Notes By: Shailesh Bdr. Pandey, TA, Computer Engineering Department, Nepal Engineering College

Introduction

Functions are a useful way of referring to blocks of code, allowing the code to be called many times. Functions take some input information (the *input parameters*), do something with that information and then return a result (the *return value*).

We are familiar with the predefined functions, such as pow( ), getch( ), printf( ) and scanf( ). We call them *Library Function or Built-In Functions.* It is possible to define your own functions in C. Being able to do this allows powerful programs to be developed in a modular manner that makes the development and debugging stages significantly easier than would otherwise be the case.

The basic philosophy of function is divide and conquer by which a complicated tasks are successively divided into simpler and more manageable tasks which can be easily handled. A program can be divided into smaller subprograms that can be developed and tested successfully.

A function is a complete and independent program which is used (or invoked) by the main program or other subprograms. A subprogram receives values called arguments from a calling program, performs calculations and returns the results to the calling program.

There are many advantages in using functions in a program they are:

1. It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.
2. the length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigation.
4. A function may be used by many other programs this means that a c programmer can build on what others have already done, instead of starting over from scratch.
5. A program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.
6. Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program

We already know that C support the use of library functions and use defined functions. The library functions are used to carry out a number of commonly used operations or calculations. The user-defined functions are written by the programmer to carry out various individual tasks.

Functions are used in c for the following reasons:

1. Many programs require that a specific function is repeated many times instead of writing the function code as many timers as it is required we can write it as a single function and access the same function again and again as many times as it is required.

2. We can avoid writing redundant program code of some instructions again and again.
3. Programs with using functions are compact & easy to understand.
4. Testing and correcting errors is easy because errors are localized and corrected.
5. We can understand the flow of program, and its code easily since the readability is enhanced while using the functions.
6. A single function written in a program can also be used in other programs also.

Defining Functions
A function is defined as follows:

```
type function_name(parameter_list)
  {
    /*
       code block
    */
  }
```

type
Functions return a value. The type of this value (int, float, char etc) has to be specified. All of the standard C types met so far may be used as a function type. Sometimes a return value is not required, for example if the function prints an error message and then returns, in which case a return type of void should be used.

function_name
This is a label that is used to refer to the function from other code blocks. The rules for valid names are the same as those for variables.

parameter_list
Values are passed to a function via its parameter list. This is a comma separated list of the values that the function takes. Each item in the list has a type specification. The parameter list can be blank, i.e. the function takes no parameters, in this case the keyword void should be used instead of a list (as is the case with "int main(void)". The names used for the parameters can be different from those used in the block of code that calls the function.

code block
The code block contains the operations that constitute the function. The parameters declared in the parameter list can be used as standard variables within this block. Temporary variables may also be declared for use **within** the block. Unless the function has a return type of void then the function needs to return a value of the correct type. This is done via the return command (see below).

A simple example
The code example below shows a simple example of defining and using a function.

```
/*****************************************************************
*           Example 1
*
```

```
*            Simple function
*
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

float product(float x, float y)
        // function to return the product of two numbers see note 2
   {
     float result;                          // a local variable - see note 3

     result = x*y;

     return result;                    // returning the value - see note 4
   }

int main(void)
   {
     float a;
     float b;
     float answer;

 // prompt user for values
     printf("Please input two values to be multiplied:");
     scanf("%f %f",&a,&b);

// call the function product
     answer=product(a,b);          // calling the function - see note 5

     printf("The product of %f and %f is %f\n",a,b,answer);

     system("pause");
     return 0;
   }
```

Notes

1. This example is rather too simple to be useful, but does illustrate how functions are defined and used.
2. The function product is defined here. It has two input parameters and returns a value of type float.
3. result is a variable declared within the product code block. It can only be used in this block.
4. The function returns the value stored in result (which is of type float).
5. The values of a and b are passed to the function product, the result is stored in the variable answer. Notice that different parameter names are used in the function and main block.

