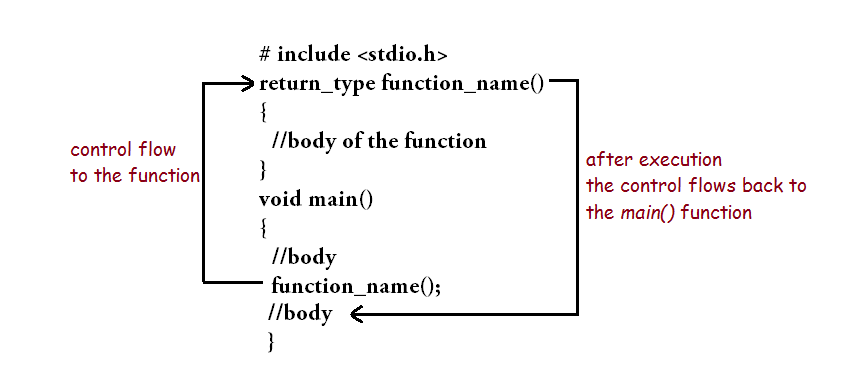
**Module – 4**

**User Defined Functions and Recursion**

**4.1 Introduction to functions**

A function is a block of code that performs a specific task. ‘C’ language allows you to define functions according to your need. These functions are known as user-defined functions. Fig 4.1 shows the control flow during the execution of function.

**For example:**



**Fig 4.1:** Control Flow during the execution of function

**4.2 User defined functions**

**Example: User-defined function**

Here is an example to add two integers. To perform this task, an user-defined function addNumbers() is defined.

**#include <stdio.h>**

**int addNumbers(int a, int b); // function prototype**

**int main()**

**{**

**int n1,n2,sum;**

**printf("Enters two numbers: ");**

**scanf("%d %d",&n1,&n2);**

**sum = addNumbers(n1, n2); // function call**

**printf("sum = %d",sum);**

**return 0;**

**}**

**int addNumbers(int a,int b) // function definition**

**{**

**int result;**

**result = a+b;**

**return result; // return statement**

**}**

* **Function Prototype**

**Function prototype**. ... In computer programming, a **function prototype** or **function** interface is a declaration of a **function** that specifies the **function's** name and type signature, but omits the **function** body.

A function prototype gives information to the compiler that the function may later be used in the program.

### Syntax of function prototype

**returnType functionName(type1 argument1, type2 argument2,...);**

In the above example, int addNumbers(int a, int b); is the function prototype which provides following information to the compiler:

1. name of the function is addNumbers()
2. return type of the function is int
3. two arguments of type int are passed to the function

The function prototype is not needed if the user-defined function is defined before the main() function.

* **Calling a function**

Control of the program is transferred to the user-defined function by calling it.

### Syntax of function call

**functionName(argument1, argument2, ...);**

In the above example, function call is made using addNumbers(n1,n2); statement inside the main().

* **Function definition**

Function definition contains the block of code to perform a specific task i.e. in this case, adding two numbers and returning it.

#### Syntax of function definition

returnType functionName(type1 argument1, type2 argument2, ...)

{

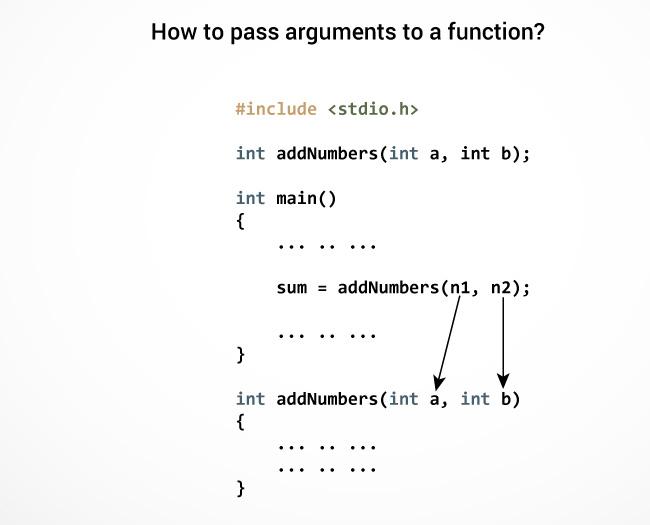
//body of the function

}

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

## Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables ***n1*** and ***n2*** are passed during function call. The parameters ***a*** and ***b*** accepts the passed arguments in the function definition. These arguments are called formal parameters of the function. Fig 4.2 shows the illustration of passing arguments to a function.



