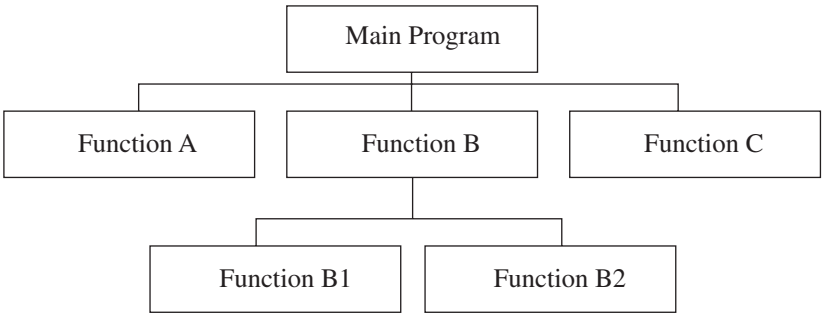
UNIT – IV

CHAPTER

FUNCTIONS

14.1 INTRODUCTION

Functions are sub programs which perform a specific task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions. Functions support modularity in software engineering. That is, a complex problem is divided into easily manageable modules of functions which can be invoked in necessary places. Similarly repetitive operations are specified within the function only once and it can be invoked in several places. So functions are used to reduce the length of the program.



The C language supports two types of functions. They are:

1. Library Functions
2. User defined functions

Library functions are built in functions. They are written and maintained by a compiler. The users can use the functions but cannot modify or change them. User defined functions are defined by the user according to their requirements. The user can modify the functions. The user understands the internal working of the functions.

14.2 COMPONENTS OF USER DEFINED FUNCTIONS

User defined functions have the following components:

1. Function declaration – known as function prototype
2. Function definition
3. Function usage – function call or function invocation

14.3 FUNCTION DECLARATION

Identifiers are used to identify the functions. All identifiers need to be declared before their first usage. Function names are identifiers, so all functions should be declared and defined properly. The function declaration may be global or local. Global declaration means, it should be declared before void main. Local declaration means, it should be declared within the declaration part of main program.

The general syntax of a function declaration is as follows:

return-type fun-name(parameter-list or parameter-type-list);

The function declaration provides the following information to the compiler at compilation time:

1. **return-type** specifies the type of data returned by the function. Suppose the function does not return any value, it is specified by void.
2. **fun-name** specifies name of the function. Function names are unique for each function. It should not be a keyword.
3. **parameter-list or parameter –type-list** specifies number of parameters or number and type of parameters passed into the function during function call.

14.4 DEFINITION OF FUNCTIONS

A function definition is an independent program that performs a specific task. Function definition includes function header and function body. Function body contains list of instructions enclosed within a pair of brackets.

The general syntax of function definition:

```
<return-type> fun-name(argument/parameter list)
{
        // body of the function
}
```

If the function returns a value, it should be specified by return-type specification. In that situation the function must include a **return** statement. If the function does not return anything then it should be **void**.

fun-name is a valid C identifier. It follows the same rules as normal identifier. Avoid using keywords or library function names as user defined fun-name. Function definition includes formal parameter list. Each variable in the list must have an individual data type declaration separated by a comma separator.

Example:

```
float divide(int a, int b, int c )      // function header
{
   float R;                // function body
   R = ( a + b + c) / 3 ;
   return(R);              // return statement
}
return statement can be any one of the following form.
   return;                 // does not return any value
   or
   return(expression);
```

14.5 FUNCTION USAGE OR FUNCTION CALL

14.5.1 Working of a Function

A function can be called by simply using the function name followed by a list of actual parameters list.

```
void main()
{
    -----;
    -----;
    abc(x,y);  -> Function call
                  x, y are actual arguments
    -----
    -----
}

abc(int l, int K)    Function definition
{                    l, k are formal or dummy arguments
    ----
    ----
    ----
    return();        return statement
}
```

