Chapter 6

User-Defined Functions

What is a function?

A function is a single comprehensive unit containing blocks of statements that performs a specific task. The specific task is repeated each time function is called. So this avoid the need of rewriting the same code again and again. Every program must contain one function named main() where the program always begin execution. When a function is called, the program execution is shifted to the first statement in the called function. After the function returns to the calling function, values can be returned and that value can be used as an operand in an expression.

Why to use function?

- To avoid rewriting of the same code again and again
- Separating the code into modular functions also make the program easier to design, understand and debug

So it is not good to write an entire program into one function. Instead, break a program into small units and write functions for each of these isolated subdivisions.

Defining a function

A function definition has name, a parenthesis pair containing one or more parameters and a body. For each parameter, there should be a corresponding declaration that occurs before the body. The general format of the function definition is given as follows:

return_type **function_name**(*datatype* **argument1**, *datatype* **argument2**, …) {

```
  statement1;
     ;
     ;
  return ;
}
```

Category of the function

The function may return varying type of value to the calling portion of the program. Any of the datatype such as int, float, char, etc. can be in the function declaration. If a function is not supposed to return value, it can be declared as type void.

Example of i)

void **myfunction**(*void*){

```
     ;
     ;
}
```

Example of ii)

```
void  myfunction(float  a, char b, int c){
    ;
    ;
}
```

Example of iii)

```
int myfunction(void){
    ;
    ;
}
```

Example of iv)

```
float myfunction(float  a, char b, int c){
    ;
    ;
}
```

Function Definition

A function definition is a unit of a function itself with the type of value it returns and its parameters declared and the statements specified in it, that are executed when the function is called.

Actual Arguments & Formal Arguments

Any variable declared in the body of a function is said to be local to that function. If the variables are not declared either or arguments or inside function body are considered "global" to the function and must be defined externally. *Arguments defined inside the function are called formal arguments* where as the *argument from which the arguments have been passed to a function is known as actual arguments*.

Example#1:

```
#include <stdio.h>
void line(void);  //function prototype or function declaration
void line(void)   //function definition
{
   int j;
   for(j=1;j<=42;j++){
      printf("*");
   }
}
```

```
void main(void){
   clrscr();
   printf("Welcome to the world of  C programming \n)";
   line();    //function call
}
```

return statement

The return() statement has two purposes. First, executing it immediately transfers control from the function back to the calling program. And second, whatever is inside the parentheses following return is returned as a value to the calling program. The return statement may or may not include an expression. If return contains no parameters then this return is used to transfer the control used only with void else if return has got argument having bracket or without bracket then this return is returning value to the calling portion of the program. There is no restriction on the number of return statements that may be present in a function. Also the return statement need not be always be present at the end of the called function.

Note: *Using a return statement, only one value can be returned by a function*

Syntax:

```
return ;
return (expression);
```

Example#2: A program to return value from a function.

```
#include<stdio.h>
#include<conio.h>
float max_value(float,float);  //function prototype or function declaration
void main(void){
  float x,y,max; printf("Enter the two numbers");
  printf("“%d %d”,&x,&y);
  max=max_value(x,y);
  printf("The maximum value is  %d",max); getch();
}
float max_value(float a,float b){
  if(a>b){
    return a;
  }else{
    return b;
  }
}
```

User Defined Functions

There are basically two types of c functions - library functions and user-defined functions. Library functions are predefined in the header files and are not required to be written by the user at the time of writing by the user at the time of writing a program. Example of some library functions are

getch(), getche();

User defined functions may be classified in the following three ways based on formal arguments passed and the usage of the return statement.

a) A function is invoked without passing any formal arguments from the calling portion of a program and doesn't return any value back to the calling function.

```
#include<stdio.h>
#include<conio.h>
void main(){
   void display(void);  //function declaration or function prototype
   display();
   getch();
}
void display(void)  //Function definition
{
  int x,y,sum;
  printf("Enter the value of x and y:");
  scanf("%d%d",&x,%y);
 sum=x+y;
 printf("Sum=%d",sum);
}
```

b) A function is invoked with formal arguments from the calling portion of a program but the function doesn't return any value back to the program.

```
#include<stdio.h>
#include<conio.h>
void square(int);
void main(){
  int max;
  printf("Enter the value for n:");
  scanf("%d",&max);
  for(int i=1;i<=max;++i)
```

```
    square(i);
  getch();
}
void square(int n){
  int value;
  value=n*n;
  printf("Square of i is %d\n",value);
}
```

c) Function is invoked with formal arguments from the calling portion of a program which returns a value back to the calling environment.

