Unit 5
Function

Introduction: Function

A function is a self-contained block of statements that perform a specific and well defined task of some kind. A C program can be made up of many small functions, each performing a particular task, rather than one large main() function. The process of breaking a large program into small and manageable, well defined functions and designed them independently is called modular programming.

Advantage of Function:

- There is no chance of code duplication.
- Each function is re-usable.
- It can be developed, tested, debugged, and compiled independently by different member of a programming team.
- Large programmer can be involved in program development.
- Program can be developed in short period of time
- It can be called any number of times in any place with different parameters.

Component of Function:

Let us consider following program.

```
#include<stdio.h>
int sum(int, int);
void main()
{
int a,b,result;
printf("\n Input any two numbers:-");
scanf("%d%d",&a,&b); result =
sum(a,b); printf("\n Sum = %d",
result);
}

int sum(int x, int y)
{
int z;
 z = x + y;
return (z);
}
```

a) Function Declaration

- It specifies function name, argument types and return value.
- It alerts compiler, that function is coming up later somewhere in the program.
- It is also known as function prototype.

Syntax: `return type function_name(argument1, argument2,….,argumentn);`

Example: `int sum_calcutaion(int, int);`

b) Function definition (Called Function)

- The function itself contains the lines of code that constitute the function.
- Syntax:
```
return type function_name(aruments)
{
```

```
        Statement1;
        Statement2;
        ……..
        ……..
}
```

Example:

i)
```
void xyz()
 {
 int i;
for(i=0;i<10;i++)
     printf("%d",i);
}
```

ii)
```
int sum_calculation(int x, int y)
```

c) Function call (Calling function)

- It is specified by function name followed by the arguments enclosed in parentheses and terminated by a semicolon.
- The return type is not mentioned in the function call.
- result = addition(a, b);

d) Function parameters

- There are two types of parameters in function and they are: formal parameter and actual parameter.
- The parameter specified in function call is known as actual parameters and those parameters specified in the function declaration (definition) are known as formal parameters.
- In above functions:
  `a, b` are actual parameter, where as `x` and `y` are formal parameters.

- Relation between actual and formal parameters:
  - ✓ Number of formal and actual parameters should be same in calling and called function ✓ Order of both parameters should be same.
  - ✓ Data type of both parameters should be identical.

e) Return type

- The return function is used for two purposes:
  - ✓ To terminate the function and transfer the program control back to the calling function.
  - ✓ To terminate the function and return a value to the calling function.
- The return statement tells the C compiler that the execution of function is over and control should go to the calling program. Function can be grouped into two types:
  i. Function that do not have return type i.e. void function.
  ii. Functions that do have a return type i.e. return (z).

Types of Function:

According to the return type, and passing parameters in a function, there are four types of functions and they are:

a) Function with passing arguments and returning values

In this function, the calling function passes the number of arguments to the called function and the called function returns the calculated value to the calling function.

The general form of this type of functions would be:

Syntax: return_type function_name(argument1, argument2,..... argumentN);

It can be illustrated in following example

```
#include<stdio.h>
float simple_interest(int,float,float);
 void main()
{
int p;
float t,r,result;
printf("\n Input Principal Amount:- ");
scanf("%d",&p); printf("\nInput Time:");
scanf("%f",&t);
printf("\n Input Rate of interest:- ");
scanf("%f",&r);
result=simple_interest(p,t,r);
printf("\n Simple Interest = %10.2f",result);
}

float simple_interest(int x,float y,float z)
{
float interest;
interest= (x*y*z)/100;
return (interest);
}
```

b) Function with passing arguments and without returning values

```
#include<stdio.h>

void simple_interest(int,float,float);
 void main()
{
int p;
float t,r;
printf("\n Input Principal Amount:- ");
scanf("%d",&p); printf("\n Input Time:");
scanf("%f",&t);
printf("\n Input Rate of interest:- ");
scanf("%f",&r); simple_interest(p,t,r);
}
 void simple_interest(int x,float y,float z)
{
 float interest;
interest= (x*y*z)/100;
printf("\n Simple Interest = %10.2f",interest);
}
```

c) Function without passing arguments but returning values.

```
#include<stdio.h>
float simple_interest(void);
 void main()
{
float result;
result = simple_interest();
```

```
printf("\n Simple interest = %10.2f",result);
}

float simple_interest(void)
{
int p;
float t,r,interest;
printf("\n Input Principal Amount:- ");
scanf("%d",&p); printf("\n Input Time:-");
scanf("%f",&t);
printf("\n Input Rate of interest:- ");
scanf("%f",&r); interest= (p*t*r)/100;
return(interest);
}
```

