Topic: User-defined Functions in C

++
++
+++++++

Objectives:

- To know what is Function and what is the need of function.
- To know general form of Function Definition.
- To know about Function Prototype and its importance.
- To learn how to write user defined functions with input parameter.
- To know use of void in function.
- To know about actual and formal parameters of function.
- To know about different forms of return statement and their uses.
- To learn about **recursive functions**.

What is Function? :

Functions are the building blocks of any C program. In fact, a C program is a collection of functions e . g main ( ) , scanf( ), printf( ), sqrt(x), pow(x,y)…… sin(x), cos(x) etc. Functions are two types one is User defined and another is Standard Functions.

Why is the need of function? :

- Dividing a large program into functions improves the understanding of the problem.

- Makes programs easy to correct (debug) and easy to maintain
- A function can be executed (called) from several locations in a program.
- It is not necessary to know the internal code of a function in order to be able to use it. For example, we do not know the code for the function printf(), but we know how to use it. Reuse of function subprograms.

Function definition (writing the function code):

The general form for function definition is:

```
function_return_type   function name  (parameter list separated by commas )
```

Function type (or function return type) is the type of data item that is returned to the caller, such as in , double , …etc.

Function name: (or function identifier) is the name of the function. The same rules for the variable names (identifiers) are applied to function names. Functions are identified and are called (referred to) by their names.

Parameter list: specifies the type of data items passed to the function. The data types are placed between parentheses, and if there is more than one item, they are separated by commas.

Function prototype (declaring a function):

Like variables, functions must be declared before they are used. The function prototype serves as a function declaration. The general form for function declarations (function prototype is):

double large(double x1, double x2, char y);

For Example:

int sum (int , int); declares a function called sum with a parameter list that consists of two int type data items. That is sum() expects two int type data items when it is called, and returns a int type data item to the caller. Specifying a name for each of the items in the parameter list is optional, but **highly recommended.**

e . g int sum (int x, int y);

Function with Input Arguments:

Till now we were using only one function, the main function in our programs. Programs can also be written where there can be *one or more function subprograms other than the main function*. The program should always have a main function whether it has function subprograms or not. Without the main function the program will not do anything as the program always starts from the main function.

Functions can be of following different types:

- *Functions returning no value and accepting no value.*
- *Function returning no value but accepting one or more values.*
- *Function returning one value or accepting one or more values.*

Let **abc** be a function name:

Then function that does not return any value and accept any value can be declared as:

void abc (void)

Function that accepts one or more value and returns no value can be declared as :

void abc (int x, float y, double d, int a)

Function that returns one value and accepts one or more values can be written as :

int abc (int b, double k, char p)

*Here the function is returning an **integer** value.*

float abc (int g, char h, double r)

*Here the function is returning a **float** value.*

Input parameters/Arguments in any function are those arguments which are declared as ordinary variable and which are used to supply some input to function and Output Parameter/Output Argument in any Function are special variables which are declared as pointer and which are used to carry/return more than one outputs from the function .

Function using only input parameter/argument can return only single value at a time while by using output parameters in any function that function can return more than one value (results) at a time.

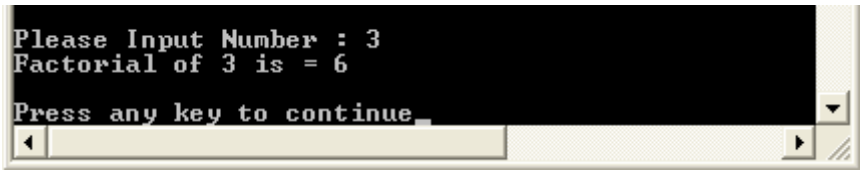
Example#1:

Write user-defined function to find factorial of a given number?

```
#include<stdio.h>
int fact (int n)  // user-defined function
{
        int i,f=1;
        for(i=1;i<=n;i++)
        {
        f=f*i;
        }
        return f;
```

```
}// end of user-defined function


void main( )  // calling main program
{
        int number,factorial;
        printf("Please Input Number : ");
        scanf("%d",&number);
        factorial=fact(number);  // function call
        printf("Factorial of %d is = %d\n\n",number,factorial);
}
```

Sample Output:



Another way of writing above user-defined function:

```
 #include<stdio.h>
void fact (int n)  // user-defined function
{
        int i,f=1;
        for(i=1;i<=n;i++)
        {
        f=f*i;
        }
   printf("Factorial of %d is = %d\n\n",n,f);
}// end of user-defined function


void main( )
```

```
{
        int number,factorial;
        printf("++++++++++++++++++++++++++++++++++++++++++++++++++\n\n");
        printf("Please Input Number : ");
        scanf("%d",&number);
        fact(number); // function call
}
```

Sample Output:

Example#2:

Write user-defined C-Function to find maximum of three numbers?

