UNIT-4

FUNCTIONS

A C-language program is nothing but collection of Function; these are the building blocks of a „C" program. Generally, a function mans a task.

"Function is a collection of logically related instructions which performs a particular task."

Functions are also used for modular programming in which a big task is divided into small modules and all the modules are inter connected.

How to create the function (Function declaration):-

<return value type><function name> (parameter list)
{
Body of the function;
[return<value>];
}
<return value type>: It is specify the data type of the returning value.
<function name>:It is specify the name of the function.
(Parameter list): It is the list of parameters used in the function.

Function definition:-

The collection of program statements in C that describes the specific task done by the function is called a Function definition. It consists of the function header and a function body.

<return type><function name>(parameter list)

{
<local variable declaration>;
<Statements>;
<return (value)>;
}

Function header:

Function header is similar to the function declaration but does not require the semicolon at the end. List of variables in the parenthesis are called Formal parameters. It consists of three parts:

1. Return type.
2. Function name.
3. Formal parameters.

Example:

int add(a, b)

Function body:

After the function header the statements in the function body (including variables and statements) called body of the function.

Example:

```
int add (int x, int y)
{
return (x+y);
}
```

Return statement:

Return statement is used return some value given by the executable code within the function body to the point from where the call was made. General form:

```
return expression;
      or
return value;
```

Where expression must evaluate to a value of the type specified in the function header for the return value.

EX: return a+b;

Or

return a;

If the type of return value has been specified as void, there must be no expression appearing in the return statement it must be written simply as

```
return;
```

Function call (Function calling):

All functions within standard library or user-written, are called with in this main() function, the function call statements involves the function, which means the program control passes to that of the function. Once the function completes its task, the program control is passed back to the calling environment.

Syntax:

<function name><(parameter list)>;

or

variable name=<function name><(parameter list)>;

Example:

1.Write a program to find the mean of the two integer numbers.

```
#include<stdio.h>
int mean(int, int);
main()
{
int p, q, m;
clrscr();
printf("enter p,q values");
scanf("%d%d",&p,&q);
```

```
m=mean(p,q);
printf("mean value is %d",m);
getch();
}

int mean(int x, int y)
{
int temp;
temp=(x+y)/2;
return temp;
}
```

2. Write a program that uses a function to check whether a given year is leap or not.

```
#include<stdio.h>
void leap(int);
main()
{
int  year;
clrscr();
printf("\n enter year \n");
scanf("%d",&year);
leap(year);
}
void leap(int yr)
{
if(year%4==0&&year%100!=0||year%400==0)
printf("leap year");
else
printf("not leap year");
}
```

Types of functions:

The functions are also used for modular programming in which a big task is divided into small modules and all the modules are interconnected. Functions are basically classified as:

1. Standard library functions(built-in functions).
2. user defined functions.

Standard library functions:-

Library functions come along with the compiler and they can be used in any program directly. User can"t be modified the library functions.

Example:

sqrt();, printf();, clrscr();, abs();, strlen();, getch();etc……

Example program:

```
#include<math.h>
main()
{
int x;
printf("enter x value \n");
```

```
scanf("%d",&x);
printf("square root value of x is %d", sqrt(x));
getch();
}
```

user defined functions: user defined functions is one which will be defined by the ser in program to group certain statements together. The scope of a user defined function is to the extent of the program only, in which it has been defined. main is also a user defined function.

Syntax:

```
<returntype><function name)(<parameter list>)
{
Function body;
<return value>;
}
```

Example program:

```
#include<stdio.h>
main()
{
int r,rd;
clrscr();
printf("enter radius for circle\n");
scanf("%d",&r);
rd=area(r);
printf("area of the circle is %d",rd);
getch();
}
area(int x)
{
return(3.14*x*x);
}
```

Types of parameters

The nature of data communication between the calling function and the called function with arguments (parameter).

The parameters are two types:

1. Actual parameters or original parameters.
2. formal parameters or duplicate parameters.

```
main()
{
 ……
 Function1(a1, a2, a3,……am);
 ……
}

 Function1(x1, x2, x3,……xn);
{
```

…….
……....
…….
}

Actual parameters(arguments):

Actual parameters means original parameters for the function call. These parameters are declared at the time of function declaration. In the above example a_1,a_2,a_3,…… a_m are the actual parameters. When a function call is made, only a copy of the values of actual arguments is passed into the called function.

