One-Week-C-Primer (Day 3)

Day 3: Functions, Recursion, and Storage Class

Note : Refer to the Attached Document along with this for the codes

Functions

Introduction to Functions in C:

A function in C is a self-contained block of code that performs a specific task or set of tasks. It is a reusable unit of code designed to perform a specific operation. Functions play a crucial role in breaking down a complex problem into smaller, manageable parts, making the code modular, organized, and easy to maintain.

Advantages of Functions in C:

1. **Modularity:** Functions enable modularity in the code by breaking it into smaller logical units. Each function can handle a specific task, making it easier to understand and maintain the codebase.
2. **Code Reusability:** Functions can be reused multiple times in different parts of the program or even in different programs. This reduces code duplication and saves development time.
3. **Easy Debugging:** With functions, you can focus on debugging individual parts of the code independently, which simplifies the debugging process and improves code reliability.
4. **Abstraction:** Functions allow you to hide the implementation details and expose only the necessary interface to the outside world. This provides a level of abstraction and makes the code more readable and easier to use.
5. **Code Organization:** By organizing code into functions, the overall structure of the program becomes more organized and easier to manage, especially for large projects.
6. **Scalability:** Functions facilitate code scalability. As the project grows, you can add new functions to handle new tasks without affecting the existing code.
7. **Testing and Maintenance:** Functions make testing and maintenance more manageable. You can test each function independently, and if any changes are required, you only need to modify the relevant function.

8. **Improved Collaboration:** Dividing a program into functions allows multiple programmers to work on different parts simultaneously. This enhances collaboration and teamwork in software development.

Syntax of Function in C:

```
return_type function_name(parameter1_type parameter1_name, parameter2_type parameter2_nam
e, ...) {
    // Function body (code that performs the task)
    // ...
    return return_value;
}
```

Example of a Simple Function in C:

```
#include <stdio.h>

// Function to calculate the square of a number
int square(int num) {
    return num * num;
}

int main() {
    int num = 5;
    int result = square(num);
    printf("The square of %d is %d\n", num, result);
    return 0;
}
```

Output:

```
The square of 5 is 25
```

In the above example, the function `square()` calculates the square of a number. This small function demonstrates the advantages of using functions in C, including modularity, code reusability, and improved code organization. As programs become more complex, functions become an essential tool for creating efficient and maintainable code.

User Defined Functions in C

Creating and using user-defined functions is a fundamental aspect of C programming. Let's go through the process of creating a user-defined function, explaining the syntax and usage, along with a practical example:

Creating a User-Defined Function:

In C, you can create a user-defined function by defining its signature, specifying the return type, function name, and parameters (if any), followed by the function body. The function signature defines the function's name and its parameters' data types.

Syntax of Function in C:

```
return_type function_name(parameter1_type parameter1_name, parameter2_type parameter2_nam
e, ...) {
    // Function body (code that performs the task)
    // ...
    return return_value; // Optional, only for functions with a non-void return type
}
```

Example: Creating a Simple User-Defined Function:

```
#include <stdio.h>

// Function to add two integers and return the result
int add(int a, int b) {
    int sum = a + b;
    return sum;
}

int main() {
    int num1 = 10, num2 = 20;
    int result = add(num1, num2);
    printf("The sum of %d and %d is %d\n", num1, num2, result);
    return 0;
}
```

Output:

```
The sum of 10 and 20 is 30
```

Using User-Defined Functions:

1. To use a user-defined function, you must declare or define the function before calling it. Typically, you define the function before the `main()` function, or you can use a function prototype before `main()` if the function is defined later in the code.
2. Function calls follow the syntax of the function signature, with actual arguments (values) being passed to the function's parameters.

Function Prototypes:

A function prototype is a declaration of a function that provides the function's signature without the actual function body. It allows the compiler to know about the function's existence and its signature before it is defined or used in the code.

Syntax of Function Prototype:

```
return_type function_name(parameter1_type parameter1_name, parameter2_type parameter2_nam
e, ...);
```

Example: Using a Function Prototype for Forward Declaration:

```
#include <stdio.h>

// Function prototype (forward declaration)
int multiply(int a, int b);

int main() {
    int num1 = 5, num2 = 4;
    int result = multiply(num1, num2);
    printf("The product of %d and %d is %d\\n", num1, num2, result);
    return 0;
}

// Function definition
int multiply(int a, int b) {
    int product = a * b;
    return product;
}
```

Output:

```
The product of 5 and 4 is 20
```

In the example above, we use a function prototype for the `multiply()` function before `main()`, allowing the `main()` function to know about the existence and signature of the `multiply()` function. The actual function definition is provided later in the code.

Using user-defined functions in C enhances code modularity, readability, and reusability. As programs grow larger and more complex, breaking down the code into smaller functions makes it easier to maintain and understand. User-defined functions are an essential tool for efficient and organized C programming.

Recursion

Recursion is a powerful programming technique in which a function calls itself to solve a problem by breaking it down into smaller, similar sub-problems. In the context of functions, recursion is the process of a function invoking itself during its execution. Recursive functions allow solving complex problems more elegantly by dividing them into smaller instances of the same problem until a base case is reached, where the function stops calling itself and starts returning results.

Recursive Function Structure:

A recursive function consists of two main components:

1. Base Case: The base case is a condition that specifies when the recursion should stop. It is the simplest version of the problem that can be directly solved without further recursion.
2. Recursive Case: The recursive case is where the function calls itself to solve a smaller version of the same problem. The function keeps calling itself repeatedly until the base case is reached.

