**UNIT-III**

**CHAPTER – 2**

**Functions**

**Functions in C:**

In C, a program is made up of one or more functions, one and only one of which must be called main. The execution of the program always starts with main, but it can call other functions to do some part of the job.

**Introduction:**

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings .

A function is known with various names like a method or a sub-routine or a procedure, etc.

Several advantages are associated with the use of functions in C or any other language. the major advantages are:

1. Problems can be factored into understandable and manageable steps.

2. Functions provide a way to reuse code that is required in more than one place in a program.

3. Like many languages, C comes with a rich and valuable library.

Ex: math.h

4. We use functions to protect data. Local data consist of data described in a function. These

data are available only to the function and only while the function is executing. When the

function is not running, the data are not accessible. Data in one function, then, cannot be

seen or changed by a function outside of its scope.

**Definition of functions:**

The general form of a function definition in C programming language is as follows:

return\_type function\_name( parameter list )

{

body of the function

}

A function definition in C programming language consists of a function header and a function body. Here are all the parts of a function:

* Return Type: A function may return a value. The return\_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword void.
* Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* Function Body: The function body contains a collection of statements that define what the function does.

Ex: /\* function returning the max between two numbers \*/

int max(int num1,int num2)

{

/\* local variable declaration \*/

int result;

if(num1 > num2)

result= num1;

else

result= num2;

return result;

}

**User defined functions:**

In C, functions must be both declared and defined. The function declaration, which needs to be done before the function call, gives the whole picture of the function that needs to be defined later. The declaration mentions the name of the function, the return type and the type and order of the formal parameters. In other words, the declaration uses only the header of the function definition ended in a semicolon. The function definition, which is traditionally coded after the function that makes the call, contains the code needed to complete the task.

**Elements of user defined functions:**

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. Function definition.

2. Function call.

3. Function declaration.

**Function Declarations:**

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

Return\_type function\_name (parameter list);

Ex: int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

Ex: int max(int ,int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

**Function calls:**

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

**Basic Function Designs:**

For better understanding of arguments and return type in functions, user-defined functions can be categorised as:

1. Void Functions without Parameters ([Function with no arguments and no return value](http://www.programiz.com/c-programming/types-user-defined-functions#no_no))
2. Non-void Functions without Parameters([Function with no arguments and returnvalue](http://www.programiz.com/c-programming/types-user-defined-functions#no_yes))
3. Void Functions with Parameters ([Function with arguments but no return value](http://www.programiz.com/c-programming/types-user-defined-functions#yes_no))
4. Non-Void Functions with Parameters ([Function with arguments and return value](http://www.programiz.com/c-programming/types-user-defined-functions#yes_yes))
5. **Void Functions without Parameters:**

In this category, the function has no arguments. It does not receive any data from the calling function. Similarly, it doesn’t return any value. The calling function doesn’t receive any data from the called function. So, there is no communication between calling and called functions.

Ex: void generateFibo();

int main()

{

generateFibo();

return 0;

}

void generateFibo()

{

int a, b, c, i, terms;

printf("Enter number of terms: ");

scanf("%d", &terms);

a = 0;

b = 1;

c = 0;

printf("Fibonacci terms: \n");

for(i=1; i<=terms; i++)

{

printf("%d, ", c);

a = b;

b = c;

c = a + b;

}

}

1. **Non-void Functions without Parameters:**

In this category, the functions has no arguments and it doesn’t receive any data from the calling function, but it returns a value to the calling function. The calling function receives data from the called function. So, it is one way data communication between calling and called functions.

Ex: #include <math.h>

#include <stdio.h>

int sum();

int main()

{

    int num;

     num = sum();

   printf("\nSum of two given values = %d", num);

   return 0;

}

int sum()

{

     int a = 50, b = 80, sum;

    sum = sqrt(a) + sqrt(b);

    return sum;

}

1. **Void Functions with Parameters:**

In this category, function has some arguments . it receives data from the calling function, but it doesn’t return a value to the calling function. The calling function doesn’t receive any data from the called function. So, it is one way data communication between called and calling functions.

void nat(int);

void main ()

{

int n;

clrscr ();

printf (“Enter n value”);

scanf (“%d”,&n);

nat(n);

getch ();

}

void nat (int n)

{

int i;

for (i=1;i<=n;i++)

printf (“%d\t”,i);

}

|  |
| --- |
|  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |

1. **Non-Void Functions with Parameters:**

In this category, functions has some arguments and it receives data from the calling function. Similarly, it returns a value to the calling function. The calling function receives data from the called function. So, it is two-way data communication between calling and called functions.

int fact(int);

void main ()

{

int n;

printf (“Enter n”);

scanf (“%d”,&n);

printf (“Factorial of the number : %d”,fact(n));

}

int fact (int n)

{

int i,f;

for (i=1,f=1;i<=n;i++)

f=f\*i;

return (f);

}

|  |  |
| --- | --- |
|  |  |
|  |  | |
|  |  | |

|  |  |  |
| --- | --- | --- |
|  |  | |
|  |  |

**Parameter passing techniques:**

There are various types of parameter passing techniques available in programming languages . These are based on how actually parameter is passed to called function (Function which is called) from calling function (Function which calls the other function also called Caller function).