Declaring Versus Defining Functions

In example 1 the code for the function product appeared before the main block. This is fine for such a simple program, but in a realistic situation where there may be many

tens if not hundreds of functions a program would rapidly become too difficult to follow. In such a situation it would be much better if the main() block comes first, followed by the functions. This is achieved using *function prototypes*. These are essentially the first line of a function definition terminated with a semicolon. For the function product() the prototype would be:

float product (float x, float y);

A function prototype tells the compiler about the function parameters and the return type - it is said to *declare* the function. The compiler uses this information to leave placeholders in the compiled code which is then filled in once the code that *defines* the code has been compiled. This definition code could come after the main function block or even be in a separate file. A function prototype must appear before the code block in which the function is used.

Example 1 could be rewritten as:

```
/*****************************************************************
*           Example 2
*
*           Simple function - with prototype
*
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

float product(float x, float y);  // see note 1.

int main(void)
{
 float a;
 float b;
 float answer;

 // prompt user for values
   printf("Please input two values to be multiplied:");
   scanf("%f %f",&a,&b);

// call the function product
   answer=product(a,b);

   printf("The product of %f and %f is %f\n",a,b,answer);

   system("pause");
   return 0;
}

float product(float x, float y)          // see note 2
        // function to return the product of two numbers
{
  float result;                          // a local variable
```

```
  result = x*y;

  return result;                    // returning the value
}
```

Notes

1. The function is **declared** here.
2. The function is **defined** here.
3. The function type and parameter list types must be the same in the declaration and definitions.

In programs that use many (>5 say) functions the prototypes are generally stored in a separate header file and this file is then added to the #include list. Includes of a file written by the programmer use speech marks instead of the angular brackets ,<>, around the file names, e.g.
#include "declarations.h"
where declarations.h is a file that contains a list of function prototypes. The angular brackets tell the preprocessor that the file is located in one of the compiler's standard directories (folders).

Types of functions:
A function may belong to any one of the following categories:
1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions with no arguments and return values.

Functions with no arguments and no return values:
Let us consider the following program

```
/* Program to illustrate a function with no argument and no return values*/
/* #includes go here followed by the three prototypes */
void statement1();
void statement2();
void starline();

void main()
{
staetement1();
starline();
statement2();
starline();
}
/*function to print a message*/
void statement1()
{
printf("\n Sample subprogram output");
}

void statement2()
```

```
{
printf("\n Sample subprogram output two");
}

void starline()
{
int a;
for (a=1;a<60;a++)
printf("%c",'*');
printf("\n");
}
```

In the above example there is no data transfer between the calling function and the called function. When a function has no arguments it does not receive any data from the calling function. Similarly when it does not return value the calling function does not receive any data from the called function. A function that does not return any value cannot be used in an expression it can be used only as independent statement.

Functions with arguments but no return values:
The nature of data communication between the calling function and the arguments to the called function and the called function does not return any values to the calling function this shown in example below:
Consider the following:
Function calls containing appropriate arguments. For example the function call

value (500,0.12,5)

Would send the values 500,0.12 and 5 to the function value (p, r, n) and assign values 500 to p, 0.12 to r and 5 to n. the values 500,0.12 and 5 are the actual arguments which become the values of the formal arguments inside the called function.

Both the arguments actual and formal should match in number type and order. The values of actual arguments are assigned to formal arguments on a one to one basis starting with the first argument as shown below:

```
main()
{
function1(a1,a2,a3……an)
}

function1(f1,f2,f3….fn);
{
function body;
}
```

here a1,a2,a3 are actual arguments and f1,f2,f3 are formal arguments.
The no of formal arguments and actual arguments must be matching to each other suppose if actual arguments are more than the formal arguments, the extra actual arguments are discarded. If the number of actual arguments is less than the formal

arguments then the unmatched formal arguments are initialized to some garbage values. In both cases no error message will be generated.

The formal arguments may be valid variable names, the actual arguments may be variable names expressions or constants. The values used in actual arguments must be assigned values before the function call is made.

When a function call is made only a copy of the values actual arguments is passed to the called function. What occurs inside the functions will have no effect on the variables used in the actual argument list.

