Functions in C Programming

1. Brief Overview

- A function is a block of code that performs a specific task.
- It helps in dividing a program into small, reusable and manageable parts.
- Functions are used to avoid repetition of code and make the program more organized and easy to maintain.
- In C, a function can take input (arguments) and can return a value (or nothing).

2. Defining a Function

- A function is defined using a return type, function name, parameter list and a block of statements enclosed in curly braces { }.

Syntax:

```
return_type function_name(parameter_list)
{
    // statements
}
```

Example:

```
int add(int a, int b)
{
    int sum = a + b;
    return sum;
}
```

→ Here, add() is a function that takes two integers and returns their sum.

3. Accessing (Calling) Functions

- To use a function, we call (or access) it by writing its name along with the arguments (if any).
- The function call is placed inside another function (usually main()).

Example:

```
#include <stdio.h>
int add(int a, int b);  // function prototype (optional)
int main()
{
    int x = 10, y = 20, result;
    result = add(x, y);  // calling the function
    printf("Sum = %d\n", result);
    return 0;
}
```

```
int add(int a, int b)
{
    int sum = a + b;
    return sum;
}
```

In this example, the main() function calls the add() function and uses the returned value.

Passing Arguments to Function and Specifying Argument Data Types in C Programming

1. Passing Arguments to a Function

- Arguments are the values that are passed to a function when it is called.
- They allow the function to work with different data each time it is called.
- In C, arguments are passed to a function using the function call syntax (in the parentheses).
- The values are passed in the same order as the parameters are declared in the function definition.

Syntax:

```
function_name(argument1, argument2, ...);
```

← function call

↑ values passed as arguments

Example:

```
int add(int a, int b);   // function prototype
int main()
{
    int x = 5, y = 7, sum;
    sum = add(x, y); // passing arguments x and y
    printf("Sum = %d", sum);
    return 0;
}
```

→ Here, x and y are passed as arguments to the add() function in the function call add(x, y).

2. Specifying Argument Data Types

- The data type of the arguments is specified in the function definition (parameter list).
- It tells the compiler what type of data the function expects.
- The same data types must be used when passing the arguments in the function call.
- If the data types do not match, it can lead to errors or unexpected results.

Example:

```
int multiply(int a, int b) // parameters with data types
{
    int result = a * b;
    return result;
}
```

Here, a and b are specified as int, so the function expects integer values.

In the function call, we must pass integer values:

```
int product = multiply(5, 10);  // passing int arguments
```

Thus, by specifying the data types in the parameter list and passing the appropriate values in the function call, we can correctly use arguments in C programming.

Function Prototypes in C Programming

1. Function Prototype

- A function prototype is a declaration of a function that tells the compiler about the function's name, return type and parameter types (without giving the actual body).
- It is used to inform the compiler about the function before it is called, especially when the function is defined after the main() function or in a different file.
- It helps in type checking and prevents errors.

Syntax:

```
return_type  function_name(parameter_list);
```

- `return_type` ↕ return type of function
- `function_name` ↕ name of function
- `parameter_list` ↓ list of parameters (with data types)

Example:

```
int add(int a, int b);   // function prototype
```

→ This tells the compiler that there is a function named add() which returns an integer and takes two integer parameters.

Another Example:

```
void display(void);  // no parameters
float average(int x, int y);  // two integer parameters
```

Recursion in C Programming

1. What is Recursion?

- Recursion is a technique in which a function calls itself in order to solve a problem of smaller size.
- It continues calling itself until a base condition is met, and then returns the result.
- It is useful for problems that can be divided into smaller subproblems, such as factorial, Fibonacci series, etc.

2. Key Points

- A recursive function must have a base case (to stop the recursion).
- Each recursive call should move closer to the base case.
- It uses the call stack to keep track of function calls.

3. Example: Factorial using Recursion

```c
#include <stdio.h>
int fact(int n)        // recursive function
{
    if (n <= 1)        // base case
        return 1;
    else
        return n * fact(n - 1);  // recursive call
}
int main()
{
    int n, result;
    printf("Enter a number: ");
    scanf("%d", &n);
    result = fact(n);
    printf("Factorial of %d = %d", n, result);
    return 0;
}
```

How it works:

1. fact(5) is called.
2. Since $5 \neq 1$, it calls fact(4).
3. This continues until fact(1) is called.
4. At fact(1), base case is true, so it returns 1.
5. The values are returned back:
   fact(1) = 1 → fact(2) = 2 →
   fact(3) = 6 → fact(4) = 24 →
   fact(5) = 120.

Recursion makes the code shorter and easier to understand for many problems.

Strings in C Programming

1. String Declaration

- A string is a sequence of characters, terminated by a null character ('\0').
- In C, a string is actually an array of characters.
- It is declared using the char data type.
- The size of the string is the number of characters (excluding the null character).

Syntax:

```
char string_name[size];
```

Example:

```
char name[20];    // string declaration
char msg[50];     // another string
```

Here, name and msg are arrays of characters that can store strings.