//Program to calculate factorial using function

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main(void){
  int x,n;
  printf("enter the number");
  scanf("%d",&n);
  x=fact(n);
  printf("The factorial is %d",&x);
  getch();
}
int fact(int n){
  int i,value=1;
  if(n==1){
    return value;
  }else{
    for(i=1;i<=n;i++){
      value=value*i;
    }
    return value;
  } //else ends here
} //function fact ends here
```

//Program to add two digits using function

```
#include<stdio.h>
#include<conio.h>
float add(float,float);
void main(){
 float x,y,sum; printf("Please enter value for x and y:\n");
scanf("%f%f",&x,&y);
sum=add(x,y);
printf("The Sum of %fand %f=%f",x,y,sum); getch()
}
float add(float c, float d){
  float result; result=c+d;
   return result;
}
```

Rules that must be remembered while defining function

- A function prototype declaration must appear at the beginning of the program following the #include statement. In a function prototype the type of the function and the type of each formal argument must be specified. The variable names used for formal arguments in a function prototypes are arbitrary; they may even be blanks. Each function prototype must be terminated by a semicolon.
- A function name may be any valid identifier. The type of the function specifies the type of the quantity which will be returned to the calling program.
- The formal arguments used in defining a function may be scalars or arrays. Each formal arguments type must be specified. For example **int add(float a,b,c)** is invalid and should be in the form **int add(float a,float b,float c)**
- Within a function definition other variables may be declared and used. Such variables are local to the function and are not known outside the body of the function.
- Within a function definition one cannot define another function.

Rules that must be remembered while calling a function

- The actual arguments in the calling function must agree in number, order and type with the formal arguments in the function declaration.
- Copies of the values of actual arguments are sent to the formal arguments and the function is computed. Thus the actual arguments remain unchanged.
- If the formal arguments is an array name the corresponding actual arguments in the calling function must also be an array name of the same type.
- A return statement in the body of the function returns a value to the calling function. The type of expression used in return statement must match with the type of the function declared. There may be functions without return statement. In such a case the function must be declared as void.
- A function can return only one value.

Recursion or Recursive Function

A function which calls itself directly or indirectly again and again is known as recursive function. Recursive functions are useful while constructing the data structures like linked list etc. There is distinct difference between normal and recursive functions. A normal function will be invoked by the main function whenever a function call is made, whereas the recursive function will be invoked itself directly or indirectly as long as the given condition is satisfied.

Recursion is the process of defining something in terms of itself, and is sometimes called circular definition. Recursion has many negatives:

- It repeatedly invokes the mechanism, and consequently increases the overhead of function calls.
- This repetition can be expensive in terms of both processor time and memory space. Each recursive call causes another copy of the function, these set of copies can consume considerable memory space.

Example:

```
#include<stdio.h>
#include<conio.h>
long fact(int);
void main(){
  int n;
  long y;
  clrscr();
  printf("Enter the factorial number");
  scanf("%d",&n);
  y=fact(n);
  printf("The factorial number is %d and the factorial of the given number is %ld",n,y);
  getch();
}
long fact(int z){
   long value;
   if(z==1){
    return 1;
   }else{
    value= z*fact(z-1); //function recursion occurs here.
    }
   return value;
}
```

Example: Recursive function to calculate Power

```
#include<stdio.h>
#include<conio.h>
double power(float val, unsigned pow);
void main(){
 int n;
 float x;
 printf("\nX=");
 scanf("%f",&x);
 printf("\nN=");
 scanf("%d",&n);
 printf("\nPower=%f",power(x,n));
}
double power(float val,unsigned pow){
if(pow==0){
     return 1.0;
}else{
   return(power(val,pow-1)*val);
}
}
```

Example: Recursive function to calculate the nth term of the fibonnacci series

```
#include<stdio.h>
#include<conio.h>
int fibo(int);
void main(){
int x,y;
scanf("%d",&x);
 y=fibo(x);
printf("%d”,y);
getch();
}
```

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

Recursion vs. Iteration

1. Both Iteration and recursion are based on control statements: Iteration uses a repetition statement (such as for, while, do-while); Recursion uses a selection statement such as (if, if-else, or switch).
2. Both Iteration and Recusrsion involves repetition: Iteration explicitly uses a repetition statement; Recursion achieves repetition through repeated function calls.
3. Both Iteration and Recursion each involve a termination test: Iteration terminates when the loop continuation condition fails; Recursion terminates when a base case is recognized.
4. Both Iteration and Recursion can occur infinitely: An infinite loop occurs in the iteration if the loop continuation test never becomes false(e.g. for(i=1;i>0;i++)); Infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.

Call By Value & Call By Reference

Arguments can be passed in function in one of the following two ways

- Passing by value(Sending the values of the arguments)
- Passing by reference(Sending the address of the arguments)

When arguments are passed by values the "value" of each actual argument in the calling function is copied into corresponding formal argument of the called function. With this method, changes made to the formal arguments in the called function have no effect on the values of the actual arguments in the calling function. As we noted the function doesn't have to access the original variable in the function definition. In fact, this provides security such that the function cannot harm the original variable.

Example: Program illustrating Call by value