Let us consider the following program

```
/*Program to find the largest of two numbers using function*/
/* #include */

void largest(int, int);

main()
{
int a,b;
printf("Enter the two numbers");
scanf("%d%d",&a,&b);
largest(a,b)
}
/*Function to find the largest of two numbers*/
void largest(int a, int b)
{
if(a>b)
printf("Largest element=%d",a);
else
printf("Largest element=%d",b);
}
```

in the above program we could make the calling function to read the data from the terminal and pass it on to the called function. But function foes not return any value. So the return type of the function is void.

Functions with arguments and return values:

The function of the type Arguments with return values will send arguments from the calling function to the called function and expects the result to be returned back from the called function back to the calling function.

To assure a high degree of portability between programs a function should generally be coded without involving any input output operations. For example different programs may require different output formats for displaying the results. Theses shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program. In the above type of function the following steps are carried out:

1. The function call transfers the controls along with copies of the values of the actual arguments of the particular function where the formal arguments are creates and assigned memory space and are given the values of the actual arguments.

2. The called function is executed line by line in normal fashion until the return statement is encountered. The return value is passed back to the function call is called function.
3. The calling statement is executed normally and return value is thus assigned to the calling function.

Return value data type of function:
A C function returns a value of type int as the default data type when no other type is specified explicitly. For example if function does all the calculations by using float values and if the return statement such as return (sum); returns only the integer part of the sum. This is since we have not specified any return type for the sum. There is the necessity in some cases it is important to receive float or character or double data type. To enable a calling function to receive a non-integer value from a called function we can do the two things:

1. The explicit type specifier corresponding to the data type required must be mentioned in the function header. The general form of the function definition is

```
Type_specifier function_name(Type_specifier1 Argument1, Type_specifier2 Argument2, …)
 {
function statement;
}
```

The type specifier tells the compiler, the type of data the function is to return.

2. The called function must be declared at the start of the body in the calling function, like any other variable. This is to tell the calling function the type of data the function is actually returning. The program given below illustrates the transfer of a floating-point value between functions done in a multiple function program.

```
float add(float,float);
double sub(double,double);

main()
{
float x,y;
 x=12.345;
y=9.82;
printf("%f\n" add(x,y));
printf("%lf\n"sub(x,y);
}
float add(float a, float b)
 {
return(a+b);
}
double sub(double p, double q)
 {
return(p-q);
}
```

We can notice that the functions too are declared along with the variables. These declarations clarify to the compiler that the return type of the function add is float and sub is double. Here we can notice that in call to the function sub(), we are sending floats and are being received in doubles p and q. Since double is a bigger float, this is valid. But you cannot send a float and receive in a character.

Function with no arguments and return types:
Let us consider the following program

```
/* Program to illustrate a function with no argument and return values*/
/* #includes go here */
int add();

void main()
{
int sum;
sum=add();
printf("Sum of two nos:%d",sum);
}
int add()
{
        int a,b,sum;
        printf("Enter two nos:");
        scanf("%d%d",&a,&b);
        sum=a+b;
        return(sum);
 }
```

In the above example there is no data transfer between the calling function and the called function. When a function has no arguments it does not receive any data from the calling function. But when it does return value the calling function receives the data from the called function and we store it in a variable. Here the variable sum receives the summation of two numbers returned from the function add().

Nesting of functions:
C permits nesting of two functions freely. There is no limit how deeply functions can be nested. Suppose a function a can call function b and function b can call function c and so on. Consider the following program:

```
float ratio(int, int);
int difference(int, int);

void main()
{
int a,b,c;
scanf("%d%d%d",&a,&b,&c);
printf("%f\n",ratio(a,b,c));
}

float ratio(int x, int y, int z)
 {
```

```
if(difference(y,z))
return(x/y-z));
else
return(0,0);
}
int difference(int p, int q)
{
if(p!=q)
return(1);
else
return(0);
}
```

the above program calculates the ratio a/b-c; and prints the result. We have the following three functions:
main(), ratio() and difference()

main reads the value of a,b,c and calls the function ratio to calculate the value a/b-c) this ratio cannot be evaluated if(b-c) is zero. Therefore ratio calls another function difference to test whether the difference(b-c) is zero or not.