Example of a Recursive Function: Factorial Calculation:

One classic example of a recursive function is computing the factorial of a non-negative integer.

Factorial of n (denoted as n!) is the product of all positive integers less than or equal to n.

Recursive Function to Calculate Factorial:

```c
#include <stdio.h>

int factorial(int n) {
    // Base case: factorial of 0 is 1
    if (n == 0) {
        return 1;
    }
    // Recursive case: factorial of n is n * factorial(n-1)
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num = 5;
    int result = factorial(num);
    printf("Factorial of %d is %d\\n", num, result);
    return 0;
}
```

Output:

```
Factorial of 5 is 120
```

In the above example, the function `factorial()` is defined recursively to calculate the factorial of a number. When `factorial()` is called with a value of `5`, it recursively calls itself with `factorial(4)`, `factorial(3)`, `factorial(2)`, `factorial(1)`, and finally, `factorial(0)`. The base case (`n == 0`) is reached, and the recursion stops. Then, the values are multiplied in reverse order, and the final result of `120` is returned.

Important Considerations for Recursive Functions:

1. Recursive functions must have a well-defined base case to ensure termination. Without a base case, the function will enter an infinite loop and cause a stack overflow.
2. Recursive functions may consume more memory as each recursive call adds a new activation record to the function call stack. Excessive recursion can lead to a stack overflow, especially for large values of input.
3. Iterative solutions are often more memory-efficient than recursive solutions for some problems. Therefore, careful consideration should be given to the problem's nature before deciding to use recursion.

4. Recursive solutions are generally more concise and easier to understand for certain problems, especially those that exhibit self-similarity and overlapping sub-problems.

In summary, recursion is a powerful technique for solving problems by breaking them down into smaller instances of the same problem. Recursive functions allow elegant and concise code to handle complex problems. However, it's essential to define a proper base case and be cautious of stack overflow issues when using recursion.

Storage Classes , Lifetime and Visibility of variables

In C, storage classes determine the scope, lifetime, and visibility of variables within a program. There are four storage classes in C:

1. **auto:**
    - The 'auto' storage class is the default storage class for all local variables declared within a function or block.
    - Variables declared as 'auto' are stored in the stack memory.
    - The 'auto' keyword is rarely used explicitly, as it is assumed by default for local variables.
2. **static:**
    - The 'static' storage class is used to declare variables with a static duration and is shared across different function calls.
    - Variables declared as 'static' are stored in the data segment of the memory.
    - A 'static' local variable retains its value between function calls and is initialized only once.
    - In the case of a global variable, 'static' limits its scope to the current source file, making it accessible only within that file.
3. **extern:**
    - The 'extern' storage class is used to declare variables that are defined in other source files.

- 'extern' is used to provide a forward declaration of the variable, indicating that the variable is defined elsewhere.
- It is commonly used to share variables across multiple source files.

4. **register:**
    - The 'register' storage class is used to suggest the compiler to store the variable in a CPU register for faster access.
    - The 'register' keyword is a hint to the compiler, and the compiler may or may not choose to store the variable in a register, depending on various factors.

Scope of Variables:

The scope of a variable determines the region of the program where the variable is visible and can be accessed.

1. **Block Scope:** Variables declared within a block (within a function or a compound statement) have block scope. They are accessible only within that block.
2. **Function Scope:** Variables declared as function parameters have function scope and are accessible within that function.
3. **File Scope:** Global variables and variables declared with the 'static' storage class have file scope and are accessible throughout the entire source file where they are declared.
4. **Prototype Scope:** Function prototypes also have scope. When you declare a function prototype, it introduces the function's name and its parameter types within its scope.

Lifetime of Variables:

The lifetime of a variable refers to the period during which the variable exists in the memory and retains its value.

1. **Automatic (Local) Variables:** Variables with 'auto' storage class have automatic duration. They are created when the block containing the variable is entered and destroyed when the block is exited.
2. **Static Variables:** Variables with 'static' storage class have static duration. They are initialized only once before the program starts and retain their value between different function calls.
3. **Global Variables:** Global variables have static duration, similar to static variables. They are initialized only once before the program starts and retain their value throughout the

program's execution.

4. **Register Variables:** The lifetime of register variables is the same as automatic variables. They are created when the block containing the variable is entered and destroyed when the block is exited. However, the compiler may choose not to store them in memory and instead use CPU registers for faster access.

Example Demonstrating Storage Classes, Scope, and Lifetime:

```
#include <stdio.h>

int globalVar = 10; // Global variable with static storage class

void demo() {
    auto int localVarAuto = 5; // Automatic (local) variable
    static int localVarStatic = 20; // Static local variable

    printf("Local auto variable: %d\\n", localVarAuto);
    printf("Local static variable: %d\\n", localVarStatic);
}

int main() {
    extern int externVar; // Declaration of an external variable (defined elsewhere)

    demo();

    printf("Global variable: %d\\n", globalVar);
    printf("Extern variable: %d\\n", externVar);

    return 0;
}
```

In this example, we demonstrate different storage classes and their scope and lifetime. 'localVarAuto' is an automatic variable, 'localVarStatic' is a static local variable, 'globalVar' is a global variable, and 'externVar' is an external variable. Their scope and lifetime are appropriately demonstrated within the 'demo()' function and 'main()' function. The 'externVar' is defined in another source file, but we use its declaration in this code to access its value.