**Fig 4.2:** Passing Arguments to Functions

* **Is Main a user defined function?**

**main**() **function** is a **user defined**, body of the **function** is **defined** by the programmer or we can say **main**() is programmer/**user** implemented **function**, whose prototype is predefined in the compiler. Hence we can say that **main**() in c programming is **user defined** as well as predefined because it's prototype is predefined.

* **Why do we use functions?**

This example highlights the two most important reasons that C programmers **use functions**. The first reason is reusability. Once a **function** is defined, it can be used over and over and over again. ... Another aspect of reusability is that a single**function** can be used in several different (and separate) programs.

## Types of function

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming

There are two types of function in C programming:

* Standard library functions
* User defined functions

### Standard library functions

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc. These functions are defined in the header file. When you include the header file, these functions are available for use.

**For example:**

The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "stdio.h" header file. There are other numerous library functions defined under "stdio.h", such as scanf(), fprintf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use

### User-defined function

As mentioned earlier, C allow programmers to define functions. Such functions created by the user are called user-defined functions. You can create as many user-defined functions as you want.

### Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

## Types of Function calls in C

Functions are called by their names; we all know that, then what is this tutorial for? Well if the function does not have any arguments, then to call a function you can directly use its name. But for functions with arguments, we can call a function in two different ways, based on how we specify the arguments, and these two ways are:

1. Call by Value
2. Call by Reference

### Call by Value

Calling a function by value means, we pass the values of the arguments which are stored or copied into the formal parameters of the function. Hence, the original values are unchanged only the parameters inside the function changes.

#**include<stdio.h>**

**void calc(int x); // Function Prototype**

**int main()**

**{**

**int x = 10;**

**calc(x);**

**// this will print the value of 'x'**

**printf("\nvalue of x in main is %d", x);**

**return 0;**

**}**

**void calc(int x)**

**{**

**// changing the value of 'x'**

**x = x + 10 ;**

**printf("value of x in calc function is %d ", x);**

**}**

**Output**

**Value of x in calc function is 20**

**Value of x in main is 10**

In this case, the actual variable **x** is not changed. This is because we are passing the argument by value, hence a copy of **x** is passed to the function, which is updated during function execution, and that copied value in the function is destroyed when the function ends(goes out of scope). So the variable **x** inside the **main()** function is never changed and hence, still holds a value of **10.**

But we can change this program to let the function modify the original **x** variable, by making the function **calc()** return a value, and storing that value in x.

**#include<stdio.h>**

**int calc(int x);**

**int main()**

**{**

**int x = 10;**

**x = calc(x);**

**printf("value of x is %d", x);**

**return 0;**

**}**

**int calc(int x)**

**{**

**x = x + 10 ;**

**return x;**

**}**

**Output:**

**Value of x is 20**

### Call by Reference

In call by reference we pass the address (reference) of a variable as argument to any function. When we pass the address of any variable as argument, then the function will have access to our variable, as it now knows where it is stored and hence can easily update its value.

In this case the formal parameter can be taken as a **reference** or a **pointer** (don't worry about pointers, we will soon learn about them), in both the cases they will change the values of the original variable.

**#include<stdio.h>**

**void calc(int \*p); // function taking pointer as argument**

**int main()**

**{**

**int x = 10;**

**calc(&x); // passing address of 'x' as argument**

**printf("value of x is %d", x);**

**return(0);**

**}**

**void calc(int \*p) //receiving the address in a reference pointer variable**

**{**

**/\* Changing the value directly that is stored at the address passed \*/**

**\*p = \*p + 10;**

**}**

**Output:**

**Value of x is 20**

## Passing Arrays to a Function

Whenever we need to pass a list of elements as argument to any function in C language, it is prefered to do so using an array. But how can we pass an array as argument to a function? Let's see how it is done.

### Declaring Function with array as a parameter

There are two possible ways to do so, one by using call by value and other by using call by reference.

1. We can either have an array as a parameter.

**int sum (int arr[]);**

1. Or, we can have a pointer in the parameter list, to hold the base address of our array.

**int sum (int\* ptr);**

We will study the second way in details later when we will study pointers.

### Passing arrays as parameter to function

Now let's see a few examples where we will pass a single array element as argument to a function, a one dimensional array to a function and a multidimensional array to a function.

#### Passing a single array element to a function

Let's write a very simple program, where we will declare and define an array of integers in our main()function and pass one of the array element to a function, which will just print the value of the element.