Functions and arrays:
We can pass an entire array of values into a function just as we pass individual variables. In this task it is essential to list the name of the array along with functions arguments without any subscripts and the size of the array as arguments

For example:
The call
summation(a,n);
Will pass all the elements contained in the array a of size n. The called function expecting this call must be appropriately defined. The summation function header might look like:

```
float summation(float array[], int size);
```

The function summation is defined to take two arguments, the name of the array and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:
float array[];
The above declaration indicates to compiler that the arguments array is an array of numbers. It is not necessary to declare size of the array here. While dealing with array arguments we should remember one major distinction. If a function changes the value the value of array elements then these changes will be made to the original array that passed to the function. When the entire array is passed as an argument, the contents of the array are not copied into the formal parameter array instead information about the address of the array elements are passed on to the function. Therefore any changes introduced to array elements are truly reflected in the original array in the calling function. We will deal with this further during pointers.

Example:

```
void summation(int a[], int);

void main()
{
int a[5]={1,2,3,4,5};
summation(a,5);
}

void summation(int a[], int size)
{
int i, sum;
sum=0;
for(i=0;i<size;i++)
        sum+=a[i];
printf("The Summation is %d",sum);
}
```

For 2-Dimensional Array, we have to mention in the prototype that the data we are sending is not an ordinary variable but an array just like in the case of 1-D Array. But here we include two sets of [ ] to indicate it is a 2D Array.

```
void summation(int a[][2], int row);
```

Here [2] indicates the number of columns in the array we are sending. The numbers of rows can also be mentioned inside the first [ ] brackets but is not mandatory. Here I have passed the number of rows and columns as a separate integer variable *row* and *col*.

Example:

```
void summation(int a[][2], int row, int col);

void main()
{
int a[3][2]={{1,2}, {4,7}, {8,3} };
summation(a, 3, 2);
}

void summation(int a[][2], int row, int col)
{
int i, j, sum;
sum=0;
for(i=0;i<row;i++)
        for(j=0;j<col;j++)
        sum+=a[i][j];
printf("Total:%d",sum);
}
```

The scope and lifetime of variables in functions:

The scope and lifetime of the variables define in C is not same when compared to other languages. The scope and lifetime depends on the storage class of the variable in c language the variables can be any one of the four storage classes:
1. Automatic Variables
2. External variable
3. Static variable
4. Register variable.

The scope actually determines over which part or parts of the program the variable is available. The lifetime of the variable retains a given value. During the execution of the program. Variables can also be categorized as local or global. Local variables are the variables that are declared within that function and are accessible to all the functions in a program and they can be declared within a function or outside the function also.

Automatic variables:
Automatic variables are declared inside a particular function and they are created when the function is called and destroyed when the function exits. Automatic variables are local or private to a function in which they are defined by default all variable declared without any storage specification is automatic. The values of variable remain unchanged to the changes that may happen in other functions in the same program and by doing this no error occurs.

/* A program to illustrate the working of auto variables*/

```
void main()
{
int m=1000;
function2();
printf("%d\n",m);
}

function1()
{
int m=10;
printf("%d\n",m);
}

function2()
{
int m=100;
function1();
printf("%d\n",m);
}
```

A local variable lives through out the whole program although it accessible only in the main. A program with two subprograms function1 and function2 with m as automatic variable and is initialized to 10, 100, 1000 in function 1 function2 and function3 respectively. When executes main calls function2 which in turns calls function1. When main is active m=1000. But when function2 is called, the main m is temporarily

put on the shelf and the new local m=100 becomes active. Similarly when function1 is called both previous values of m are put on shelf and latest value (m=10) become active, a soon as it is done main (m=1000) takes over. The output clearly shows that value assigned to m in one function does not affect its value in the other function. The local value of m is destroyed when it leaves a function.

External variables:
Variables which are common to all functions and accessible by all functions of a program are internal variables. External variables can be declared outside a function.

Example

```
int sum;
float percentage;

main()
{
…..
…..
}
function2()
{
….
….
}
```

The variables sum and percentage are available for use in all the three functions main, function1, and function2. if a local variable has the same name as the global variable which is syntactically correct then the global variable has precedence over the local variable. A global value can be used in any function all the functions in a program can access the global variable and change its value the subsequent functions get the new value of the global variable, it will be inconvenient to use a variable as global because of this factor every function can change the value of the variable on its own and it will be difficult to get back the original value of the variable if it is required.

