. It allows the compiler to recognize the function before its actual definition.

a. The function declaration must be done above the main() function.
b. Syntax: return_type function_name (list of data_types);
c. A semicolon (;) is mandatory at the end of the function declaration.

Function declarations improve code structure and enable the compiler to verify function usage before execution, a fundamental principle of structured programming [6].

3.3.2 Function Call

Function call: It is used to execute the function definition from main() or another function definition. It triggers the execution of the function's code.

d. The function call must be done inside the `main()` function.
a. Syntax: function_name (list of parameters or arguments);
b. A semicolon (;) is mandatory at the end of the function call.

Function calls enable code reuse and modular program execution, which improves software maintainability and readability [7].

3.3.3 Function Definition

Function definition: It contains the actual logic of the function. It specifies what operations the function will perform when called.

a. A function definition in C can be placed outside the main() function, which means it can appear either before or after main(). If the function definition is placed before the main() function, a separate function declaration (prototype) is not required. However, if the function definition appears after main(), a function declaration must be provided before main() to inform the compiler about the function.

b. Syntax: return_type function_name (list of parameters)

```
{
    local variable declaration;
    executable statements;
    return;
}
```

c. In C programming, function definitions do not end with a semicolon.

Function definitions encapsulate program logic and support modular programming by separating tasks into independent units [6].

To illustrate the concepts of function declaration, function call, and function definition, a simple program that calculates the sum of three numbers is used. The program was developed and executed using Dev-C++.

```
//Example 1. C program to find sum of three numbers.
#include<stdio.h>
int add(int, int, int); //The function declaration must be done above the main().
int main()
{
  int a,b,c,sum=0;
  printf("Enter the values of a, b, and c :");
  scanf("%d%d%d",&a,&b,&c);
  sum=add(a,b,c);//The function call must be done inside the main().Calling function.
  printf("The sum is = %d.",sum);
 return 0;
}
//The function definition can be placed outside the main() function. Called Function.
int add(int x,int y,int z)
{
   int d;
   d=x+y+z;
   return d;
}
```

Compilation and execution of the program.

```
Enter the values of a, b, and c : 10 5 3
The sum is = 18.
```

Example 1

C Program to find sum of three numbers [1]

In the function declaration, add(int,int,int) accepts three parameters of integer data type and returns an integer value to the parent function. The execution of the function definition is triggered through a function call written inside the main() function. In other words, in order to execute the function definition, the developer must call it from the main() function. This reference inside main() is called a function call. [3],[6].

The function call must use the same function name and the same number of parameters as defined in the function declaration and function definition. If the function declaration specifies three parameters, then the function call must also provide three values (for example, a, b, and c). These values are passed to the user-defined function add, and the result is stored in shuma. Modular function calls and parameter passing support structured programming and code reusability, which are essential principles in introductory computer science education [7], [8].

In the function definition (child function), we write the logic for addition, and in the end, we return the value to the `main()` function (parent function). The function in which we write the function call is called the parent function or calling function. The function in which we write the logic is called the function definition or child function, and it is referred to as the called function.

The parameters used in the function call are called actual parameters, while the parameters defined in the function definition are called formal parameters. Any function created by the developer will return control to the main() function (parent function). The return may be either an empty return (void return) or a value return. In the Example 1, the user-defined function returned the result to the calling function inside the main() function, demonstrating function communication and result passing [3], [6], [8].

In Example 2, the result is printed inside the function definition itself therefore, an empty return is sent back to the calling function in the main() function.

```c
//Example 2. C program to find sum of three numbers.(Empty return)
#include<stdio.h>
void add(int, int, int); //The function declaration must be done above the main().
int main()
{
   int a,b,c,sum=0;
   printf("Enter the values of a, b, and c :");
   scanf("%d%d%d",&a,&b,&c);
   add(a,b,c); //The function call must be done inside the main(). Calling function.
  return 0;
}
//The function definition can be placed outside the main(). Called function.
void add(int x, int y, int z)
{
   int d;
   d=x+y+z;
   printf("The sum is = %d.",d);
}
```

Actual parameters

Formal parameters

If the output statement is written inside the user-defined function itself, a return type is not required.