Formal parameters (duplicate arguments):

Formal parameters mean duplicate parameters for function call. These parameters are declared at the time of function definition. In the above example x_1, x_2, x_3,……x_n are the formal parameters, the actual and formal arguments should match in the number, type and order, the values of actual arguments are assigned to the formal arguments on a one to one basis, starting with the first argument.

Types of functions based on parameters:

A function depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:

1. Functions with no arguments and no return value.
2. Functions with arguments and no return value.
3. Functions with arguments and return value.
4. Functions with no arguments and return value.

Functions with no arguments and no return value:

Function which does not have any argument, receive no data from the calling function and not return any value to calling function. Example program:

```
#include<stdio.h>
main()
{
int ch;
printf("1. addition");
printf("2. substraction");
printf("3. multiplication");
printf("enter ur choice from (1-3)");
scanf("%d",&ch);
if(ch>3)
error();
getch();
}
error()
{
```

```
printf("ur choice is wrong");
}
```

Functions with arguments and no return value:

This is another type where one way communication is possible between the callings and called function. That is, the called function will receive data from the calling function, but will not transfer any data to it.

Example program:

```
#include<stdio.h>
main()
{
int a,b,c;
printf("enter 3 numbers");
scanf("%d%d%d",&a,&b,&c);
big(a,b,c);
}
big(x,y,z)
int x,y,z;
{
if(x>y&&x>z)
printf("\n %d is biggest",x);
else if(y>z)
printf("\n %d is biggest",y);
else
printf("\n %d is biggest",z);
}
```

Functions with arguments and return value:

In the type, two way data communication takes place. That is, both the called and calling functions receive and transfer data from each other. Example program:

```
#include<stdio.h>
main()
{
float x,y,z,area();
printf("enter base and height");
scanf("%f%f",&x,&y);
c=area(x,y);
printf("the area is %f sq.units",c);
getch();
}
float area(float b, float h)
{
return(5*b*h);
}
```

Functions with no arguments and return value.

Function which does not have any argument, receive no data from the calling function and return a value to calling function.

Example program:

```
#include<stdio.h>
main()
{
int x;
clrscr();
x=square();
printf("square of 3 is %d",x);
getch();
}
square()
{
return(3*3);
}
```

Nested functions:-

One important advantage of c functions is that they can be called from and within another function. All the called and calling functions transfer and receive data with ach other and this is called "Nested function".

Example program:

```
#include<stdio.h>
main()
{
printf("\n I am in main");
fun1();
printf("\n again I am in main");
getch();
}
fun1()
{
printf("\n I am in function1");
fun2();
}
fun2()
{
printf("\n I am in function2");
}
```

Function prototype:

Function prototype means a function have return type, function name, parameter list and must be end with semicolon(;),

Example program:

```
#include<math.h>
#include<stdio.h>
int squareroot(int);      ---→ function prototype
main()
{
```

```
int m,n;
clrscr();
printf("enter n value \n");
scanf("%d",&n):
m=squareroot(n);
printf("the squareroot of n is %d",m);
getch();
}
int squareroot(int x)
{
return(sqrt(x));
}
```

Storage classes:

The variables are declared by the type of data they can hold. During the execution of the program, these variables may be stored in the registers of the CPU or the primary memory of the computer. C provides four storage class specifier they are:

1. automatic.
2. external.
3. registers.
4. static.

The storage class specifier precedes the declaration statement for a variable.

Syntax:

<storageclass specifier><data type><variable name>;

Automatic storage class:

By default all variables declared with in the body of any function are automatic. The keyword "auto" is used in the declaration of a variable to explicitly specify its storage class.

Ex:

```
auto int a=10;
```

"a" is a variable that can hold a integer and its storage class is automatic. Even if the variable declaration statement in the function body does not include the keyword "auto", such declared variables are implicitly specified as belonging to the automatic storage class.