d) Function without returning value and without passing arguments.

```
/*Function without passing arguments and without returning values*/
#include<stdio.h>
#include<conio.h>

void simple_interest(void);
 void main() {
simple_interest();
}
 Void simple_interest(void)
{
int p;
float t,r,interest;
printf("\n Input Principal Amount:- ");
scanf("%d",&p); printf("\n Input Time:");
 scanf("%f",&t);
printf("\n Input Rate of interest:- ");
scanf("%f",&r); interest= (p*t*r)/100;
printf("\n Simple interest = %10.2f",interest);
}
```

Passing Values: by values and reference

A function defined outside the *main ( )* function should be called inside the body of *main ( )* in order to execute. There are two ways to call a function and they are:

- Call by value
- Call by address / reference

a) Call by value

If a function is called by passing the value of a variable as actual argument, then the function is known as function call by value.

Limitation of call by value:

- The change made in formal argument does not make any effect on the actual argument.

```
#include<stdio.h>
void increment(int);
void main()
{
int x = 100;
printf("\n The value of x before increment is = %d", x);
increment(x);
```

```
printf("\n The value of x after increment is = %d",x);

}

void increment(int x)
{

x = x+1;
}
```

Output:

```
The value of x before increment is = 100
The value of x before increment is = 100
```

In above program, the function *increment ( )* receives the value 100 passed by the *main ( )* in a formal argument *x.* although the name of the actual and formal argument is same but they reserves separate memory block. Therefore the increment statement *(x = x+1)* is implemented only in the block of formal argument *x* and not on the block of actual argument *x.* therefore the second *printf ( )* statement in *main ( )* print the value stored in the block of the actual argument *x* which is *100* in the program

- Only one value can be returned by the called function at a time.

```
#include<stdio.h>
void swap(int, int);
void main()
{
int a, b;
printf("\n Input First Number:- ");
scanf("%d", &a);
printf("\n Input Second Number:- ");
scanf("%d", &b);
printf("\n The value of A = %d and B = %d before swapping", a, b);
swap(a, b);
printf("\n The value of A = %d and B = %d after swapping", a, b);
}
void swap(int a, int b)
{

int temp;
temp =
a; a =
b; b =
temp;

}
```

Output:

```
Input First Number:- 100
Input Second Number:- 200
The value of A = 100 and B = 200 before swapping
The value of A = 100 and B = 200 after swapping
```

In above program, the expected result is *"The value of A = 200 and B = 100 before swapping"* but we got *"The value of A = 100 and B = 200 after swapping"*. This is due the function is called by value and any change made in

formal *a* and *b* does not effect on actual parameter *a* and *b* the function *swap ( )* cannot return two values. **b) Call by address / reference**

If a function is called by passing the address of a variable as actual argument, then the function is know as function call by address / reference. The address of a variable is used as actual argument and the pointer is used as formal argument. Once the address of a variable is collected in the formal pointer variable, the desired operation can be performed on the variable of calling function.

```
#include<stdio.h>
void increment(int);
void main()
{
int x = 100;
printf("\n The value of x before increment is = %d", x); increment(&x);
printf("\n The value of x after increment is = %d", x);
}

void increment(int *num)
{
*num = *num+1;

}
```

Output:

```
The value of x before increment is = 100
The value of x before increment is = 101
```

In above program, the content of *num* is incremented through the pointer variable *num*. Any change made in formal argument *(pointer num)* cause the same change in actual argument *(x)*. The value of *x* gets incremented by the function increment through the pointer *num.*

```
#include<stdio.h>
void swap(int *, int *);
void main()
{
int a, b;
printf("\n Input First Number:- ");
scanf("%d", &a);
printf("\n Input Second Number:- ");
scanf("%d", &b);
printf("\n The value of A = %d and B = %d before swapping", a, b);
 swap(&a, &b);
printf("\n The value of A = %d and B = %d after swapping", a,
b);

}

void swap(int *x, int *y)
{
int temp;
temp =*x;
*x = *y;
*y = temp;

 }
```

Output:

```
Input First Number:- 100
Input Second Number:- 200
The value of A = 100 and B = 200 before swapping
The value of A = 200 and B = 100 after swapping
```

In above program, before the function call the value of *a* is *100* and *b* is *200*. Since the address of *a* and *b* is passed to the function which is collected in two pointer variables *x* and *y.* The operation done in *x* and *y* causes the effect in *x* and *y.* Therefore after function call the value of *a* and *b* are interchanged.