Compilation and execution of the program.

```
Enter the values of a, b, and c : 10 5 3
The sum is = 18.
```

Example 2

Program to find sum of three numbers (Empty return) [1]

3.4 Four Categories of Function Definitions in C Programming Language

In the C programming language, function definitions can be classified into four categories depending on whether they use arguments and return values. These categories help programmers understand different ways in which functions can interact with other parts of a program and support modular software design [6].

a) Function definition with arguments and with return value.
b) Function definition with arguments and without return value.
c) Function definition without arguments and with return value.
d) Function definition without arguments and without return value.

To explain the four categories of function definitions, **the same program that calculates the sum of three numbers is used as an illustrative example**. Modular function design, such as these categories, improves code organization and reusability, principles emphasized in software engineering literature [7].

3.4.1 Function Definition with Arguments and with Return Value

A **function definition with arguments and a return value** receives input values (arguments), processes them, and then returns a result to the program. In Example

3 below, the function definition takes three numbers as arguments and returns their sum.

```c
//Example 3. Function definition with arguments and with return value
#include<stdio.h>
int add(int, int, int); //Function declaration with arguments and with return value.
int main()
{
  int a,b,c,sum=0;
  printf("Enter the values of a, b, and c :");
  scanf("%d%d%d",&a,&b,&c);
  sum=add(a,b,c); //Function call with arguments and with return value.
  printf("The sum is = %d.",sum);
  return 0;
}
//Function definition with arguments and with return value. Called function
int add(int x,int y,int z)
{
   int d;
   d=x+y+z;
   return d;
}
```

Compilation and execution of the program.

```
Enter the values of a, b, and c : 10 5 3
The sum is = 18.
```

Example 3

Function definition with arguments and with return value [1]

3.4.2 Function Definition with Arguments and without Return Value

A **function with arguments and without a return value** receives input values (arguments) but does not return any result to the main() function. It performs a specific task, such as displaying or processing the given values.

In Example 4 below, we have a **function definition** that takes three inputs as arguments but does not return any value to the main() function. The function definition performs the summing internally without sending a result back.

```c
//Example 4. Function definition with arguments and without return value.
#include<stdio.h>
void add(int,int,int);//Function declaration with arguments and without return value
int main()
{
   int a,b,c;
   printf("Enter the values of a, b, and c :");
   scanf("%d%d%d",&a,&b,&c);
   add(a,b,c); //Function call with arguments and without return value.
  return 0;
}
//Function definition with arguments and without return value. Called function.
void add(int x,int y,int z)
{
   int sum=0;
   sum=x+y+z;
   printf("The sum is = %d.",sum);
}
```

Compilation and execution of the program.

```
Enter the values of a, b, and c : 10 5 3
The Sum is = 18.
```

Example 4

Function definition with arguments and without return value [1]

3.4.3 Function Definition without Arguments and with Return Value

A function without arguments and with a return value in Example 5 calculates the sum of three predefined numbers inside the function definition and returns the result to the main program. The main program can then use the returned value for further processing or display.

```c
//Example 5. Function definition without arguments and with return value.
#include<stdio.h>
int add(); //The function declaration without arguments and with return value.
int main()
{
   int sum=0;
   sum=add(); //The function call without arguments and with return value.
   printf("The sum is = %d.",sum);
   return 0;
}
//Function definition without arguments and with return value. Called function.
int add()
{
   int a,b,c,d=0;
   printf("Enter the values of a, b, and c :");
   scanf("%d%d%d",&a,&b,&c);
   d=a+b+c;
  return d;
}
```

Compilation and execution of the program.

```
Enter the values of a, b, and c : 10 5 3
The sum is = 18.
```

Example 5

Function definition without arguments and with return value [1]