Example program:

```
#include<stdio.h>
int i=10,x=20;
main()
{
auto int i=20;
printf("the I value is %d",i);
printf("\n x value is %d",x);
getch();
}
```

Automatic storage class variable scope starts when it is created and the scope ends when the function is closed.

Storage area: RAM

Default value: garbage value

Scope: local to the block in which the variable is defined.

Life time: till the control remains with in the block in which the variable is defined.

External storage class:

Usually a big program is divided into a number of small programs, so that maintenance of the program becomes easy. A variable defined in a program can be accessed by another program as well, such a variable is called as extern (external) variable.

The keyword for declaring such global variable is extern.

Syntax:

```
extern datatype variablename;
```

Example program-1:

```
#include<stdio.h>
int i=10;
main()
{
int i=20;
printf(" %d",i);
incre();
}
incre()
{
printf("\n%d",i);
getch();
}
```

A program can have same variable name as global variable and local variable. In such case the local will have precedence over the global variable.

Storage area: RAM

Default value: zero

Scope: global.

Life time: external.

Example program-2:

```
extern int i=10;
main()
{
clrscr();
printf("\n %d",i);
incr();
getch():
}
incr()
{
printf("\n %d",++i);
}
```

output:

1 0
1 1

Static storage class:

The static storage class used for fixed storage for variable within a specified region. Two kinds of variables are allowed to be specified as static variables: static variables and local variables. The local variables are also referred to as internal variables while the global variables are also known as external variables. The default value of static variables is zero.

Storage area: RAM
Default value: zero
Scope: local.
Life time: internal.

Syantax:

```
static int a=10;
```

Example program-1:

```
#include<stdio.h>
main()
{
clrscr();
printf("\n first class\t ");
show();
printf("\n second class \t");
show():
printf("third class \t");
show();
getch();
}
show()
{
static int i;
printf("%d",i);
i++;
}
```

Example program-2:

```
#include<stdio.h>
main()
{
int x;
clrscr();
printf("factorial of 1-7 \n");
printf(" %d\n",show(x));
}
getch();
}
show(int z)
```

```
{
static int i=1,fact=1;
for( ;i<=z;i++)
fact=fact*i;
return fact;
}
```

Registers storage class variables:

Variables stored in the registers of the CPU are accessed in much lesser time than those stored in the primary memory. The keyword this storage class is "registers", this keyword precedes the normal declaration statement of a variable. Example program:

```
#include<stdio.h>
int power(int,int)
main()
{
int num=4;
int k=3;
int x;
x=power(num,k);
printf("\n value of num=%d, raised to k=%d is %d",num,k,x);
}
int power(int p,rgister int q)
{
register int temp;
temp=1;
int i;
for(i=1;i<=q;i++)
temp=temp*i;
return temp;
}
```

Storage area: Registers
Default value: garbage value
Scope: local.
Life time: internal.

Scope rules:-

The scope relatives to the accessibility, the period of existence, and the boundary of usage of variables declared in a statement block or function. If any variables declared inside the block, that variables scope is in the block only. If any variables declared in the main(), that variable scope is with in the main and their sub blocks only. If any variable declared outside the main(). As global, that variable can accessed entire program means, main() and their sub program.

Block structure:

Block structure means it is the blocks of executable statements, constructed with curly braces({}). The block of statements enclosed with open curly brace and closed curly brace. The scope rules in the block of C code, lying with in curly braces({}),

basically specifies the accessibility duration of existence and boundary of usage of the variable.

Example program: for scope rules and block structure.

```
#include<stdio.h>
main()
{
int x=3;
printf("\n x value in outer block %d",x);
{
int x=45;
printf("\n x value in inner block %d",x);
}
printf("\n again x value in outer block %d",x);
getch();
}
```

Recursion:

A recursion function is one that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self call.