EXAMPLE PROGRAM

```
Void main()
{
int x,y,z;
float Result;
scanf("%d%d%d",&x,&y,&z);
Result = divide(x,y,z);
```

```
    printf(%f",Result);
    }
float divide(int a, int b, int c )
{
  float R;
  R = ( a + b + c) / 3 ;
  return(R);
}
```

Output

```
10
100
1000
```

1. **Actual Arguments**: The argument list in the function calls is called actual arguments. It should match both in number and type with formal argument listed in the function declaration.
2. **Formal/Dummy Arguments**: The argument list in the function definition is called formal or dummy arguments. While executing the function, the formal arguments are replaced by the actual argument values.
3. **Function Name:** A name given to the function similar to variable name.
4. **Variables:** There are two kinds of variables. They are local and global.
5. **Local Variables:** The variables which are declared inside the function definition are called local variables.
6. **Global Variable**: Variables which are declared outside the main function are called global variables.
7. **Return Value**: The result obtained by the function is sent back by the function to the function call through the return statement. Maximum of a function returns only one value at a time.

The working of a function is given below:

In the above example, the main program instructions are executed one by one. Initially it executes the declaration statement and allocates a three 2 byte memory for the three integers x, y, and z, and a four byte memory for the float variable Result. scanf statement reads three integers and stores all the three values in the respective locations already allocated for x, y and z. Next instruction includes function call for divide function with actual list of arguments. Now the compiler suspends the current program execution (void main) and stores the return address, transfers the control to named sub program.

14.6 CATEGORY OF FUNCTION PROTOTYPES

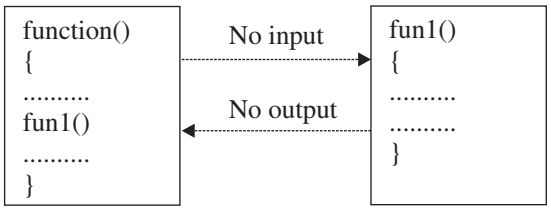
The function prototypes are classified into four types. They are:

1. Function with no argument and no return value.
2. Function with no argument and with return value.

3. Function with argument and no return value.
4. Function with argument and with return value.

1. Function with no argument and no return value

The function which includes invocation of another function is called calling function. The function which is invoked is referred as called function.



No data communication

When a function has no argument and does not return any value from the called function, it is mentioned by the void return type specification in function declaration and also in function header in definition. No data communication between function and only the controls alone are transferred. The dotted line indicates there is only a transfer of control but no data.

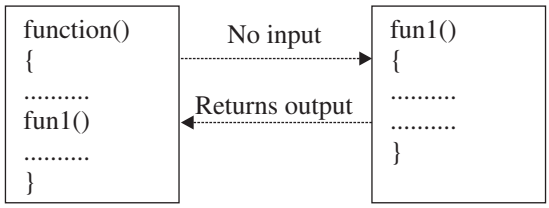
- ❖ Neither data is passed through the calling function nor is the data sent back from the called function.
- ❖ There is no data transfer between calling and the called function.
- ❖ The function is only executed and controls alone transferred.
- ❖ The function acts independently. It reads data values and print result in the same block.

EXAMPLE PROGRAM

```
#include<stdio.h>
void add();   //function with no arguments
void main()
{
clrscr();
add();
getch();
}
void add()    //function with no return type
{
int a,b,c;
a=10;
b=5;
c=a+b;
printf("\nThe sum=%d",c);
}
```

```
Output
The sum = 15
```

2. Function with no argument and with return value



One way data communication

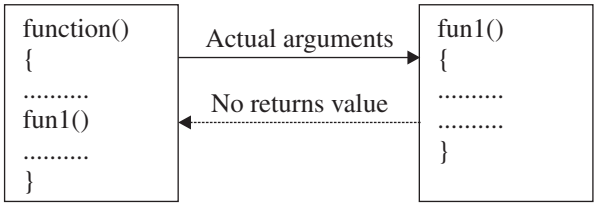
- In the above type of function, no arguments are passed from the main function (specified by dotted line). But the called function returns the value (specified by solid line).
- The called function is independent. It reads values from the keyboard and returns value to the function call.
- Called function must include a return statement.
- Here both the called and calling functions partly communicate with each other.