3.4.4 Function Definition without Arguments and without Return Value

In Example 6, the function definition reads the three inputs directly from the keyboard and does not receive them from the main function. All operations are performed within the function definition, and in the main() function there is only the function call because in the main() function we can remove all instructions except the function call.

```c
//Example 6. Function definition without arguments and without return value.
#include<stdio.h>
void add(); //The function declaration without arguments and without return value.
int main()
{
  add(); //The function call without arguments and without return value.
  return 0;
}
//Function definition without arguments and without return value. Called function.
void add()
{
    int a,b,c,sum=0;
    printf("Enter the values of a, b, and c :");
    scanf("%d%d%d",&a,&b,&c);
    sum=a+b+c;
    printf("The sum is = %d.",sum);
}
```

Compilation and execution of the program.

```
Enter the values of a, b, and c : 10 5 3
The sum is = 18.
```

Example 6

Function definition without arguments and without return value [1]

3.5 Recursive Functions

Recursive functions in C language are functions that call themselves to solve a problem by breaking it into smaller sub-problems. They must have a base condition to stop recursion and prevent infinite function calls.

```c
/*Example 7. A recursive function to sum n numbers reads input values from the keyboard and adds them using recursive calls.*/
int sumRecursive(int n)
{
   int num;
   if (n == 0)
   return 0;  // Base case.
   printf("Enter number : ");
   scanf("%d", &num);
  return num + sumRecursive(n - 1);  // Recursive function call
}
int main()
{
   int total;
   total = sumRecursive(3);  // Add three numbers
   printf("The sum is = %d.\n", total);
  return 0;
}
```

Compilation and execution of the program.

```
Enter number : 10
Enter number : 5
Enter number : 3
The sum is = 18.
```

Example 7

C program to find sum of three numbers, using recursive function [1]

3.6 Call by Value and Call by Reference

Two methods to pass parameters as arguments from function call to function definition are Call by Value and Call by Reference.

3.6.1 Call by Value.

In Call by Value, a copy of the variable's value is passed to the function definition from function call in the main() function, so changes inside the function definition do not affect the original variable in function call in main() function. Example 1-4.

The actual parameters in function call are copied into the formal parameters of the function definition. The original parameters remain unchanged because only their values are duplicated and passed to the function definition [6]. The actual parameters in function call have their own memory addresses, while the formal parameters in function definition are stored in different memory locations. If the function definition modifies the formal parameters, only their values change, while the values of the actual parameters in function call remain the same because they do not share the same memory location [6].

3.6.2 Call by Reference.

In Call by Reference, the memory address of the variable is passed, allowing the function to modify the original variable.

```
// Example 8. C program to find sum of three numbers, using call by reference
#include<stdio.h>
//The Function declaration with arguments and without return value.
void add(int *, int *, int *, int *);
int main()
{
   int x,y,z,result;
   printf("Enter the values of x, y, and z :");
   scanf("%d%d%d",&x, &y, &z);
   add(&x, &y, &z, &result); //Actual parameters.
   printf("The sum is = %d.",result);
   return 0;
}
/*Function definition with arguments and with return value. Called function. The function add receives the addresses of the variables. */
void add(int *a, int *b, int *c, int *result) //Formal parameters
{
 //Access the values stored at those addresses. The calculated sum is stored in result.
   *result = *a + *b + *c;
}
```

Compilation and execution of the program.

```
Enter the values of x, y, and z : 10 5 3
The sum is = 18.
```

Example 8

C program to find sum of three numbers, using call by reference[1]

In the **call by reference** mechanism, the actual parameters and the formal parameters refer to the same memory locations. Any modification made to the formal parameters in function definition directly affects the actual parameters in function call because the function definition works with their addresses.

4 Results of the Educational Survey

To evaluate the effectiveness of the teaching approach based on a single simple example, a short survey was conducted with 200 first-year computer science students. The students were asked to complete a short conceptual questionnaire before the lesson (pre-test) and after the lesson (post-test). The results are presented in Figure 2.