Need of recursion

1. decomposition into smaller problems of same type
2. recursive calls must reduce problem size
3. a recursive function would call itself indefinitely
4. recursion is like a top-down approach

Factorial of a number:

n!=n*(n-1)*(n-2)*……..*1 for n>0

0!=1

Example program:

```
#include<stdio.h>
main()
{
int m,n;
clrscr();
printf("enter n value");
scanf("%d",&n);
m=fact(n);
printf("factorial of n is %d",m);
getch();
}
fact(int x)
{
if(x==1)
return 1;
else
return(x*(fact(x-1));
}
```

Recursion for fibonacci sequence:

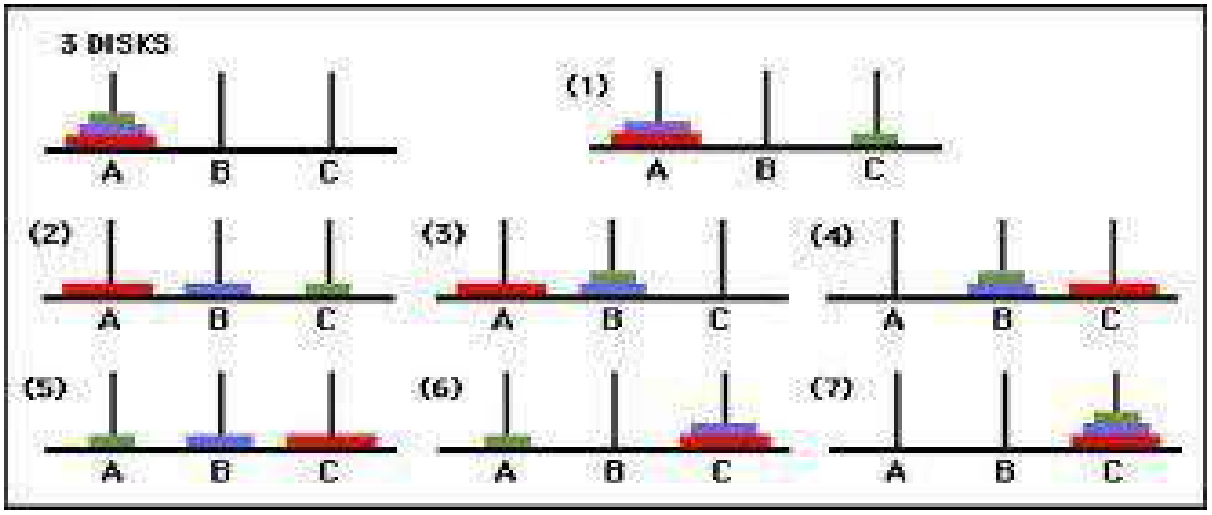
```
#include<stdio.h>
int fib(int val);
main()
{
int i,j;
clrscr();
printf("\n enter th number of terms");
scanf("%d",&i);
printf("Fibonacci sequence for %d terms is:",i);
for(j=0;j<=i;j++)
printf("%d",fib[j]);
getch();
}
int fib(int val)
{
if(val<=2)
return 1;
else
return(fib(val-1)+fib(val-2));
}
```

The towers of Hanoi:

The towers of Hanoi problem is a classic case study in recursion, it involves moving a specified number if disks from one tower to another using a third as an auxiliary tower.

Specification move n disks from peg A to peg C, using peg B as needed.

- Only one disk may be moved at a time.
- This disk must be the top disk on a peg.
- A larger disk can never b placed on top of a smaller disk.



Towers of Hanoi

The problem is not to focus on the first step, but on the hardest step i.e, moving the bottom disk to peg C.

- move disk3 from peg A to peg C
- move disk2 from pg A to peg B
- move disk3 from peg C to peg B
- moving disk1 from peg A to peg C
- moving disk3 from peg B to peg A
- moving disk2 from peg B to peg C
- moving disk3 from peg A to peg C

Parameter passing:

Parameter passing from calling function to called function is in two ways. One is call by value, second is call by reference.

Call by value:

While calling a function, the values in the arguments are passed by value to the formal parameters of the function. In fact, only copies of the values held in the arguments are sent to the formal parameters.

Example program:

```
#include<stdio.h>
main()
{
int a,b;
clrscr();
printf("enter a,b values");
scanf("%d%d",&a,&b);
printf("before swapping a=%d, b=%d",a,b);
swap(a,b);
getch();
}
swap(int x, int y)
{
int c;
c=x;
x=y;
y=c;
printf("after swapping a=%d, b=%d",x,y);
}
```

Call by reference:

In another function call technique known as "call by reference", more strictly termed as "call by address" where values are passed by handling over the address of arguments to the called function.

