FUNCTIONS

LEARNING C PROGRAMMING

1 INTRODUCTION

In the earlier lessons we have already seen that C supports the use of library functions, which are used to carry out a number of commonly used operations or calculations. C also allows programmers to define their own functions for carrying out various individual tasks. In this lesson we will cover the creation and utilization of such user-defined functions.

2 OBJECTIVES

After going through this lesson you will be able to
explain of function
describe access to function
define parameters data types specification
explain function prototype and recursion
define storage classes – automatic, external, static variables

3 MODULAR APPROACH

The use of user-defined functions allows a large program to be broken down into a number of smaller, self-contained components, each of which has some unique, identifiable purpose. Thus a C program can be modularized through the intelligent use of such functions. There are several advantages to this modular approach to program development. For example many programs require a particular group of instructions to be accessed repeatedly from several different places within a program. The repeated instruction can be placed within a single function, which can then be accessed whenever it is needed. Moreover, a different set of data can be transferred to the function each time it is accessed. Thus, the use of a function avoids the need for redundant (repeating) programming of the same instructions.

The decomposition of a program into individual program modules is generally considered to be an important part of good programming.

4 DEFINING A FUNCTION

The question arises what is a function? So, function is a self-contained program segment that carries out some specific well-defined task. Every C program consists of one or more functions. The most important function is main. Program execution will always begin by carrying out the instruction in main. The definitions of functions may appear in any order in a program file because they are independent of one another. A function can be executed from anywhere within a program. Once the function has been executed, control will be returned to the point from which the function was accessed. Functions contains special identifiers called parameters or arguments through which information is passed to the function and from functions information is returned via the return statement. It is not necessary that every function must return information, there are some functions also which do not return any information for example the system defined function printf.

Before using any function it must be defined in the program. Function definition has three principal components: the first line, the parameter declarations and the body of the functions.

The first line of a function definition contains the data type of the information return by the function, followed by function name, and a set of arguments or parameters, separated by commas and enclosed in parentheses. The set of arguments may be skipped over. The data type can be omitted if the function returns an integer or a character. An empty pair of parentheses must follow the function name if the function definition does not include any argument or parameters.

The general term of first line of functions can be written as:
data-type function-name (formal argument 1, formal argument 2...formal argument n)
The formal arguments allow information to be transferred from the calling portion of the program to the function. They are also known as parameters or formal parameters. These formal arguments are called actual parameters when they are used in function reference. The names of actual parameters and formal parameters may be either same or different but their data type should be same. All formal arguments must be declared after the definition of function. The remaining portion of the function definition is a compound statement that defines the action to be taken by the function. This compound statement is sometimes referred to as the body of the function. This compound statement can contain expression statements, other compound statements, control statements etc. Information is returned from the function to the calling portion of the program via the return statement. The return statement also causes control to be returned to the point from which the function was accessed.
In general terms, the return statement is written as
return expression;
The value of the expression is returned to the calling portion of the program. The return statement can be written without the expression. Without the expression, return statement simply causes control to revert back to the calling portion of the program without any information transfer. The point to be noted here is that only one expression can be included in the return statement. Thus, a function can return only one value to the calling portion of the program via return. But a function definition can include multiple return statements, each containing a different expression. Functions that include multiple branches often require multiple returns.
It is not necessary to include a return statement altogether in a

program. If a function reaches the end of the block without encountering a return statement, control simply reverts back to the calling portion of the program without returning any information.

Let us consider an example of function without returning any information.

```
#include <stdio.h>
main()
{
int x,y;
maxi(int, int); /*function declaration*/
printf("Enter two integer values");
scanf("%d %d"' &x,&y);
maxi(x,y); /*call to function*/
}
maxi(x,y)
/*function definition*/
int x,y;
{
int z;
z=(x>=y)?x:y;
print("\n\n Maximum value %d",z);
return;
Body of the
function
maxi
}
```

This 'maxi' function do not return any value to the calling program, it simply returns the control to the calling programs, so if it is even not present, then also program will work efficiently.