**#include<stdio.h>**

**void giveMeArray(int a);**

**int main()**

**{**

**int myArray[] = { 2, 3, 4 };**

**giveMeArray(myArray[2]); //Passing array element myArray[2] only.**

**return 0;**

**}**

**void giveMeArray(int a)**

**{**

**printf("%d", a);**

**}**

**Output**

4

#### Passing a complete One-dimensional array to a function

To understand how this is done, let's write a function to find out average of all the elements of the array and print it. We will only send in the name of the array as argument, which is nothing but the address of the starting element of the array, or we can say the starting memory address.

**#include<stdio.h>**

**float findAverage(int marks[]);**

**int main()**

**{**

**float avg;**

**int marks[] = {99, 90, 96, 93, 95};**

**avg = findAverage(marks); // name of the array is passed as argument.**

**printf("Average marks = %.1f", avg);**

**return 0;**

**}**

**float findAverage(int marks[])**

**{**

**int i, sum = 0;**

**float avg;**

**for (i = 0; i <= 4; i++) {**

**sum += marks[i];**

**}**

**avg = (sum / 5);**

**return avg;**

**}**

**Output:**

94.6

#### Passing a Multi-dimensional array to a function

Here again, we will only pass the name of the array as argument.

**#include<stdio.h>**

**void displayArray(int arr[3][3]);**

**int main()**

**{**

**int arr[3][3], i, j;**

**printf("Please enter 9 numbers for the array: \n");**

**for (i = 0; i < 3; ++i)**

**{**

**for (j = 0; j < 3; ++j)**

**{**

**scanf("%d", &arr[i][j]);**

**}**

**}**

**// passing the array as argument**

**displayArray(arr);**

**return 0;**

**}**

**void displayArray(int arr[3][3])**

**{**

**int i, j;**

**printf("The complete array is: \n");**

**for (i = 0; i < 3; ++i)**

**{**

**// getting cursor to new line**

**printf("\n");**

**for (j = 0; j < 3; ++j)**

**{**

**// \t is used to provide tab space**

**printf("%d\t", arr[i][j]);**

**}**

**}**

**}**

**Output:**

Please enter 9 numbers for the array:

1

2

3

4

5

6

7

8

9

The complete array is:

1 2 3

4 5 6

7 8 9

## Passing Strings to Function

Strings can be passed to a function in a similar way as arrays.

### Example: Passing string to a Function

**#include <stdio.h>**

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

int **main()**

**{**

char **str[50];**

**printf("Enter string: ");**

**gets(str);**

**displayString(str); // Passing string to a function.**

return **0;**

**}**

void **displayString(**char **str[])**

**{**

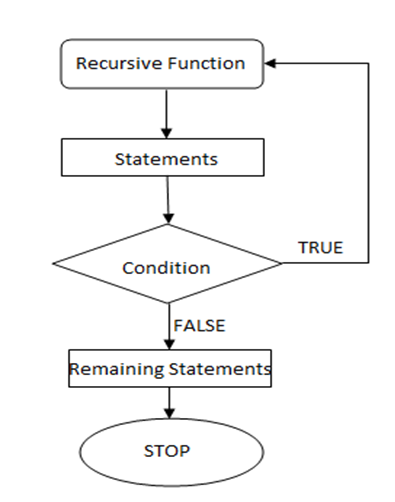
**printf("String Output: ");**

**puts(str);**

**}**

**4.3 Recursion in ‘C’ Language**

**Recursion** is a programming technique that allows the programmer to express operations in terms of themselves. In **C**, this takes the form of a function that calls itself. A useful way to think of **recursive** functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". Flow chart of recursion is shown in Fig 4.3.



**Fig 4.3: Flowchart Showing Recursion**

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be **careful to define an exit condition from the function, otherwise it will go into an infinite loop.**

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

**How recursion works?**

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

**void recurse()**

**{**

**... .. ...**

**recurse();**

**... .. ...**

**}**

**int main()**

**{**

**... .. ...**

**recurse();**

**... .. ...**

**}**

The recursion continues until some condition is met to prevent it.To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

Base condition in recursion:

In recursive program, the solution to base case is provided and solution of bigger problem is expressed in terms of smaller problems.

**int fact(int n)**

**{**

**if (n < = 1) // base case**

**return 1;**

**else**

**return n\*fact(n-1);**

**}**

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

**Different ways of defining recursion**

Some of the ways in which recursive functions are characterized. The characterizations are based on:

1. whether the function calls itself or not (direct or indirect recursion).

2. whether there are pending operations at each recursive call (tail-recursive or not).

3. the shape of the calling pattern -- whether pending operations are also recursive (linear or tree-recursive).

* **Direct Recursion:**

A C function is directly recursive if it contains an explicit call to itself. For example, the function

**int foo(int x)**

**{**

**if (x <= 0)**

**return x;**

**return foo(x - 1);**

**}**

includes a call to itself, so it's directly recursive. The recursive call will occur for positive values of x.