Example program:

```
#include<stdio.h>
main()
```

```
{
int a,b;
clrscr();
printf("enter a,b values");
scanf("%d%d",&a,&b);
printf("before swapping a=%d, b=%d",a,b);
swap(&a,&b);
getch();
}
swap(int *x, int *y)
{
int *c;
*c=*x;
*x=*y;
*y=*c;
printf("after swapping a=%d, b=%d",*x,*y);
}
```

C pre-processor:

C pre-processor is also called a macro processor. A macro is defined as an open-ended subroutine the preprocessor provides its own language that can be a very powerful tool for the programmer. These tools are instructions to the preprocessor and are called directions. The **C Preprocessor** is not part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP.

All preprocessor commands begin with a pound symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column.

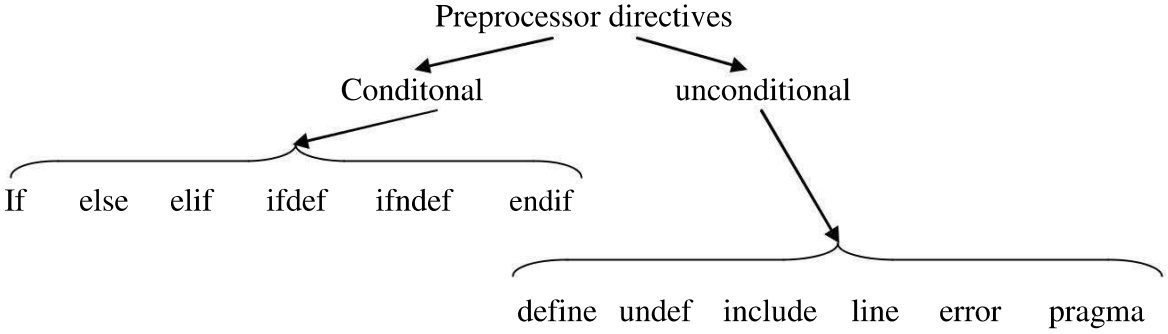
Following section lists down all important preprocessor directives:

Advantages:

1. Program development easier.
2. Program easier to read.
3. Modification of programs easier.
4. Transferable code.

Types of c preprocessor directives:

Pre-processor directives can be classified into two categories: unconditional and conditional.



Conditonal:

#if:

here #if is a conditional directive of the preprocessor

Syntax:

```
#if <expression>
<statement>
#endif
```

Ex:

```
#if a>0
printf("a is positive");
#endif
```

#else:

The #else is also usd with this directive if required. #if and #else pair operates in a away similar to the if-else.

Syntax:

```
#if <expression>
<statement>
#else
<statement>
#endif
```

Ex:

```
#if a>b
printf(" a is big");
#else
printf(" b is big");
#endif
```

#elif:

Syntax:

```
#if <expression>
<statement>
#elif<expression>
<statement>
#else
<statement>
#endif
```

Ex:

```
#if a>b&&a>c
printf(" a is big");
#elif b>c
printf(" b is big");
#else
```

```
printf(" c is big");
#endif
```

#ifdef and #ifndef:

The #ifdef directive executes a statement sequence if the macro-name is defined, if the macro-name is not defined, the #ifdef directive executes a statement sequence.

#ifdef:

Syntax:

```
#ifdef macroname
<statement sequence>
#endif
```

Ex:

```
#define a 1
#define b 0
main()
{
#ifdef a
printf(" a is true");
#endif
#ifdef b
printf(" b is false");
#endif
```

#ifndef:

Syntax:

```
#ifndef macroname
<statement sequence>
#endif
```

Ex:

```
#define a 1
#undef b 0
main()
{
#ifndef a
printf(" a is defined");
#endif
#ifndef b
printf(" b is not defined");
#endif
```

Unconditional:

#define:

#define directive is used to make substitutions throughout the program in which it is lacated.

Syntax:

```
#define macro-name replace-string
```

Ex:

```
#define max 10
main()
{
#if max>0
printf("condition is true");
#endif
 }
```

#undef:

This directive undefines a macro. A macro must be undefined before being redefined to different value. Undef takes single argument. Syntax:

```
#undef macro_name
```

Ex:

```
#undef value
#define value 10
#define max 20
main()
{
#if max>value
printf("condition is tru");
#endif
}
```

#include:

This directive include the files into present working directory.