Most C compilers permit the keyword void to appear as a type specifies when defining a function that does not return anything. So the

function definition will look like this if void is add to it
void maxi (int, int);

5 ACCESSEMENT OF A FUNCTION

A function can be accessed by specifying its name, followed by a list of parameters or arguments enclosed in parentheses and separated by commas. If the function call does not require any arguments an empty pair of parentheses must follow the function's name. The function call may appear by itself or it may be one of the operands within a more complex expression. The parameters in the body of the functions are called actual arguments as stated earlier, they may be expressed as constants, single variables or more complex expressions.
Let us consider another example of function.

```
#include <stdio.h>
main()
{
int a,b,c;
printf("Enter two numbers");
scanf("%d%d", &a,&b);
c=sum_v(a,b,);
printf("\n The sum of two variables is %d\n,",c);
}
sum_v(a,b)
int a,b
{
int d;
d=a+b;
return d;
}
```

This program returns the sum of two variables a and b to the calling program from where sum_v is executing. The sum is present in the variable c through the 'return d' statement. There may be several different calls to the same function from various places within a program. The actual parameters may differ from one function call to another. Within each function call, the actual arguments must correspond to the formal arguments in the function definition, i.e. the number of actual arguments must be same as the number of formal arguments and each actual argument must be of the same data type as its corresponding formal argument.

Let us consider an example.

```
#include <stdio.h>
main()
{
int a,b,c,d;
printf("\n Enter value of a=");
scanf("%d", &a);
printf("\n Enter value of b=");
scanf("%d",&b);
printf("\n Enter value of c=");
scanf("%d", &c);
d=maxi(a,b);
printf("\n maximum =%d", maxi(c,d));
}
maxi(x,y);
int x,y
{
int z;
z=(x>=y)? x:y;
return z;
}
```

The function maxi is accessed from two different places in main. In the first call actual arguments are a, b and in the second call c, d are the actual arguments.

If a function returns a non-integer quantity and the portion of the program containing the function call precedes the function definition, then there must be a function declaration in the calling portion of the program. The function declaration effectively informs the compiler that a function will be accessed before it is defined. A function declaration can be written as.

datatype function name ( );

Function calls can span several levels within a program; function A can call function B so on.

6 PASSING ARGUMENT TO A FUNCTION

Arguments can be passed to a function by two methods, they are called passing by value and passing by reference. When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change. This procedure for passing the value of an argument to a function is known as passing by value.

Let us consider an example

```
#include <stdio.h>
main()
{
int x=3;
printf("\n x=%d(from main, before calling the
function"),x);
change(x);
```

```
printf("\n\nx=%d(from main, after calling the
function)",x);
}
change(x)
int x;
{
x=x+3;
printf("\nx=%d(from the function, after being
modified)",x);
return;
}
```

The original value of x (i.e. x=3) is displayed when main begins execution. This value is then passed to the function change, where it is sum up by 3 and the new value displayed. This new value is the altered value of the formal argument that is displayed within the function. Finally, the value of x within main is again displayed, after control is transferred back to main from change.

```
x=3 (from main, before calling the function)
x=6 (from the function, after being modified)
x=3 (from main, after calling the function)
```

Passing an argument by value allows a single-valued actual argument to be written as an expression rather than being restricted to a single variable. But it prevents information from being transferred back to the calling portion of the program via arguments. Thus, passing by value is restricted to a one-way transfer of information. Arrays are passed differently than single-valued entities. If an array name is specified as an actual argument, the individual array elements are not copied to the function. Instead the location of the array is passed to the function. If an element of the array is accessed within the function, the access will refer to the location of that array element relative to the location of the first element. Thus,

any alteration to an array element within the function will carry over to the calling routine.

7 SPECIFICATION OF DATA TYPES OF ARGUMENTS

The calling portion of a program must contain a function declaration if a function returns a non-integer value and the function call precedes the function definition. Function declaration may be included in the calling portion of a program even if it is not necessary. It is possible to include the data types of the arguments within the function declaration. The compiler will then convert the value of each actual argument to the declared data type and then compare each actual data type with its corresponding formal argument. Compilation error will result if the data types do not agree. We had already been use, data types of the arguments within the function declaration. When the argument data types are specified in a function declaration, the general form of the function declaration can be written as

data-type function name (argument type1, argument type2, ... argument type n);

Where data-type is the data type of the quantity returned by the function, function name is the name of function, and argument type/refer to the data types of the first argument and so on. Argument data types can be omitted, even if situations require a function declaration.

Most C compilers support the use of the keyword void in function definitions, as a return data type indicating that the function does not return anything. Function declarations may also include void for the same purpose. In addition, void may appear in an argument list, in both function definitions and function declarations, to indicate that a function does not require arguments.

8 FUNCTION PROTOTYPES AND RECURSION

Many C compilers permits each of the argument data types within a function declaration to be followed by an argument name, that is data-type function name (type1 argument 1, type 2 argument2... type n argument n); Function declarations written in this form are called function prototypes.

Function prototypes are desirable, however, because they further facilitate error checking between the calls to a function and the corresponding function definition. Some of the function prototypes are given below:

int example (int, int); or int example (int a, int b);

void example 1(void); or void example 1(void);

void fun (char, long); or void fun (char c, long f);

The names of the arguments within the function declaration need not be declared elsewhere in the program, since these are "dummy" argument names recognized only within the declaration. "C" language also permits the useful feature of 'Recursion'.