EXAMPLE PROGRAM

```
#include<stdio.h>
int add();
void main()
{
int a,b,c;
clrscr();
c=add();
printf("\nThe sum=%d",c);
getch();
}
int add() //with return type and return type
{
int a,b,c;
a=10;
b=5;
c=a+b;
return(c);
}
```

```
Output
The sum = 15
```

3. Function with argument and no return value



One way data communication

- ❖ In the above type of function, arguments are passed through the calling function. The called function operates on the values but no result is sent back.
- ❖ The functions are partly dependent on the calling function. The result obtained is utilized by the called function.

EXAMPLE PROGRAM

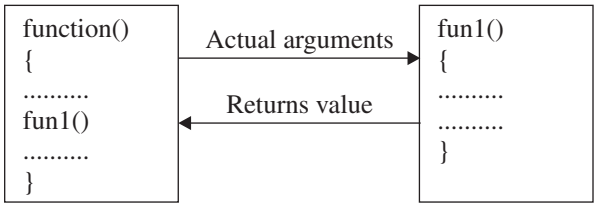
```
#include<stdio.h>
void add(int,int);
void main()
{
int a,b;
clrscr();
a=10;b=5;
add(a,b);
getch();
}

void add(int a,int b)//with arguments and no return type
{
int c;
c=a+b;
printf("\nThe sum=%d",c);
}
```

Output

```
The sum=15
```

4. Function with argument and with return value



Two way data communication

- ❖ In the above type of function, data is transferred between calling and called function.
- ❖ Both communicate with each other.
- ❖ The list of actual arguments passed from the calling must match in number and type with the list of formal arguments listed in the called function.

EXAMPLE PROGRAM

```
#include<stdio.h>
int add(int,int);     //with arguments and return type
void main()
{
int a,b,c;
clrscr();
a=10;b=5;
c=add(a,b);
printf("\nThe sum=%d",c);
getch();
}
void add(int a,int b)
{
int c;
c=a+b;
return(c);
}
```

Output

```
The sum = 15
```

14.7 PARAMETER PASSING METHODS

There are two ways by which arguments are passed in the function. They are:

1. Call by value
2. Call by reference

1. Call by value: In this type, values of actual arguments are passed to the formal arguments and the operation is done on the dummy arguments. Any change made in the formal arguments does not affect the actual arguments because formal arguments are photocopies of actual arguments.

EXAMPLE PROGRAM

```
#include<stdio.h>
void swap(int,int);
void main()
{
int a,b;
```

```
clrscr();
a=10;
b=5;
swap(a,b);
getch();
}
void swap(int a,int b)
{
int c;
c=a;
a=b;
b=c;
printf("\nAfter swapping\n");
printf("\na=%d",a);
printf("\nb=%d",b);
}
```

Output

```
After swapping
a = 5
b = 10
```

2. Call by Reference: In this type, addresses are passed. Function operates on addresses rather than values. Here, the formal arguments are pointers to the actual arguments. Any changes made within the function affects the original value. So it is reflected in the actual argument's values.

EXAMPLE PROGRAM

```
#include<stdio.h>
void swap(int*,int*);
void main()
{
int a,b;
clrscr();
a=10;b=5;
printf("\n The values before swapping …");
printf("\n a = %d \n b = %d",a,b);
swap(&a,&b);
printf("\nAfter swapping\n");
printf("\na=%d",a);
printf("\nb=%d",b);
getch();
}
void swap(int *x,int *y)
{
int c;
```

