Unit 7
Functions

Definition of Function

- A function is a block of code that performs a specific task.
- The function contains the set of programming statements enclosed by {}.
- A function can be called multiple times to provide reusability and modularity.
- The function is also known as procedure or subroutine in other programming languages.

Types of Functions

- There are two types of functions in C programming:

Library Functions:

- Library functions are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.

User-defined functions:

- User-defined functions are the functions which are created by the C programmer, so that he/she can use it many times.
- It reduces the complexity of a big program and optimizes the code.

Function Aspects

- There are three aspects of a C function.

Function declaration:

- A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
  Syntax:
  ```
  return_type function_name (argument list);
  ```

Function call Function:

- A function can be called from anywhere in the program.
- The parameter list must not differ in function calling and function declaration.
- We must pass the same number of functions as it is declared in the function declaration.
  Syntax:
  ```
  function_name (argument_list);
  ```

Function definition:

- It contains the actual statements which are to be executed.
- It is the most important aspect to which the control comes when the function is called.
- Here, we must notice that only one value can be returned from the function.
  Syntax:

```
return_type function_name (argument list)
{
        function body;
}
```

Example:

```
#include<stdio.h>
void display(); //Function Declaration
int main()
{
        display();        //Function Call
}
void display()   //Function Definition
{
        printf("I am a function");
}
```

Function Arguments and Return Types

- Functions based on the return values and arguments.
    - Function with No argument and No return type.
    - Function with No argument and Return something.
    - A function that takes argument but returns nothing.
    - Functions that take an argument and also return something.

Example:

```
#include<stdio.h>
int add(int, int); //Function Declaration
int main()
{
        int a = 6, b = 8, sum = 0;
        sum = add(a, b);        //Function Call
        printf("The sum is :%d", sum);
}
int add(int x, int y)      //Function Definition
{
        return x+y;
}
```

Output:

```
The sum is :14
```

Nested and Recursive Function

- Defining a function in terms of itself is called recursion.
- We call a method that calls itself a recursive method.
- Recursive algorithms can be done iteratively each recursive call has its own distinct set of parameters and local variables.
- A recursive call is a separate entry on the execution stack.

Parts of recursion:

i. *The Basis Case:*
   In order for recursion to work correctly, every recursive method must have a basis case. The basis case is an input for which the method does not make a recursive call. The basis case prevents infinite recursion.

ii. *Reduced case that works to the base case:*
   Determine the general case.

Recursive Example

```
#include<stdio.h>
int power(int, int);
int main()
{
        int testpower;
        testpower = power(5, 2);
       printf("%d", testpower);
}

/* function definition */

int power(int base, int exp)
{
   if (exp == 0)
         return 1;
   else
         return (base * power(base, exp-1));
}
```

Output:

```
25
```

Call by value and Call by reference

Call by value

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we cannot modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Call by reference

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Example: Call by value

```
#include <stdio.h>
void swap(int, int);
int main ()
{
        /* local variable definition */
        int a = 100;
        int b = 200;

        printf("Before swap, value of a : %d\n", a );
        printf("Before swap, value of b : %d\n", b );

        /* calling a function to swap the values */
        swap(a, b);

        printf("After swap, value of a : %d\n", a );
        printf("After swap, value of b : %d\n", b );
        return 0;
}
void swap(int x, int y)
{
        int temp;
        temp = x; /* save the value of x */
        x = y;    /* put y into x */
        y = temp; /* put temp into y */
}
```

Output:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
```

Example: Call by reference

```
#include <stdio.h>
void swap(int*, int*);
int main ()
{
        /* local variable definition */
        int a = 100;
        int b = 200;

        printf("Before swap, value of a : %d\n", a );
        printf("Before swap, value of b : %d\n", b );

        /* calling a function to swap the values */
        swap(&a, &b);

        printf("After swap, value of a : %d\n", a );
        printf("After swap, value of b : %d\n", b );
        return 0;
}
void swap(int *x, int *y) {
        int temp;
        temp = *x; /* save the value of x */
        *x = *y;    /* put y into x */
        *y = temp; /* put temp into y */
}
```

Output:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

Passing Arrays to Function

First way:

return_type function(type arrayname[])

Declaring blank subscript notation [] is the widely used technique.

Second way:

return_type function(type arrayname[SIZE])

Optionally, we can define size in subscript notation [].

Third way:

*return_type function(type *arrayname)*

Pointer can also be used.

Example

```
// Program to calculate the sum of array elements by passing to a function
#include <stdio.h>
float calculateSum(float num[]);
int main()
{
        float result, num[] = {23.4, 55, 22.6, 3, 40.5, 18};
        // num array is passed to calculateSum()
        result = calculateSum(num);
        printf("Result = %.2f", result);
        return 0;
}

float calculateSum(float num[])
{
        float sum = 0.0;
        for (int i = 0; i < 6; ++i)
        {
                sum += num[i];
        }
        return sum;
}
```

Output

```
Result = 162.50
```

Passing Strings to Function

Function declaration to accept one dimensional string

```
returnType functionName(char str[]);
```

Example

```
void displayString(char str[]);
```

Example

```
#include <stdio.h>
void displayString(char []);
int main(void)
{
        // variables
        char
        message[] = "Hello World";
        // print the string message
        displayString(message);
        return 0;
}
void displayString(char str[])
{
        printf("String: %s\n", str);
}
```

Output:

```
String: Hello World
```

Scope, visibility and lifetime of a variable

- Scope determines the region in a C program where a variable is available to use,
- Visibility of a variable is related to the accessibility of a variable in a particular scope of the program and
- Lifetime of a variable is for how much time a variable remains in the system's memory.

Scope

- Scope is defined as the availability of a variable inside a program, scope is basically the region of code in which a variable is available to use.
- There are four types of scope:
  1. file scope
  2. block scope
  3. function scope and
  4. prototype scope

1. File Scope

- File scope of variables in C is defined as having the availability of the variable throughout the file/program.
- It means that the variable has a global scope, and it is available all around for every function and every block in the program.

Example

```
#include <stdio.h>
// variable with file scope
int x = 10;

void func() {
  // x is available in func() function,
  // x now equals 10 + 10 = 20
  x += 10;
  printf("Value of x is %d\n", x);
}

int main() {

  func();
  // x is also available in main() function
  x += 30; // x now equals 20 + 30 = 50
  printf("Value of x is %d", x);
  return 0;
}
```

Output

```
Value of x is 20
Value of x is 50
```

2. block scope

- Block scope of variables in C is defined as when the variable has a limited scope, and the memory occupied by the variable will be deleted once the execution of the block ends.
- The variable is not accessible or available outside the block.
- A block of code can be defined in curly braces {code_block}.

Example

```
#include <stdio.h>
int main() {
  int a = 5;
  int b = 10;

  // inner block of code having block scope
  {
    int sum = a + b;
    printf("Sum of a and b: %d", sum);
  }

  // the below statement will throw an error because,
  // sum variable is not available outside the scope of above block,
  // printf("Sum of a and b: %d", sum);

  return 0;
}
```

Output

```
Value of x is 20
Value of x is 50
```

3. Function Scope

- Function scope of variables in C begins with the left curly brace { and ends with a closing right curly brace }.
- A variable declared inside a function has a function scope. It has been allocated memory when the function is called, and once the function returns something, the function execution ends and with it, the variable goes out of scope, i.e. it gets deleted from the memory.

Example

```
#include <stdio.h>

void findAge() {
  // the age variable is not accessible outside the function findAge()
  // as it is having local scope to the function i.e. function scope
  int age = 18;
}

int main() {

  printf("Age is %d", age);
  return 0;
}
```

Output

```
[Error] 'age' was not declared in this scope
```

4. Function Prototype Scope

- Function prototype scope of variables in C are declared in some function as its parameters.
- These variables are similar to the function scope variables where a variable's memory gets deleted once the function execution terminates.

Example

```
#include <stdio.h>

// variables a and b are available only inside the function and
// both variables have function prototype scope
int findSum(int a, int b) {
  return a + b;
}

int main() {
  int sum = findSum(3, 5);
  printf("Sum of 3 and 5 is %d", sum);
  return 0;
}
```

Output

```
Sum of 3 and 5 is 8
```

Visibility

- Visibility of a variable is defined as if a variable is accessible or not inside a particular region of code or the whole program.

Example

```
// C Program to demonstrate that a variable is not accessible
// even if it is available in a block of code.
#include <stdio.h>

int main() {
  int scope; // outer scope variable
  scope = 10;

  // inner block of code
  {
    float scope; // inner scope

    scope = 2.98;

    printf("Inner block scope : %f\n", scope);
  }

  printf("Outer block scope : %d\n", scope);

  return 0;
}
```

Output

```
Inner block scope : 2.980000
Outer block scope : 10
```

Lifetime

- Lifetime of a variable is the time for which the variable is taking up a valid space in the system's memory, it is of three types:
  a. static lifetime
  b. automatic lifetime and
  c. dynamic lifetime

a. Static Lifetime

- Objects/Variables having static lifetime will remain in the memory until the execution of the program finishes.
- These types of variables can be declared using the static keyword, global variables also have a static lifetime: they survive as long as the program runs.

Example:

```
static int count = 0;
```

The count variable will stay in the memory until the execution of the program finishes.

b. Automatic Lifetime

- Objects/Variables declared inside a block have automatic lifetime.
- Local variables (those defined within a function) have an automatic lifetime by default: they arise when the function is invoked and are deleted (together with their values) once the function execution finishes.

Example:

```
{ // block of code
        int auto_lifetime_var = 0;
}
```

auto_lifetime_var will be deleted from the memory once the execution comes out of the block.

c. Dynamic Lifetime

- Objects/Variables which are made during the run-time of a C program using the Dynamic Memory Allocation concept using the malloc() and calloc() functions are stored in the memory until they are explicitly removed from the memory using the free() function.
- These variables are said to have a dynamic lifetime.
- These variables are stored in the heap section of our system's memory, which is also known as the dynamic memory.

Example:

```
int *ptr = (int *)malloc(sizeof(int));
```

Memory block (variable) pointed by ptr will remain in the memory until it is explicitly freed/removed from the memory or the program execution ends.

Scope Rules (Global, Local, and Formal)

- Scope of variables in C also have some rules demonstrated below. Variables can be declared on three places in a C Program :

i. Variables that are declared inside the function or a block of code are known as *local variables.*
ii. Variables that are declared outside of any function are known as *global variables* (usually at the start of a program).
iii. Variables that are declared in the definition of a function parameters as its *formal parameters.*

Exercise

1. Why do we need a break and continue statement? Define formal argument and actual argument in function with examples. Identify and list the errors in the following code. (10) [TU 2079]

```
int main(){
   int a,b,c
   scanf("%d%d%d, &a, &b, &c);
   sum(a, b, c);
   return -1;
}
void sum(int x, int y, int z){
   int sum;
   sum = a + b + c;
   return sum;
}
```

2. Write a program to find the sum of digits of a given integer using recursion. (5) [TU 2079]
3. Write a program using your own function to find sum of two numbers. (5) [TU 2078]
4. Write a program to find product of two integers using your own function. (5) [TU 2077]
5. What is function? Discuss the benefits of using function. (5) [TU 2074]