Contents

1 Subprograms: Functions Procedures **3**
1.1 Introduction . . . 3
1.2 Definition and Purpose of Subprograms . . . 3
1.2.1 Why Use Subprograms? . . . 3
1.3 Types of Subprograms . . . 3
1.3.1 Functions . . . 4
1.3.2 Procedures . . . 4
1.4 Function Syntax and Calling . . . 4
1.5 Function Call and Execution Flow . . . 5
1.5.1 Memory Allocation During Function Call . . . 5
1.6 Call by Value vs. Call by Reference . . . 5
1.6.1 Call by Value . . . 5
1.6.2 Call by Reference . . . 5
1.7 Local and Global Variables . . . 6
1.7.1 Local Variables . . . 6
1.7.2 Global Variables . . . 6
1.8 Best Practices for Functions . . . 7
1.9 Exercises . . . 7
1.10 Recursive functions . . . 8
1.10.1 Definition . . . 8
1.10.2 Basic Structure of a Recursive Function . . . 8
1.11 Exercises . . . 9

Chapter 1

Subprograms: Functions Procedures

1.1 Introduction

Subprograms are a fundamental concept in C programming. They help in structuring code, improving readability, and enabling reusability. This course covers the definition, declaration, types, and implementation of subprograms in C, supported with examples and exercises.

1.2 Definition and Purpose of Subprograms

A **subprogram** is a block of code that performs a specific task and can be reused multiple times. It takes inputs, processes them, and returns an output.

1.2.1 Why Use Subprograms?

Subprograms allow us to:

- Improve code readability and structure.
- Reuse code without rewriting it multiple times.
- Facilitate error detection and debugging.
- Break complex problems into smaller, manageable parts (modularity).
- Use blocks of code without needing to understand their internals (abstraction).

1.3 Types of Subprograms

There are two main types of subprograms:

1.3.1 Functions

A **function**:

- Takes input parameters (optional).
- Performs a specific task.
- Returns a value as output.

Example: Function to calculate the sum of two numbers:

1.3.2 Procedures

A **procedure**:

- Takes input parameters (optional).
- Performs a specific task.
- Does **not** return a value.

Example: Procedure to print a message on the screen.

1.4 Function Syntax and Calling

In C, a function is defined using the following syntax:

```
return_type function_name(parameters) {
// Function_body
return value; // Optional
}
```

- **return_type** specifies the data type of the returned value.
- **function_name** is a unique identifier.
- **parameters** (optional) provide input values.
- **function_body** contains the set of instructions.

Example: Function that prints the sum of two numbers:

```
void printSum(int a, int b) {
printf("Sum: %d\n", a + b);
}

int main() {
```

```
6 printSum(5, 3);
7 return 0;
8 }
```

1.5 Function Call and Execution Flow

A function must be called to execute. When called, execution control temporarily transfers to the function, executes statements, and returns control back to the caller.

1.5.1 Memory Allocation During Function Call

When a function is called, a new stack frame is allocated for storing:

- Function parameters.
- Local variables.
- Return address.

After execution, the frame is removed from the stack (Last In, First Out - LIFO mechanism).

1.6 Call by Value vs. Call by Reference

1.6.1 Call by Value

A copy of actual parameters is passed to the function, meaning modifications do not affect the original variables.

Example:

```
1  void add(int a, int b) {
2  int sum = a + b;
3  printf("Sum: %d", sum);
4  }
5
6  int main() {
7  int x = 10, y = 20;
8  add(x, y);
9  return 0;
10 }
```

1.6.2 Call by Reference

Instead of passing values, we pass memory addresses, allowing functions to modify actual data.

Example:

```c
void swap(int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp;
}

int main() {
int x = 10, y = 20;
swap(&x, &y);
printf("x = %d, y = %d", x, y);
return 0;
}
```

> **Note:** Arrays in C are always passed by reference.

1.7 Local and Global Variables

1.7.1 Local Variables

Defined inside a function and accessible only within it.

Example:

```c
void function() {
int localVar = 10;
printf("Local variable: %d\n", localVar);
}

int main() {
function();
return 0;
}
```

1.7.2 Global Variables

Defined outside all functions, accessible throughout the program.

Example:

```c
int globalVar = 20;

void function() {
printf("Global variable: %d\n", globalVar);
}

int main() {
function();
```

```
 9 printf("Global␣variable␣in␣main:␣%d\n", globalVar);
10 return 0;
11 }
```

> **Note:** If a local and global variable have the same name, the local variable takes precedence.

1.8 Best Practices for Functions

- **Single Responsibility**: Each function should perform one task.
- **Descriptive Names**: Use meaningful names.
- **Limit Parameters**: Keep functions simple.
- **Documentation**: Use comments.
- **Testing**: Test functions independently.

1.9 Exercises

1. Write a function that returns the maximum value in an integer array.
2. Write a function that reverses an array.
3. Write a function that prints the frequency of each unique element in an array.

1.10 Recursive functions

So far we have been using loops to repeat certain statements under some conditions, but it's not the only way. In a matter of fact, we can use something else called recursive functions.

1.10.1 Definition

Recursion is a process in which a function calls itself. Therefore, a recursive function is a function that calls itself to solve smaller versions of the same problem.

```
1 fun(){
2     fun();
3 }
```

1.10.2 Basic Structure of a Recursive Function

In general, a recursive function will have this basic structure.

```
1  fun(){
2     if() {
3         //base case
4     }
5     else {
6         //recursive case
7     }
8 }
```

- **Base Case**: The condition that stops the recursion. It is like a "stop button" for recursion.
- **Recursive Case**: The part of the function where it calls itself to solve a smaller part of the problem.

To write a recursive function we need to follow these steps:

1. Specify the base case to stop the recursion.
2. Divide the initial problem into smaller versions. In other words, we need to break the problem into smaller parts to reduce the size of the initial problem.

Example : Write a recursive function to compute the factorial of a number.

In order to write a recursive function to calculate the factorial of any given number, first we must specify the base case. To find the base case, think of an instance of the problem which is very easy and straightforward to solve without any computation.
For the factorial, we know that **factorial(1) = 1**, so we can take it as a base case.

The second step is to divide the problem into smaller parts.

Step-by-Step for n = 4:

```
factorial(4) = 4 * factorial(3)
             = 4 * (3 * factorial(2))
             = 4 * (3 * (2 *factorial(1)))
             = 4 * (3 * (2 * 1)) = 24
```

After this, there is nothing much left to be done, we just have to specify the base condition related to the base case. The base condition is one that does not require calling the function again and is used to stop the recursion.

For the factorial, the base condition can be $(n == 1)$.

```
1 int factorial(int n) {
2     if (n == 1) // Base case: Stop when n reaches 1
3         return 1;
4     else
5         return n * factorial(n - 1); // Recursive case: Break into n *
              (n-1)!
6 }
```

This example demonstrates how recursion breaks down a large task into smaller subproblems.

> **Note:** we must specify the base condition to terminate the recursion, otherwise we get an infinite loop and execution error of type stackoverflow.

1.11 Exercises

1. Write a recursive function to compute the sum of the first n numbers.
2. Write a recursive function to compute X raised to the power of n.
3. Write a recursive function to compute the nth term of the Fibonacci sequence.
4. Write a recursive function that displays all even numbers smaller or equal to n.
5. Write a recursive function to determine the length of a given string.
6. Implement a recursive function to print a string in reverse order.
7. Implement a recursive function to compute the greatest common divisor (GCD) of two integers.