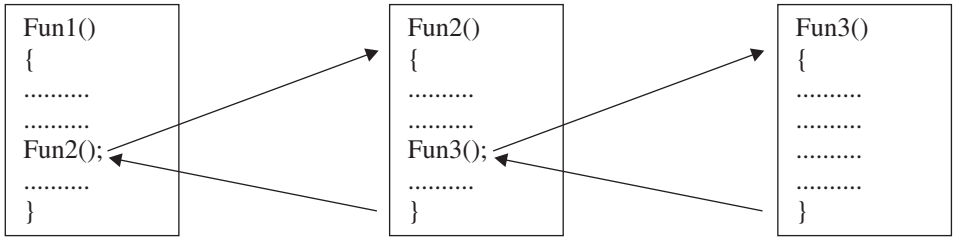
```
c=*x;
*x=*y;
*y=c;
}
```

Output

```
The values before swapping …
a = 10
b = 5
After swapping
a = 5
b = 10
```

14.8 NESTING OF FUNCTIONS

When a function calls another function which includes one more function call then it is called nesting of function.



Nesting of three functions

Example:

Write a C program to calculate $\frac{A}{B-C}$ and print the result.

This can be implemented by three functions. main() function reads the input values and calls the ratio() function, ratio() calls another function diff() to find the B-A value, and calculates RESULT, and returns that value to main().

EXAMPLE PROGRAM

```
#include<stdio.h>
#include<conio.h>
float ratio(int,int,int);
     int diff(int , int);
void main()
{
int A,B,C;
float Result;
printf("\n  Enter the input values ….")
scanf("%d%d%d",&A,&B,&C);
```

```
Result = ratio( A,B,C);
printf("\n   Result  = %f ",Result);
}
float ratio(int x,int y, int z)
{
int  t;
t = diff(y,z);
if(!=0)
     return( x / t);
else
     return(0.0);
}
int diff(int p, int q)
{
     if( p != q)
     return( p - q);
else
     return(0);
}
```

Output

```
Enter the input values …
5    4   2
Result = 2.50000
```

14.9 RECURSION

When a function calls itself, then it is called a recursive function or function recursion. Recursive function must include a conditional statement which forces to exit from the execution of body of the recursive function. Otherwise the function will never stop its execution.

Example program which clearly explains the working of a recursive function is to find the factorial of a given number.

EXAMPLE PROGRAM

```
#include<stdio.h>
#include<conio.h>
long int fact((int);
void main()
{
long int L;
int n;
printf("\n Enter the input…");
scanf("%d",&x);
L = fact(x);
```

```
Printf("\n  Factorial of %d is %ld",x,L);
}

long int fact(int n)
{
int f;
if ( n ==1)
      return(1);
else
      f = n * fact(n-1);
return(f);
}
```

Output

```
      Enter the input… 5
      Factorial of 5 is 120
Let us see how the recursion works.
Let us assume the input value x = 5
While executing the function fact, the value of n is not 1, so
      F = 5 * fact (4)
                  ↓
                  4 * fact(3)
                        ↓
                        3 * fact (2)
                              ↓
                              2 * fact (1)
                                    ↓
                                    1
At this iteration, n = 1, will return 1.
So the final answer is
            fact = 5 * fact(4)
                 = 5 * 4 * fact(3)
                 = 5 * 4 * 3 * fact(2)
                 = 5 * 4 * 3 * 2 * fact(1)
                 = 5 * 4* 3 * 2 * 1
                 = 120
```

14.10 SCOPE, VISIBILITY AND LIFE TIME OF A VARIABLE IN FUNCTIONS

In C, not only do all the variables have data type, they also have storage classes. The following storage classes are more relevant to functions.

1. Automatic variables
2. External variables
3. Static variables
4. Register variables

The above storage classes have its own scope, visibility and life time. Scope of a variable defines up

to what part of a program, the variable is active. Visibility refers to the accessibility of a variable from the memory.

1. Automatic variables: Automatic variables are created and utilized within the function. They can be destroyed automatically when they exit from the function. Hence, the name automatic. The keyword **auto** can be used explicitly to declare automatic variables. They are local to the particular functions only. Because of this property, automatic variables are also referred as local or internal variables. A variable declared inside a function without storage class specification is, by default an automatic variable.