* **Indirect Recursion:**

A C function foo is indirectly recursive if it contains a call to another function which ultimately calls foo.

The following pair of functions is indirectly recursive. Since they call each other, they are also known as mutually recursive functions.

**int foo(int x)**

**{**

**if (x <= 0)**

**return x;**

**return bar(x);**

**}**

**int bar(int y) {**

**return foo(y - 1);**

**}**

* **Tail Recursion:**

A recursive function is said to be tail recursive if there are no pending operations to be performed on return from a recursive call.

Tail recursive functions are often said to "return the value of the last recursive call as the value of the function." Tail recursion is very desirable because the amount of information which must be stored during the computation is independent of the number of recursive calls. Some modern computing systems will actually compute tail-recursive functions using an iterative process.

The "infamous" factorial function fact is usually written in a non-tail-recursive manner:

**int fact (int n)**

**{ /\* n >= 0 \*/**

**if (n == 0)**

**return 1;**

**return n \* fact(n - 1);**

**}**

Notice that there is a "pending operation," namely multiplication, to be performed on return from each recursive call. Whenever there is a pending operation, the function is non-tail-recursive. Information about each pending operation must be stored, so the amount of information is not independent of the number of calls.

The factorial function can be written in a tail-recursive way:

**int fact\_aux(int n, int result)**

**{**

**if (n == 1)**

**return result;**

**return fact\_aux(n - 1, n \* result)**

**}**

**int fact(n)**

**{**

**return fact\_aux(n, 1);**

**}**

The "auxiliary" function fact\_aux is used to keep the syntax of fact(n) the same as before. The recursive function is really fact\_aux, not fact. Note that fact\_aux has no pending operations on return from recursive calls. The value computed by the recursive call is simply returned with no modification. The amount of information which must be stored is constant (the value of n and the value of result), independent of the number of recursive calls.

**Advantages and Disadvantages of Recursion**

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc.

Recursive program has greater space requirements than iterative program as all functions will remain in stack until base case is reached. It also has greater time requirements because of function calls and return overhead.

**4.4 Example Programs: Recursion**

**Sum of Natural Numbers**

**#include <stdio.h>**

**int sum(int n);**

**int main()**

**{**

**int number, result;**

**printf("Enter a positive integer: ");**

**scanf("%d", &number);**

**result = sum(number);**

**printf("sum=%d", result);**

**}**

**int sum(int num)**

**{**

**if (num!=0)**

**return num + sum(num-1); // sum() function calls itself**

**else**

**return num;**

**}**

Initially, the sum() is called from the main() function with number passed as an argument.

Suppose, the value of num is 3 initially. During next function call, 2 is passed to the sum() function. This process continues until num is equal to 0.

When num is equal to 0, the if condition fails and the else part is executed returning the sum of integers to the main() function.

**4.5 Example: Factorial of a positive integer**

The following example calculates the factorial of a given number using a recursive function

**#include <stdio.h>**

**unsigned long long int factorial(unsigned int i)**

**{**

**if(i <= 1) {**

**return 1;**

**}**

**return i \* factorial(i - 1);**

**}**

**int main()**

**{**

**int i = 12;**

**printf("Factorial of %d is %d\n", i, factorial(i));**

**return 0;**

**}**

When the above code is compiled and executed, it produces the following result −

Factorial of 12 is **479001600**

**4.6 Example: Fibonacci Series**

The following example generates the Fibonacci series for a given number using a recursive function

**#include <stdio.h>**

**int fibonacci(int i)**

**{**

**if(i == 0)**

**{**

**return 0;**

**}**

**if(i == 1)**

**{**

**return 1;**

**}**

**return fibonacci(i-1) + fibonacci(i-2);**

**}**

**int main()**

**{**

**int i;**

**for (i = 0; i < 10; i++)**

**{**

**printf("%d\t\n", fibonacci(i));**

**}**

**return 0;**

**}**

**When the above code is compiled and executed, it produces the following result −**

**0**

**1**

**1**

**2**

**3**

**5**

**8**

**13**

**21**

**34**

**Scope visibility and Lifetime of variables**

**Scope** & **Lifetime**: The **scope** of a declaration is the part of the program for which the declaration is in effect. C/C++ use lexical scoping. The **lifetime** of a **variable** or object is the time period in which the **variable**/object has valid memory. **Lifetime** is also called "allocation method" or "storage duration."

* ***Automatic Variables:*** The variables which are declared inside a block are known as **automatic** or **local variables**; these variables allocates memory automatically upon entry to that block and free the occupied memory upon exit from that block.