```
#include<stdio.h>
#include<conio.h>
void swap(int ,int);
void main(){
   int m=5;
   int n=10;
   clrscr();
   swap(m,n);
   printf("m=%d\tn= %d",m,n);
   getch();
}
void swap(int a,int b){
   int t;
   t=a;
   a=b;
   b=t;
  printf("a=%d\tb=%d",a,b);
}
```

Output:

```
a=10  b=5     m=5    n=10
```

In the above case when the function is called, it gets a copy of the arguments ("Call By Value"). The function cannot affect the value that was passed to it, only its own copy.

If it is necessary to change the original values, the addresses of the arguments can be passed as shown below, which is called **Call By Reference**.

Passing arguments by reference uses a different mechanism. Instead of a value being passed to a function, a reference of the original variable, in the function definition is passed. In this case the address of actual arguments in the calling function are copied into formal arguments of the called function

The primary advantage of passing by reference is that the function can access the actual variables in the calling program. The other benefit is that it provides a mechanism for returning more than one value from the function back to the calling program.

Example: Program illustrating Call By Reference

```
#include<stdio.h>
#include<conio.h>
void swap(int *,int *);
void main(){
   int m=5;
   int n=10;
   clrscr();
   swap(&m,&n);
   printf("The value of m and n after swapping is %d and %d",m,n);
   getch();
}
void swap(int *a,int *b){
   int t;
   t=*a;
   *a=*b;
   *b=t;
}
```

Macros

We have already seen that the #define statement can be used to define symbolic constants within a program. At the beginning of a compilation process, all symbolic constants are replaced by their equivalent text. Thus, symbolic constants provide a form of shorthand notation that can simplify the organization of a program.

The #define statement can be used to define macros; i.e., single identifiers that are equivalent to expressions, complete statements or group of statements.

Example:

```
#include<stdio.h>
#include<conio.h>
#define area length*breadth
main(){
  int length,breadth;
  printf("Enter Length and Breadth:\n"); scanf("%d%d",&length, &breadth);
  printf("Area=%d",area); getch();
}
```

Example: Macro with argument

```
#include<stdio.h>
#include<conio.h>
#define area(x) (3.14*x*x)
main(){
 float r1=6.25,r2=2.5,a; a=area(r1);
 printf("\n Area of Circle=%f",a); a=area(r2);
printf("\nArea of Circle=%f",a); getch();
}
```

Though macro calls are like function calls, they are not really the same thing. In a macro call the preprocessor replaces the macro template with its macro expansion. Whereas, in a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function. Usually macros make the program run faster but increases the program size, whereas the function make the program smaller and compact.

If we use macro hundred times in a program, the macro expansion goes into our source code at hundred different places, thus increasing the program size. On the other hand, if a function is used, then even if it is called from hundred different places in the program, it would take the same amount of space in the program. So, if the macro is simple and sweet like in our examples, it makes a nice shorthand and avoids the overheads associated with function calls. On the other hand, if we have a fairly large macro and it is used fairly often, replace macro with a function.

Scope, Visibility & Lifetime of Variables

There are two basic types of scope: *local scope* and *global scope*. A variable declared outside all functions is located into the **global scope**. Access to such variables can be done from anywhere in the program. These variables are located in the global pool of memory, so their lifetime coincides with the lifetime of the program. A variable declared inside a block (part of code enclosed in curly brackets) belongs to the **local scope**. Such a variable is not visible (and therefore not available) outside the block, in which it is declared.

In C not only do all variables have a data type, they also have **storage class**. The following variable storage classes are most relevant to functions:

1. Automatic Variables,
2. External Variables,
3. Static Variables,
4. Register Variables

Automatic Variables

Automatic variable is a local variable which is allocated and de-allocated automatically when program flow enters and leaves the variable's scope. The keyword **auto** is used to declare automatic variables explicitly.

```
main(){
        auto int number;
        …
}
```

External Variables

It is possible to define variables that are external to all functions, that is, variables that can be accessed by name by any function. External variables are both ***active*** and ***alive*** throughout the entire program so these are also known as ***global variables***.

```
int global_variable;
void main(void) {
  global_variable = 1;
  // Statements
}
```

Static Variables

Static variables have a property of preserving their value even after they are out of their scope. Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.

Syntax:

```
static data_type var_name = var_value;
```

Example:

```
#include<stdio.h>
#include<conio.h>
int fun(){
  static int count = 0;
  count++;
  return count;
}

void main(){
  clrscr();
  printf("%d ", fun());
  printf("%d ", fun());
  getch();
}
```

Register Variables

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using register keyword. Register variables, are a special case of automatic variables, are kept by **Compiler** in one of the **machine's registers** *instead of* keeping **in the memory** for faster execution.

Syntax:

```
register int count = 0;
```