Passing both values and address:

It is not always true that a function is called by passing only the value or by passing only the address but in most cases we need to pass both values and address at once. This process is called mixed function call. It can be demonstrated in following program.

```
#include<stdio.h>
void calculate(int, int, int *, int *);
void main()
{
int a, b, a1, b1;
printf("\n Input First Number:- ");
 scanf("%d", &a);
printf("\n Input Second Number:- ");
scanf("%d", &b);
printf("\n The value of A = %d and B = %d before swapping", a, b);
calculate(a, b,&a1, &b1);
printf("\n The summation = %d and subtraction = %d", a1, b1);
}

void calcuate(int x, int y, int *x1, int *y1)
{
*x1 = x + y;
*y1 = x - y;

 }
```

Output:

```
Input First Number:- 400
Input Second Number:- 200
The summation = 600 and subtraction = 200
```

In above program, four arguments are passed to the function. They are *a (Value of a)*, *y (value of b)*, *&a1 (address of a1)* and *&b1(Address of b1).* The address of *a1* and *b1* is collected in two pointers *x1* and *x2* respectively. Through the pointer variable result of *x + y* is stored in *x1* and *x – y* is stored in *x2.* Finally the value of *x1* and *x2* is displayed from the *main()* function.

Storage Class:

Every variable and function in C has two attributes: type and storage class

The period of time during which memory is associate with a variable is characterized by storage class. There are two kinds of locations in a computer where variables (value) may be kept: memory and CPU register. A variable's storage class tells us:

- Where the variable would be stored
- What will be the initial value of the variable, if the initial value is not assigned i.e. default initial value.
- What is the scope of the variable
- What is the life of the variable i.e. how long would the variable exist.

There are four types of storage class in C

a) automatic storage class
b) register storage class
c) static storage class
d) external storage class

a) Automatic storage class

Features:

- Keyword : auto
- Storage : memory
- Default initial value : Garbage value
- Scope : Local to the block where the variable is defined
- Life : till the control remains within the block, where the variable is defined
- Use :

b) Register storage class Features:

- Keyword : register
- Storage : CPU register
- Default initial value : Garbage value
- Scope : Local to the block where the variable is defined
- Life : till the control remains within the block
- Use : for those variables which are being used very often in a program, such as for loop counter

c) Static storage class Features:

- Keyword : static
- Storage : memory
- Default initial value : zero
- Scope : Local to the block where the variable is defined
- Life : value of the variables continue between different function calls
- Use : if we want the value of a variable to persist between different function calls, such as in recursive functions.

d) External storage class

Features:

- Keyword : extern
- Storage : memory
- Default initial value : zero
- Scope : global
- Life : As long as the programs execution doesn't come to an end.
- Use : for those variables which are being used almost by all the functions in the program. Declaring all the variables as extern would amount a lot of wastage of memory space, because these variables remains active throughout the life of the program. Note: Local variable gets preference over global variable.

Pre-processor directives:

Preprocessor is a program that processes our source program before it is passed to the compiler. Preprocessor commands are often called directives. The transformations can be inclusion of header file, macro expansions etc. All preprocessing directives begin with a # symbol.