Example:

```
Void main()
{
int  num;
---------
--------
}
Void addition()
{
auto int num;
----------
---------
}
```

The above example shows that the same variable name is used in more than one function, because it can be created and destroyed within the same function without any confusion to the compiler.

EXAMPLE PROGRAM

```
#include<stdio.h>
#include<conio.h>
void fun1(void);
void fun2( );
void main( )
{
int m = 1000;
fun2( );
printf("\n %d", m);
}

Void fun1( )
{
int m = 100;
printf("\n %d", m);
}
```

```
Void fun2( )
{
auto int m = 10;
fun1( );
printf:\n %d", m);
fun1();
}
```

Output

```
10
100
1000
```

2. External variables: External variables are declared globally, above all function definitions including main() function. These variables are active and alive throughout the program. Unlike normal variables, external variables are accessed by any function in the program. The memory allocated for external variables are shared by all the functions which utilize them. So any modification done by any function will affect the original value. So, subsequent functions can refer that new value.

EXAMPLE PROGRAM

```
#include<stdio.h>
#include<conio.h>
int  fun1( );
int fun2( );
int fun3( );
int N;   // global declaration
void main()
{
    N = 10;
printf("\n   N = %d",N);
printf("\n   N = %d", fun1( ));
printf(:\n   N = %d", fun2( ));
printf("\n   N = %d", fun3( ) );
}
int fun1( )
{
    N = N +10;       // global declaration
}
int fun2( )
{
    int N;           //  local declaration
    N = 5;
    return(N);
}
```

```
int fun3()
{
    N = N+10;        // global declaration
}
```

Output

```
N = 10
N = 20
N = 5
N = 30
```

One other aspect of a global variable is that it is available from the point of declaration to the end of the program.

Example:

```
main()
{
   y = 5;
---------
--------
}
int y;  // global declaration
fun( )
{
y = y + 1;
}
```

As far as main is considered, the variable y is undefined. So the compiler generates an error message.

External declaration The above issue can be solved by the storage declaration **extern** as follows:

```
main()
{
  extern int y = 5;   // external declaration
---------
--------
}
fun( )
{
 extern int y = y + 1;    // external declaration
----------
----------
}
int y;  //  definition
```

The variable y has been declared after both the function definitions. The extern declaration of y in

both the functions informs to the compiler that y is an integer type variable defined somewhere in the same program. The extern declaration does not allocate memory for its storage.

3. Static variables: The static variable values persist until the end of the program. Any variable can be declared static using the keyword static. A static variable is initialized only once, when a program is compiled. It is never initialized again. Further reference take the pre assigned value as it is.

```
static int x;
static float y;
```

A static variable may be internal or external depending on the place of declaration. Internal static variables are declared inside the function. The scope of internal variables extends up to the end of the program. It is similar to auto variables.

EXAMPLE PROGRAM

```
#include<stdio.h>
#include<conio.h>
Void stat( );
Main ()
{
  int  i ;
  for ( i = 1; i < = 3 ; i++)
  stat( );
}

Void stat( )
{
  static int x = 0;
  x = x + 1;
  printf("\n  X = %d", x);
}
```

Output

```
X = 1
X = 2
X = 3
```

During the first call to **stat, x** is incremented to 1. Because x is static, this value persists and therefore next call to **stat** adds another 1 to **x**, now the value of x is 2.

4. Register Variables: Normally the computer will store all the variables in a physical memory.

Sometimes it is possible to store the values in registers. Since register access is much faster than memory access. So the frequently accessed variables are kept in registers which increases the speed of execution.

Declaration of register variable

register int count;

Registers are able to store only restricted size value, so most of the compilers will allow only int and char variables to be placed in the register. Registers are limited in count, therefore it is important to select required variables for this purpose. The C compiler automatically converts **register** variables into non register variables once the limit is reached,

14.11 PASSING ARRAYS TO FUNCTIONS

14.11.1 One Dimensional Array

Like a basic primitive normal variable, it is possible to pass the values of an entire array to a function. To pass a one dimensional array to a function, it is sufficient to mention the name of the array without any subscript, and the size of an array as next argument.