Following are types of parameter passing techniques which are widely used in programming language

1. Call by Value
2. Call by Reference

1. Call by value Parameter passing technique:

When we call the function, we pass parameter to called function and it is by default passed by the value of variable in calling function. All such type of parameter passing technique is called as "call by value".

Ex: int calsum (int  x, int y , int z ) ;

main( )   
 {   
 int  a, b, c, sum ;   
 printf ( "\nEnter any three numbers " ) ;   
 scanf ( "%d %d %d", &a, &b, &c ) ;   
 sum = calsum ( a, b, c ) ;

printf ( "\nSum = %d", sum ) ;   
 }

calsum (int x, int y, int z )   
 {   
 int d ;   
 d = x + y + z ;   
 return ( d ) ;   
 }

Output: Enter any three numbers 10 20 30

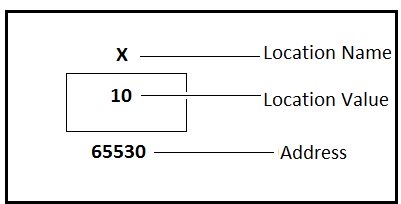
Sum = 60

2. Call by value Reference passing technique:

This type of parameter passing technique is also called as call by address because we are passing the address of actual variable as an argument to called function instead of value of actual argument. Each variable has some fixed address which remains constant throughout execution of program in memory. Using this address of variable, it is also possible to access and modify the value of variable by using pointer.

Consider declaration of variable.

int x= 10; which tells the compiler   
1) To allocate 2 byte (depending on type of compiler and type of variable ) memory location

on stack frame of that particular function in which it is declared.  
2) To hold value of type integer and also associate this address with name x and   
3) This memory location is initialized with value 10.  
  
 

Memory Map

Selection of address for particular variable is compiler dependent and is always whole number. We can access value 10 by using variable name x in this case and by address 65530 also.  
‘&’ is a special operator in c used as ‘address of’.  
Eg : The expression &x in above case return address of variable x which happens to be 65530.

As address is always positive, we use operator %u  for unsigned integer.  
‘\*’ is special operator in c used as ‘value at address’ also called indirection operator.  
Eg : \*(&x) which means value at address(&x) i.e. value at address (65530) which happens to be value of variable x=30.  
  
Ex: #include<stdio.h>

Void main( )   
 {   
 int  x = 10;   
 printf ( "\nAddress of x = %u", &x ) ;   
 printf ( "\nValue of x = %d", x ) ;   
 printf ( "\nValue of x = %d", \*( &x ) ) ;   
 }   
  
Output: Address of x = 65530  
 Value of x = 10  
 Value of x = 10

In call by reference we pass the address of actual parameter in calling function and is copied in formal argument in called function .As we are passing address of actual parameter, changes made to formal argument would reflect in calling function.  
Ex: #include<stdio.h>

Void exchange(a,b);

void main( )   
 {   
 int  a = 10, b = 20 ;  
 printf ( "\na = %d b = %d", a, b ) ;   
 exchange( &a, &b ) ;   
 printf ( "\na = %d b = %d", a, b ) ;   
 }

void exchange( int \*x, int \*y )   
 {   
 int  t ;   
 t = \*x ;   
 \*x = \*y ;   
 \*y = t ;   
 }   
  
Output : a = 10 b = 20   
 a = 20 b = 10   
  
Consider declaration of variable int \*x= 10;

which means x is declared as pointer variables, i.e. variable is capable of holding address which is always positive whole number. int \* x does not mean that x is going to hold integer value rather it simply means it is going to hold address  (memory location ) of  integer type of variable. We are passing address of variables a, b to function exchange()  so while receiving we are using two pointer to integer.

Generally we use call by value mechanism if we don’t need the changes made in calling function to reflect in called function. But if we required the changes to reflect in called function we use call by reference mechanism.

Also if we want to return more than one value to called function from calling function we go for call by reference value which is not possible ordinarily.

**Recursion :**

A function that calls itself is known as recursive function and this technique is known as recursion in C programming.

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 in infinite loop.

Recursive function is very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

Ex: #include <stdio.h>

int sum(int n);

int main()

{

int num,add;

printf("Enter a positive integer:\n");

scanf("%d",&num);

if (num<0)

{

printf(“entered negative numer”);

exit(0);

}

else

{

add=sum(num);

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

}

}

int sum(int n)4 ,3

{

if (n==0)

return n;

else

return n+sum(n-1); /\*self call to function sum() \*/

}

Example: Generating Fibonacci Series

#include<stdio.h>

int Fibonacci(int);

int main()

{

int n, i = 0, c;

scanf("%d",&n); 5

printf("Fibonacci series\n");

for ( c = 0 ; c < n ; c++ )

{

printf("%d\n", Fibonacci(i));

i++;

}

return 0;

}

int Fibonacci(int n)

{

if ( n == 0 )

return 0;

else if ( n == 1 )

return 1;

else

return ( Fibonacci(n-1) + Fibonacci(n-2) );

}

0 11 2 3

Fib(2)=fib(1)+fib(0)=1+0=1

Fib(3)=fib(2)+fib(1)=1+1=2

Fib(4)=fib(3)+fib(2)=2+1=3