These variables have local scope to that block only that means these can be accessed in which variable declared.

Keyword 'auto' may be used to declare automatic variable but we can declare these variable without using 'auto' keywords.

**Consider the following declarations**

**int main()**

**{**

**auto int a;**

**int b;**

**....**

**return 0;**

**}**

Here, both variables a and b are automatic variables.

## Automatic variables in other user defined functions

An automatic or local variable can be declared in any user define function in the starting of the block.

**Consider the following code**

**void myFunction(void)**

**{**

**int x;**

**float y;**

**char z;**

**...**

**}**

**int main()**

**{**

**int a,b;**

**myFunction();**

**....**

**return 0;**

**}**

In this code snippet, variables x, y and z are the local / automatic variable of myFunction() function, while variables a and b are the local / automatic variables of main() function.

* ***External Variables:*** In the C programming language, an external variable is a variable defined outside any function block. On the other hand, a local (automatic) variable is a variable defined inside a function block.

For most C implementations, every byte of memory allocated for an external variable is initialized to zero. The scope of external variables is global, i.e. the entire source code in the file following the declarations. All functions following the declaration may access the external variable by using its name.

* ***Static variable:*** Static variable is one that is not seen outside the function in which it is declared but which remains until the program terminates. It also means that the value of the variable persists between successive calls to a function.

**static data\_type var\_name = var\_value;**

**For example**

| **#include<stdio.h>**  **int fun()**  **{**  **static int count = 0; // Static Variable**  **count++;**  **return count;**  **}**    **int main()**  **{**  **printf("%d ", fun());**  **printf("%d ", fun());**  **return 0;**  **}** |
| --- |

Output:

1 2

But the same program when executed using normal auto variables prints the output

| **#include<stdio.h>**  **int fun()**  **{**  **int count = 0; // Auto Variable**  **count++;**  **return count;**  **}**    **int main()**  **{**  **printf("%d ", fun());**  **printf("%d ", fun());**  **return 0;**  **}** |
| --- |

Output:

1 1

* **Register Variables**

Registers are faster than memory to access, so the variables which are most frequently used in a **C** program can be put in registers using **register** keyword. The keyword **register** hints to compiler that a given variable can be put in a **register**. It's compiler's choice to put it in a **register** or not.

**What is a Register Variable?**

1. **Register variables** are stored in the CPU registers. Its default value is a garbage value
2. **Variable** stored in a CPU **register** can always be accessed faster than the one that is stored in memory. ...
3. **Variables** for loop counters can be declared as **register**.

*Example:*

***register******int****x=5;*

**4.8 Example Programs:**

**1. /\* C programming source code to convert either binary to decimal or decimal to binary according to data entered by user. \*/**

#include <stdio.h> #include <math.h>

int binary\_decimal(int n); int decimal\_binary(int n); int main()

{

int n; char c;

printf("Instructions:\n");

printf("1. Enter alphabet 'd' to convert binary to decimal.\n"); printf("2. Enter alphabet 'b' to convert decimal to binary.\n"); scanf("%c",&c);

if (c =='d' || c == 'D')

{

printf("Enter a binary number: "); scanf("%d", &n);

printf("%d in binary = %d in decimal", n, binary\_decimal(n));

}

if (c =='b' || c == 'B')

{

printf("Enter a decimal number: "); scanf("%d", &n);

printf("%d in decimal = %d in binary", n, decimal\_binary(n));

}

return 0;

}

int decimal\_binary(int n) /\* Function to convert decimal to binary.\*/

{

int rem, i=1, binary=0; while (n!=0)

{

rem=n%2;

n/=2; binary+=rem\*i; i\*=10;

}

return binary;

}

int binary\_decimal(int n) /\* Function to convert binary to decimal.\*/

{

int decimal=0, i=0, rem; while (n!=0)

{

rem = n%10; n/=10;

decimal += rem\*pow(2,i);

++i;

}

return decimal;

}

## Output:

## Instructions:

1. Enter alphabet 'd' to convert binary to decimal.
2. Enter alphabet 'b' to convert decimal to binary.

Enter a binary number: 110111

110111 in binary = 55 in decimal

**2. With using user-defined function write a program to find length of string.**

#include<stdio.h>

// Prototype Declaration int FindLength(char str[]);

int main()

{

char str[100];

int length;

printf("\nEnter the String : ");

gets(str);

length = FindLength(str);

printf("\nLength of the String is : %d", length);

return(0);

}

int FindLength(char str[])

{

int len = 0;

while (str[len] != '\0')

len++;

return (len);

}