Global variables are usually declared in the beginning of the main program i.e., before the main program however c provides a facility to declare any variable as global this is possible by using the keyword storage class extern. Although a variable has been defined after many functions the external declaration of y inside the function informs the compiler that the variable y is integer type defined somewhere else in the program. The external declaration does not allocate storage space for the variables. In case of arrays the definition should include their size as well. When a variable is defined inside a function as extern it provides type information only for that function. If it has to be used in other functions then again it has to be re-declared in that function also.

Example:

```
main()
{
int n;
out_put();
```

```
extern float salary[];
……
…..
out_put();
}

void out_put()
{
extern float salary[];
int n;
….
…..
}
float salary[size];
```

a function when its parameters and function body are specified this tells the compiler to allocate space for the function code and provides type info for the parameters. Since functions are external by default we declare them (in calling functions) without the qualifier extern.

Functions and structures:
We can pass structures as arguments to functions. Unlike array names however, which always point to the start of the array, structure names are not pointers. As a result, when we change structure parameter inside a function, we don't affect its corresponding argument.

Passing structure elements to functions:
A structure may be passed into a function as individual member or a separate variable. A program example to display the contents of a structure passing the individual elements to a function is shown below.

```
# include < stdio.h >
void main()
{
int emp_id;
char name[25];
char department[10];
float salary;
};

struct emp1={125,"sampath","operator",7500.00};
/* pass only emp_id and name to display function*/
display(emp1.emp_id,emp1.name);
}

/* function to display structure variables*/
display(int e_no,char e_name[])
 {
printf("%d%s",e_no,e_name);
```

in the declaration of structure type, emp_id and name have been declared as integer and character array. When we call the function display() using
display(emp1.emp_id,emp1.name);
we are sending the emp_id and name to function display();
it can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing a better way would be to pass the entire structure variable at a time.

Passing entire structure to functions:
In case of structures having to having numerous structure elements passing these individual elements would be a tedious task. In such cases we may pass whole structure to a function as shown below:

```
# include stdio.h>
{
int emp_id;
char name[25];
char department[10];
float salary;
};

void main()
{
struct employee emp1=
{
12,
“sadanand”,
“computer”,
7500.00
};

/*sending entire employee structure*/
display(emp1);
}

/*function to pass entire structure variable*/
display(struct employee empf)
{
printf(“%d%s,%s,%f”, empf.empid,empf.name,empf.department,empf.salary);
}
```

Recursion
In C it is possible for a function to call itself, a process known as recursion. [Note: Not all computer languages allow recursion. FORTRAN77, a language widely used for numerical applications, is one of those that do not.] A function that calls itself is said to be *recursive*. A recursive function requires a call to itself and a test to see if some stop condition has been met bringing to an end the recursion.
Why would you want to use recursion? Well in some circumstances it can lead to rather elegant programming solutions. Consider for example the calculation of a factorial. You might initially think of using a loop to do this:

```
unsigned int factorial (unsigned int n)
{
  int i;
  int m=1;

  if (n>=1)
     for (i=1;i<=n;i++) m*=i;

  return m;
}
```

Note the use of unsigned int in order to ensure that only positive integers are used. Another way of calculating a factorial comes out of realizing that:

$$n! = n.(n-1)!$$

i.e. n factorial is n times (n-1) factorial.
This can be implemented in C as:

```
unsigned int factorial (unsigned int n)

{
   if (n<=1) return 1;          // Note 1:

   return (n*factorial(n-1)); // Note 2

}
```

Notes

1. This line terminates the recursion process.
2. This is the recursive call to factorial
3. This function has two return statements. This is allowed in C

Recursion is not just for moderately obscure mathematical functions. It turns out that many problems when considered properly can be written in terms of recursion. The reading/writing of lists from/to a file can readily be implemented in a recursive manner. Another example is the Towers of Hanoi problem that was very popular as a toy in Victorian times.