Example: sum(A,10);

This statement pass 10 values of an array named A into the function **sum**. The called function's declaration is as follows:

int sum(int P[ ], int size)

The header contains two arguments, the array name and the size of the array which specifies number of elements in the array. The declaration of formal argument is made as follows:

int P[ ]; // pair of brackets informs the compiler that P is an array of integer numbers. It is not necessary to specify the size explicitly.

The following is an example of C program to sort n numbers using array and functions. The main function is used to read an unsorted array element and to print the sorted array. A sorting function is used for sort procedure.

EXAMPLE PROGRAM

```
#include<stdio.h>
#include<conio.h>
Void sorting(int A[ ], int);
Void main()
{
int X[25],i,n;
printf("\n  Enter size of the array…");
scanf("%d",&n);
printf("\n Array elements….");
for(i=0;i<n;i++)
      scanf("%d",&X[i]);
printf(\\n Array elements before sorting....\n");
for(i=0;i<n;i++)
      printf("\t %d",X[i]);
sorting(X,n);
```

```
printf(\\n Array elements after sorting....\n");
for(i=0;i<n;i++)
      printf("\t %d",X[i]);
}
      void sorting(int A[],int m)
      {
      int t,i,j;
      for(i=0;i<n;i++)
      {
           for(j=i+1;j<n;j++)
               {
               if( A[i] >= A[j])
                    {
                    t = A[i];
               A[i] = A[j];
               A[j] = t;
               }
           }
      }
}
```

Output

```
Enter size of the array…
5
Array elements….
18
6
11
78
33
Array elements before sorting....
18    6           11          78         33

Array elements after sorting....
6     11          18          33         78
```

14.11.2 Two Dimensional Arrays

Like one dimensional array, we can pass multidimensional arrays to functions.

The following rules must be adopted.

1. The function must be invoked by passing the array name only.
2. In the function definition, include two sets of brackets to indicate that the array has two dimensions.
3. The size of the second dimension must be specified.
4. The prototype declaration should be similar to the function header.

The following is the function program to find the addition of two matrices. Here two different functions are used. One function is used to read a matrix, another one is used to print the matrix.

EXAMPLE PROGRAM

```
#include<stdio.h>
#include<conio.h>
void readmat(int M[][3],int,int);
void printmat(int M[][3],int,int);
void main()
{
int A[3][3],B[3][3],C[3][3],i,j;
int r=3,c=3;
readmat(A,3,3);
readmat(B,3,3);
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
C[i][j] = A[i][j] + B[i][j];
}
}
printf("\n First Matrix is ….");
printmat(A,3,3);
printf("\n Second Matrix is ….");
printmat(B,3,3);
printf("\n Addition Matrix is ….");
printmat(C,3,3);
           }

           void readmat(int M[][3],int,r1,int c1)
           {
              int i,j;
              printf("\n Enter the elements of a 3 * 3 Matrix…\n");
              for(i=0;i<r1;i++)
  {
   for(j=0;j<c1;j++)
   {
      scanf("%d",&M[i][j];
     }
 }
}

           void printmat(int M[][3],int,r1,int c1)
           {
              int i,j;
              printf("\n \n");
```

```
            for(i=0;i<r1;i++)
  {
   for(j=0;j<c1;j++)
   {
      printf("\t%d",M[i][j];
     }
 printf("\n");
 }
}
```

Output

```
Enter the elements of a 3 * 3 Matrix…
2
2
2
2
2
2
2
2
2
Enter the elements of a 3 * 3 Matrix…
3
3
3
3
3
3
3
3
3
First Matrix is ….
2       2       2
2       2       2
2       2       2

Second Matrix is ….
3     3     3
3     3     3
3     3     3

Addition Matrix is….
5     5    5
5     5    5
5     5    5
```