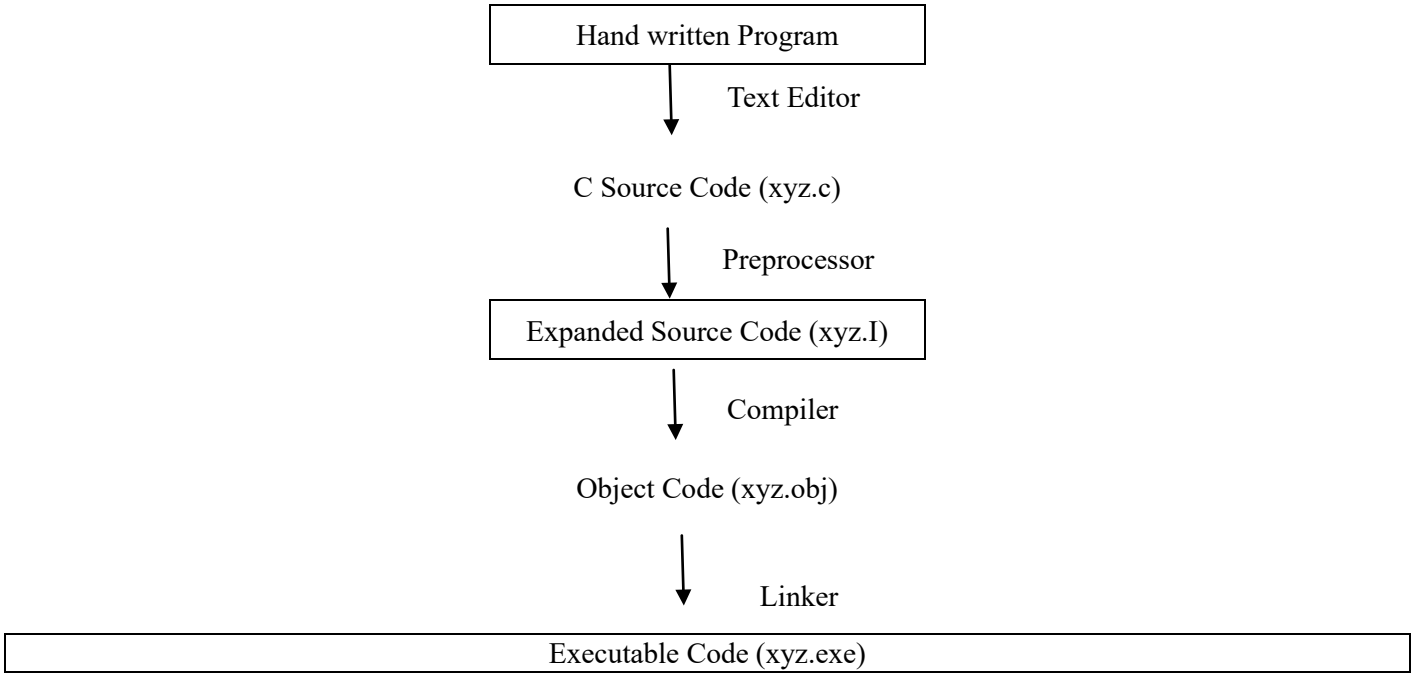

Fig: C Preprocessor

There are four types of pre-processor directives and they are:

a) Macro expansion
b) File inclusion
c) Miscellaneous directives
d) Conditional compilation

Macro expansion

Macros in C are string replacements commonly used to define constants. When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the **#define** directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.

Syntax:

```
#define macro_template macro_expanson
```

Example:

```
#define PI 3.13333
```

The working of macro can be illustrated in following program

```
#include<stdio.h>
#define PI 3.14

void main()
{
float r,area;
printf("\n Input Radius:- ");
scanf("%f", &r);
area=PI*r*r;
printf("\n Area = %10.2f",area);
}
```

Predefined Macros:

There are certain macros that have been defined in the C compiler and we can't modify it. Some of them are listed below:

Macro	Description
__DATE__	It display the current system date (MM:DD:YY) as a string
__TIME__	It display the current system time (HH: MM: SS) as a string
__FILE__	It is used to display the name of current file as a string
__LINE__	It is used to display the total number of lines within a program

Working of predefine macros is illustrated in following program:

```
#include<stdio.h>

void main()
{
printf("\n Name of file is = %s", __FILE__ );
printf("\n Total Number of lines compiled =
%d",__LINE__); printf("\n Current System Date =
%s",__DATE__); printf("\n Current System Time =
%s",__TIME__); }
```

Output:

Name of file = Macro_te.c
Total Number of lines compiled = 7
Current System Date =
Current System Time =

C libraries:

The C Standard Library is a set of C built-in functions, constants and header files like <stdio.h>, <stdlib.h>, <math.h>, etc. The prototype and data definitions of the functions are present in their respective header files, and must be included in our program to access them.

There are many library functions available in C programming which helps us to write good and efficient programs.