Syntax:

```
#include<filename>
```

Ex:

```
void display()
{
int a=10;
printf("a value is %d",a);
}
```

This file saved as disp.c

```
#include<disp.c>
main()
{
clrscr();
display();
getch();
}
```

#error:

The directive #error is used for rporting errors by the preprocessors.

Syntax:

```
#error error_mssage
```

#line:

This directive is used to change the value of the line and file variables.

Syntax:

#line line_number <filename>

Ex: #line 20 disp.c

#pragma:

The #pragma directive is implementation specific, uses vary from compiler to compiler.

Header file creation:

In computer programming, a **header file** is a file that allows programmers to separate certain elements of a program's source code into reusable files. Header files commonly contain forward declarations subroutines, variables, and other identifiers. Programmers who wish to declare standardized identifiers in more than one source file can place such identifiers in a single header file, which other code can then include whenever the header contents are required. A header file is a file with extension **.h** which contains C function declarations and macro definitions and to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that come with your compiler.

Header file contains no.of functions, these functions are frequently used in all programs, user can also create the own header files.

Creation of header file:

```
int fact(int n)
{
int f=1,i=1;
if(n>7)
{
printf(“sorry! Can‟t find a factorial for a number>7”);
return -1;
}
else
{
for(i=1;i<=n;i++)
{
f=f*I;
}
return f;
}
}
int rverse(int n)
{
int r,rv=0;
while(n>0)
{
r=n%10;
rev=rv*10+r;
```

```
n=n/10;
}
return rev;
}
int evenodd(int n)
{
if(n%2==0)
printf("even");
else
printf("odd");
}
void Armstrong(int n)
{
int r,arm=0,n1;
n1=n;
while(n>0)
{
r=n%10;
arm=arm+(r*r*r);
n=n/10;
}
if(n1==arm)
printf("Armstrong");
else
printf("not Armstrong"):
}
```

Save the above file as prasad.h (we cant execute this file because main() is absent)

```
#include<stdio.h>
#include<prasad.h>   →use of header file prasad.h
main()
{
int n1,n2,ch;
clrscr();
printf("enter n1 value");
scanf("%d",&n1);
printf("1.factorial\n 2.reverse\n 3.evenodd\n 4.armstrong");
printf("enter ur choice from above menu");
scanf("%d",&ch);
switch(ch)
{
case 1: n2=fact(n1);
            if(n2!=-1)
            printf("factorial n1 is %d", n2);
            break;
case 2: n2=reverse(n1);
        printf("revrse no.of n1 is %d",n2);
```

```
                    break;
        case 3: evenodd(n1);
                    brak;
        case 4: armstrong(n1);
                    break;
        default: printf("your choice is wrong");
        }
        }
```

Passing 1-D arrays, 2-D arrays to functions:

Passing arrays to functions:

Arrays can also be arguments of functions. When an entire array is an argument of a function, only the address of the array is passed and not the copy of the complete array. Therefore, when functions is called with the name of the array as the argument.

Passing 1-D array:

Syntax:

<return type><function name>(default arrayname[size]);

Example:

```
#include<stdio.h>
int maximum(int []);
main()
{
int values[5],i,max;
printf("enter elements \n");
for(i=0;i<5;i++)
scanf("%d",& values[i]);
max=maximum(values);
printf("maximum element is %d\n",max);
getch();
}
int maximum(int b[])
{
int max=0,i;
for(i=0;i<5;i++)
{
if(b[i]>max)
max=b[i];
}
return max;
}
```

Passing 2-D array to functions:

Syntax:

<returntype><function name>(datatype arrayname[size1][size2]);

Example:

```
#include<stdio.h>
main()
```

```
{
int a[2][2],i,j,s;
clrscr();
printf("enter array elements \n");
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
scanf("%d",&a[i][j]);
}
}
s=sum(a);
printf("sum is %d\n",s);
getch();
}
sum(int b[2][2])
{
int i,j,s;
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
s=s+b[i][j];
}
}
return s;
}
```