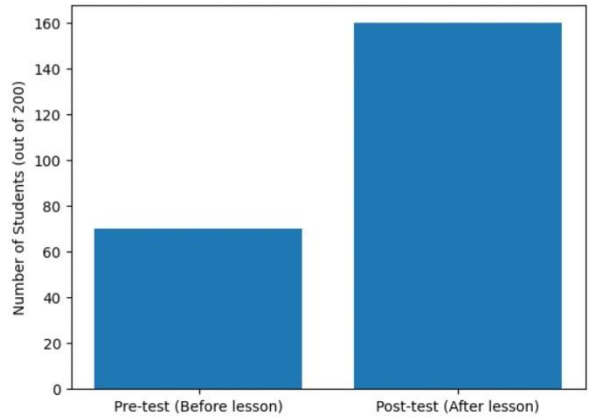

Figure 2

Understanding of C Functions Before and After the Lesson

As shown in Figure 2, only 70 out of 200 students correctly understood the basic concepts of functions before the lesson. After the lesson, the number increased to 160 students. This improvement indicates that presenting the concepts through a single simple example significantly helped students understand how functions operate in C programming.

Table 1 summarizes the results of the conceptual questionnaire administered before and after the instructional session. The results show a clear improvement in students' understanding of functions after the lesson.

Test Type	**Number of Students(out of 200)**	**Percentage**
Pre-Test(Before the Lesson)	70	35%
Post-Test (After the Lesson)	160	80%

Table 1

Summary of Pre-Test and Post-Test Results

The number of students who correctly understood the basic concepts of functions increased from 70 students in the pre-test to 160 students in the post-test. This represents an improvement of approximately **45 percentage points**, indicating a substantial increase in students' conceptual understanding after the instructional explanation.

In addition to measuring conceptual understanding, students were also asked to evaluate whether the use of a single simple programming example helped them better understand the concept of functions.

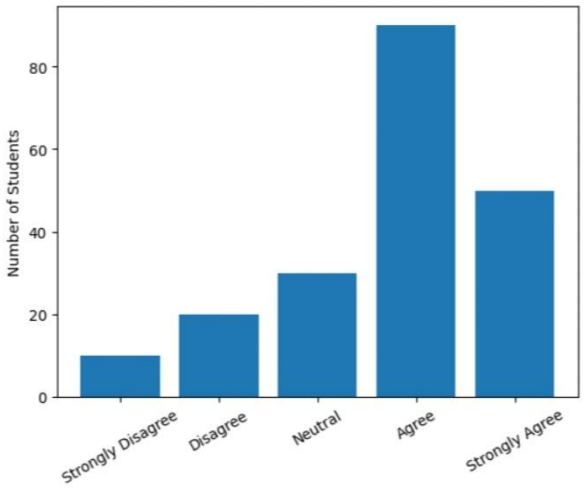

Figure 3

Student Perception of the Use of a Single Simple Example

As shown in Figure 3, the majority of students positively evaluated the teaching approach. Most students selected *Agree* or *Strongly Agree*, indicating that the use of a simple and focused example helped them understand the relationship between function declaration, function call, and function definition. Only a small number of students expressed disagreement, while some students remained neutral, as summarized in Table 2.

Response	**Number of Students(out of 200)**
Strongly Disagree	10
Disagree	20
Neutral	30
Agree	90
Strongly Agree	50

Table 2

Student Perception of the Teaching Approach

5 Discussion

The use of functions significantly improves program structure and readability by allowing complex problems to be divided into smaller and more manageable components [6]. For first-year computer science students, learning how to design and use functions represents an important step toward understanding modular programming and structured software design [7]. Functions encourage students to develop systematic problem-solving strategies and support the principles of structured programming, which are fundamental in introductory programming education.

Furthermore, understanding the different categories of functions helps students learn how data can be transferred between program components through arguments and return values. This process strengthens students' ability to organize program logic and improves their comprehension of code modularity, reusability, and maintainability [3], [8].

The results of the educational survey conducted in this study provide empirical support for these observations. The comparison between the pre-test and post-test results shows a clear improvement in students' understanding of function-related concepts after the instructional explanation and programming examples. In particular, the percentage of students who correctly understood the basic concepts of functions increased from **35% in the pre-test to 80% in the post-test**, indicating a substantial improvement in conceptual understanding.