Advantage:

- They are simple, easy to use and are workable.
- They are able to create most efficient code optimized for maximum performance.
- It save considerable amount of development time and effort.
- They are portable

Header Files and Prototyping:

A header file is a file which contains C function declarations and macro definitions to be shared between several source files. Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone. There are two types of header files: User defined header files

- In built header files

Some in built header files are as follows:
<stdio.h> <conio.h> <string.h> <math.h> <stdlib.h> <iostream.h>, etc

Recursion:

Literal meaning of recursion is again and again. In programming, recursion is a process of calling a function by itself until certain condition has been satisfied. The function which calls itself is known as recursive function. It is also known as circular definition. Generally recursion is used to solve various mathematical problems by dividing it into smaller problems.
There are two types of recursion:

- Direct recursion
- Indirect recursion

1. Direct recursion

A function is said to be direct recursive if it calls itself directly. It can be demonstrated in following program.

```
#include<stdio.h>
long int factorial(int);
void main()
{
 int num;
printf("\n Input a number:- ");
scanf("%d", &num);
if(num<0)
printf("\n Invalid Number");
else if(num==0 || num==1)
 printf("\n Factorial of %d is 1",num);
else
 printf("\n Factorial of %d is %ld", num, factorial(num));

}

long int factorial(int num)
{
if(num==1)
return 1;
else
 return (num*factorial(num-1));
}
```

---

2. Indirect recursion

A function is said to be indirect recursive if it calls another function and the new function calls the first calling function again. It can be illustrated in following program.

```
int function1( int x)
{
if (x <=1)
return 1;
else
 return function2(x)
}

int function2(int x)
{
return function1(n);
}
```

Comparison between iteration and recursion:

Iteration	Recursion
It is used to repeat the number of statements for finite or infinite times	It is used to repeat the number of statements for infinite times
It is used for simple logics	It is used for complex data structure
It requires less memory space for execution	It requires large memory space for execution
The execution speed is higher than recursive operation	The execution speed is lower than iterative operation
It uses repetition statements such as while, do – while, for.	It uses selection statements such as if, if – else, switch – case.

Programs Related to Recursion

1. Write a program to read a number in decimal and convert it into its equivalent binary, using recursion.

```
#include<stdio.h>
int binary_conv(int);

void main()
{
int num;
printf("\n Input a decimal number:- ");
scanf("%d", &num);
printf("\n Decimal number = %d and Binary Digit = %d", num,
binary_conv(num));
}

int binary_conv(int num)
{
if(num==0)
return 0;
```

```
else
 return (num%2 + 10 * binary_conv(num/2)); }
```

2. Write a program to display Fibonacci series up to nth term using recursive function.

```
#include<stdio.h>
int fibonacci(int);
void   main()
{
int term,i;
printf("\n How many terms:- ");
scanf("%d", &term);
printf("\n Given Fibonacci Series :- ");
for(i=0;i<term;i++)
printf("%5d",fibonacci(i));
}

int fibonacci(int num)
{
if(num ==0)
return 0;
else if (num ==1)
return 1;
else
 return (fibonacci(num-1) + fibonacci(num-2));
}
```

Alternate Method:

```
#include<stdio.h>
#include<conio.h>

int fibonacci(int, int, int);
 void main() { int
first=0,second=1,term;
clrscr();
printf("\n How many terms:- ");
scanf("%d", &term); fibonacci(first,
second, term); getch();
}
int fibonacci(int first, int second, int term)
{
 if(term!=0)
 {
 printf("%5d",first);
fibonacci(second,second+first,term-1);
 }
}
```

3. **Write a program to calculate the power using recursion.**

```
#include<stdio.h>
#include<conio.h>
long int power(int, int);
void main()
{
int base,exp;
```

```
clrscr();
printf("\n Input value of base:- ");
scanf("%d", &base);
printf("\n Input value of exponent:- ");
scanf("%d", &exp);
printf("\n The exponential value = %ld", power(base, exp));
getch();
}

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

4. **Write a program to read a number and display its reverse value using recursion.**

```
#include<stdio.h>
#include<conio.h>

long int reverse(long int);

void main()
 {
 long int num;
clrscr();
printf("\n Input a number:- ");
scanf("%ld", &num);
printf("Reverse of %ld is %ld", num, reverse(num));
getch();
}

long int reverse(long int num)
{
static long int rem=0;
if(num ==0)
return 0;
rem=rem*10+num%10;
reverse(num/10);
return (rem);
}
```

5. **Write a program to read a number and display the sum of individual digits using recursion.**

```
#include<stdio.h>
#include<conio.h>

int sum_digit(long int);
void main()
{ long int num;
```

```
clrscr();
printf("\n Input a number:- ");
scanf("%ld", &num);
printf("\n Sum of individual digits of %ld is %d", num, sum_digit(num));
getch(); }
int sum_digit(long int num)
{
 if(num!=0)
 return (num%10 + sum_digit(num/10));
else
 return 0;
}
```