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a precious result. In order to solve a problem recursively, two conditions must be satisfied. The problem must be written in a recursive form, and the problem statement must include a stopping condition. The best example of recursion is calculation of factorial of a integer quantity, in which the same procedure is repeating itself.

Let us consider the example of factorial:

```
#include <stdio.h>
main()
{
int number;
```

```
long int fact(int number);242 :: Computer Applications
printf("Enter number");
scanf("%d", & number);
printf("Factorial of number is % d\n", fact(number));
}
long int fact(int number)
{
if(number <=1)
return(1);
else
return(number *fact(number-1));
}
```

The point to be noted here is that the function 'fact' calls itself recursively, with an actual argument (n-1) that decrease in value for each successive call. The recursive calls terminate the value of the actual argument becomes equal to 1.

When a recursive program is executed, the recursive function calls are not executed immediately. Instead of it, they are placed on a stack until the condition that terminates the recursion is encountered. The function calls are then executed in reverse order, as they are popped off the stack.

The use of recursion is not necessarily the best way to approach a problem, even though the problem definition may be recursive in nature.

9 STORAGE CLASSES – AUTOMATIC, EXTERNAL, STATIC VARIABLES

There are four different storage-class specification in 'C', automatic, external, static and register. They are identified as auto, extern, static and register respectively.

Automatic variables are always declared within a function and are local to the function in which they are declared, that is their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another. The location of the variable declarations within the program determine the automatic storage class, the keyword auto is not required at the beginning of each variable declaration.

These variables can be assigned initial value by including appropriate expressions within the variable declarations. An automatic variable does not retain its value once control is transferred out of its defining function. It means any value assigned to an automatic variable within a function will be lost once the function is exited. The scope of an automatic variable can be smaller than an entire function. Automatic variables can be declared within a single compound statement.

External variables are not confined to single functions. Their scope extends from the point of definition through the remainder of the program. External variable are recognized globally, that means they are recognized throughout the program, they can be accessed from any function that falls within their scope. They retain their assigned values within their scope. Therefore, an external variable can be assigned a value within one function and this value can be used within another function. With the use of external variables one can transfer the information between functions.

External variable definitions and external variable declarations are not the same thing. An external variable definition is written in the same manner as an ordinary variable declaration. The storage-class specifier extern is not required in an external variable definition, because these variables will be identified by the location of their definition within the program. An external variable declaration must begin with the storage class specifier extern. The name of the exter-

nal variable and its data type must agree with the corresponding external variable definition that appears outside of the function. The declaration of external variables cannot include the assignment of initial values. External variables can be assigned initial values as a part of the variable definitions, but the initial values must be expressed as constants rather than as expression. These initial values will be assigned only once, at the beginning of the program. If an initial value is not included in the definition of an external variable, the variable will automatically be assigned a value of zero.

Static variables are defined within individual functions and therefore have a same scope as automatic variables, i.e. they are local to the functions in which they are defined. Static variables retain their values throughout the program. Thus, if a function is exited and re-entered later, the static variables defined within that function will retain their former values. Static variables are defined within a function in the same manner as automatic variables, but its declaration must begin with the static storage class designation. They cannot be accessed outside of their defining function. Initial values can be included in static variable declarations. The initial value must be expressed as constants, not expression, the initial values are assigned to their respective variables at the beginning of program,execution. The variables retain these values throughout the program, unless different values are assigned during the program. This is all for storage classes auto, extern and static.

Let us consider and example of static variables:

static int a;

If the keyword static is replaced with the keyword auto, the variable is declared to be of storage class auto. If a static local variable is assigned a value the first time the function is called, that value is still there when the function is called second time.

display_number()

```
{
static int number=2;
printf("number=%d\n", number);
number++;
}
```

When the first time display_number is called, it prints the value 2, to which number is initialized. Then number is incremented to 3, and terminates. The second time display_number is called, it prints the value of 3. On the third call, the value printed is 4 and so on. Point to be noted here is that the initialization is not performed after the first call. An initialization used in a declaration occurs only once-when the variable is allocated. Since a static variable is allocated only once, the initialization occurs only during the entire program, no matter how many times the function is called. When the display_number function in the example is called the second time, the value found in the variable number is the value left there by the previous call to the function.

10 WHAT YOU HAVE LEARNT

In this lesson you have learnt about functions, how to define and declare functions, what are the functions different data types. You are now able to execute function from different places of a program. You are now familiar with useful feature of functions say recursion. In this lesson we are also discussed different storage classes like auto, extern and static.