In addition, the student perception survey indicates that most participants considered the use of a **single simple programming example** helpful in understanding the relationship between function declaration, function call, and function definition. These findings suggest that simplified instructional examples can play an important role in reducing conceptual complexity for novice programmers.

From an educational perspective, presenting programming concepts through clear and focused examples may help students concentrate on the fundamental logic of functions without being distracted by unnecessary program complexity. This approach aligns with educational research in computer science that emphasizes the importance of structured explanations and carefully designed examples for improving conceptual understanding in introductory programming courses. Similar challenges in teaching introductory programming concepts have also been discussed in previous studies on programming education, which highlight the difficulties novice programmers face when learning fundamental programming structures [2], [10**]**.

6 Limitations of the Study

This study has several limitations that should be considered when interpreting the results. First, the survey was conducted with students from a single institution, which may limit the generalizability of the findings to other educational contexts. Second, the evaluation focused primarily on students' conceptual understanding immediately after the lesson, and long-term retention of the concepts was not measured. Future studies could involve students from multiple institutions and examine the long-term impact of simplified instructional examples on programming learning outcomes.

Conclusions

Functions represent one of the most important concepts in the C programming language and play a key role in the development of structured and modular programs. By dividing complex problems into smaller subprograms, functions improve program organization, code reusability, and maintainability [6].

This article presented an overview of functions in C programming, including predefined and user-defined functions, as well as the concepts of function declaration, function call, and function definition. In addition, the four main categories of function definitions were discussed and illustrated through simple programming examples designed for introductory computer science students.

The results of the educational survey conducted with first-year students indicate that the use of simplified programming examples can support students' conceptual understanding of functions and their role in modular programming. The comparison between pre-test and post-test results suggests that presenting function concepts through a clear and coherent example can significantly improve students' comprehension of how functions operate within a C program.

Overall, the findings highlight the importance of using structured explanations and simple instructional examples when introducing fundamental programming concepts. Such approaches can help novice programmers better understand the logical structure of programs and support the development of effective programming skills in introductory computer science education.

Future research may explore the effectiveness of the single-example instructional approach in teaching other programming concepts such as loops, arrays, and data structures.

Acknowledgement

The author would like to express gratitude to the first-year Computer Science students whose learning experiences inspired the preparation of this study. Their interaction during programming courses has helped highlight the importance of

presenting fundamental programming concepts, such as functions in C, through clear explanations and simple examples.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

References

[1] Shehu, B: Functions in C Programming. Lecture Notes, 2019, Polytechnic University of Tirana. https://sites.google.com/view/bojkenshehu/c-language

[2] Bennedsen, J., Caspersen, M. E: Failure rates in introductory programming. ACM SIGCSE Bulletin, 2007, 39(2), 32–36.

[3] Kernighan, B. W., Ritchie, D. M: The C Programming Language, 2nd ed, 1988, Prentice Hall.

[4] Schildt, H: C: The Complete Reference ,3rd ed, 2000, McGraw-Hill.

[5] Deitel, P., Deitel, H: C How to Program, 8th ed, 2016, Pearson.

[6] Pressman, R. S., Maxim, B. R: Software Engineering: A Practitioner's Approach, 2014, McGraw-Hill.

[7] Sommerville, I: Software Engineering, 10th ed, 2015, Pearson.

[8] Tempero, E: An Experiment on the Effects of Modularity on Code, *2023*, ACM. https://kblincoe.github.io/publications/2023_ACE_Modularity.pdf

[9] Lavy, I: The Circumstances in which Modular Programming becomes the Favor Choice by Novice Programmers, 2018, IJ–MECS. https://www.mecs-press.org/ijmecs/ijmecs-v10-n7/IJMECS-V10-N7-1.pdf

[10] Robins, A., Rountree, J., Rountree, N*:* Learning and teaching programming:A review and discussion. Computer Science Education, 2003, 13(2), 137–172.