- To initialize a string at the time of declaration:

Syntax:

```
char string_name[] = "string";
```

Example:

```
char city[] = "Delhi";
```

– Here, the string "Delhi" is stored in the array city.

2. String Functions

- C provides several built-in string functions in the <string.h> header file to perform various operations on strings.
- These functions make string handling easy and efficient.

Important String Functions:

Function	Description	Example
strlen()	Returns the length of a string.	len = strlen(str);
strcpy()	Copies one string to another.	strcpy(dest, src);
strcat()	Concatenates (appends) one string to another.	strcat(str1, str2);
strcmp()	Compares two strings.	result = strcmp(str1, str2);
strchr()	Finds the first occurrence of a character in a string.	p = strchr(str, 'a');
strstr()	Finds the first occurrence of a substring in a string.	p = strstr(str, "abc");
strrev()	Reverses a string. (Non-standard function, available in some compilers.)	strrev(str);
strlwr()	Converts a string to lowercase.	strlwr(str);
strupr()	Converts a string to uppercase.	strupr(str);

Example:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[20] = "Hello";
    char str2[20] = "World";
    strcat(str1, str2);   // str1 becomes "HelloWorld"
    printf("%s\n", str1);
    return 0;
}
```

String functions save time and effort while working with strings in C programming. They are very useful for input, output and text processing.

String Manipulation in C Programming

1. What is String Manipulation?

- String manipulation refers to performing various operations on strings such as copying, concatenating, comparing, finding length, etc.
- C provides a set of built-in string functions in <string.h> header file to make string handling easy.

2. Important String Functions

Function	Description	Example
strcpy()	Copies one string to another.	strcpy(dest, src);
strcat()	Concatenates (appends) one string to another.	strcat(str1, str2);
strcmp()	Compares two strings.	strcmp(str1, str2);
strlen()	Returns the length of a string.	strlen(str);
strchr()	Finds the first occurrence of a character in a string.	strchr(str, ch);
strstr()	Finds the first occurrence of a substring in a string.	strstr(str, substr);
strrev()	Reverses a string (non-standard function, available in some compilers).	strrev(str);
strlwr()	Converts a string to lowercase.	strlwr(str);
strupr()	Converts a string to uppercase.	strupr(str);

Example:

```
#include <stdio.h>
#include <string.h>
int main()  {
    char str1[50] = "Hello";
    char str2[50] = "C Programming";
    strcat(str1, str2);    // Concatenate strings
    printf("Result: %s\n", str1);
    printf("Length: %lu\n", strlen(str1));
    return 0;
}
```

String manipulation functions help to handle and process text data efficiently in C programming.

Program Structure and Storage in C Programming

1. Program Structure

- A C program follows a fixed structure and is divided into four main parts:

1. Preprocessor Directives - (e.g. #include, #define)
2. Global Declarations - (e.g. variable, function prototypes)
3. main() Function - (entry point of the program)
4. User Defined Functions - (other functions written by the programmer)

Example of a C program structure:

```
#include <stdio.h>        // Preprocessor directive

int add(int, int);        // Function prototype

int main() {
    int a, b, sum;        // Local variables
    sum = add(a, b);
    printf("Sum = %d\n", sum);
    return 0;
}
```

2. Storage in C Programming

- C uses different types of memory to store data. Each variable or function has a specific storage location.

Types of Storage:

1. Stack - Stores local variables, function parameters and return addresses. (automatic, temporary)
2. Heap - Used for dynamic memory allocation (malloc, calloc, realloc, free).
3. Static Storage - Retains value throughout the program's life (static variables).
4. Global Storage - Variables declared outside all functions. Accessible throughout the file.

Example:

Variable	Storage Type	Scope
int x; (inside function)	Stack	Local
static int y;	Static	Local (retains value)
int g; (outside function)	Global	Global
int *p = malloc(10);	Heap	Dynamic

Class: Automatic, External and Static Variables in C Programming

1. Automatic Variables

- Automatic variables are the default type of local variables in C.
- They are created when a block (function or loop) is entered and destroyed when the block is exited.
- They are stored in the stack memory.
- Their value is not retained between function calls.

Example:

```
void func() {
    int x = 10;    // automatic variable
    printf("x = %d", x);
}
```

Note:
If you call func() again, x will be created again with the initial value 10.

2. External Variables

- External variables are declared outside any function (global scope).
- They can be accessed by any function in the program using the extern keyword (if declared in another file).
- They are stored in the data segment (global memory).
- Their value is retained throughout the program execution.

Example:

```
int count = 0;    // external (global) variable
void show() {
    printf("Count = %d", count);
}
```

Note:
If you want to use an external variable in another file, you can declare it with
extern int count;

3. Static Variables

- Static variables are also local to a function, but they do not lose their value after the function ends.
- They are initialized only once and retain their value in the next call.
- They are stored in the data segment (not stack).

Example:

```
void counter() {
    static int x = 0;    // initialized only once
    x++;
    printf("x = %d", x);
}
```

Note:
- If you call counter() multiple times, the value of x will keep increasing (1, 2, 3,...).

In short:

Automatic → local, stack, lost after function
Static → local, data segment, retained
External → global, data segment, retained