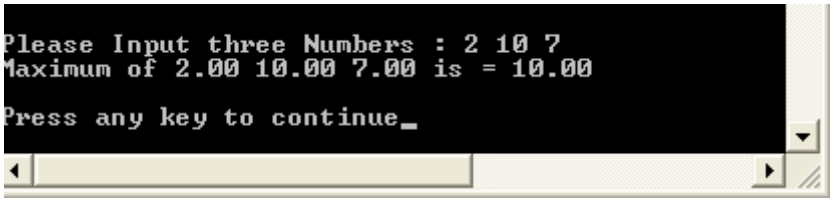
```
#include<stdio.h>
double max (double a, double b, double c) // user-defined function
{
        double big;
        if(a>b && a>c)
                big=a;
        else if(b>a && b>c)
                big=b;
        else
                big=c;
        return big;
} // end of user defined function

void main()  // calling main program
```

```
{
        double x,y,z,maximum;
        printf("+++++++++++++++++++++++++++++++++++++++++\n\n");
        printf("C Programming \n\n");
        printf("+++++++++++++++++++++++++++++++++++++++++\n\n");
        printf("Please Input three Numbers : ");
        scanf("%lf %lf %lf",&x,&y,&z);
        maximum=max(x,y,z); // function call
        printf("Maximum of %0.2lf %0.2lf %0.2lf is = %0.2lf\n\n",x,y,z,maximum);
}
```

Another way of writing above user-defined function

```
 #include<stdio.h>
void max (double a, double b, double c) // user-defined function
{
        double big;
        if(a>b && a>c)
                big=a;
        else if(b>a && b>c)
                big=b;
        else
                big=c;
   printf("Maximum of %0.2lf %0.2lf %0.2lf is = %0.2lf\n\n",a,b,c,big);
} // end of user defined function

void main( ) //calling main program
{
        double x,y,z;
        printf("Please Input three Numbers : "); scanf("%lf %lf
        %lf",&x,&y,&z); max(x,y,z);  // function call
}
```

Sample Output:



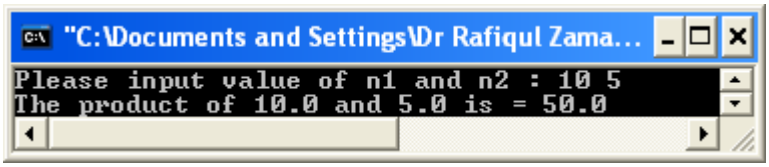
Example#3:

Write single user-defined function to find product and division of two numbers?

```
 #include<stdio.h>
float mul(float x, float y) // user defined function mul
{
float p;
p=x*y;
return(p);
} // end of mul function

float division(float number1, float number2) // user defined function division
{
float div;
div=number1/number2;
return div;
} // end of division function

void main( ) // calling function
```

```
{
float n1, n2, product, d;
printf("Please input value of n1 and n2 : ", &n1, &n2);
scanf("%f %f", &n1, &n2);

product = mul(n1, n2); // function call
d = division(n1, n2);  // function call

printf("The product of %0.1f and %0.1f is = %0.1f\n", n1, n2, product);
printf("The division of %0.1f and %0.1f is = %0.1f\n", n1, n2, d);
return;
} // end of main
```

Sample Output:



Argument List Correspondence Rules:

The number of actual arguments/parameter, the order of actual argument and type of actual arguments used in a call to a function must be the same as the number of formal parameters, the order of formal parameters and type of formal parameters/arguments listed in the function definition.

Use of return Statement in User-defined Function:

An return statement can take one of the following form:

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

The first , the „plain" return does not return any value, just it returns control to calling function.

e.g

```
if(error)
    return;
```

The second form of return with an expression returns the value of the expression.

e.g

```
int mul(int x, int y)
  {
    int p;
    p=x*y;
    return(p);
    }
```

In this above last two statements can be combined into one statement as follows :

```
return (x*y);
```

Can a function have more than one return Statements? :

Yes.

A function may have more than one return statements . This situation arises when the value returned is based on certain conditions:

e.g

```
if(x<=0)
    return(0);
else
    return(1);
```

Note: All functions by default returns int type data. But we can make it any type.

Recursive Functions:

What is Recursive Function? :

Recursive functions are the functions which call themselves repeatedly until some condition is met and then function stops calling and returns to the main program.

A recursive function has two parts :

(1). Base case (stopping condition)

(2). General case (recursive step: which must always get closer to base case from one invocation to another.)

e.g

1 for n=0 (base case)

n!= {

n*(n-1)! for n>0 (general case)

A recursive solution always needs a stopping condition to prevent an infinite loop. And we achieve it by using base case.

e.g

```
power(int x, int y)
  {
   if(y==0) return 1; /* base case */
   else
     return (x*power(x,y-1));  /*general case */
   }
```

Any problem which we can solve using recursion ,we can also solve that problem using iteration .

Generally, a recursive solution is slightly less efficient, in terms of computer time, than an iterative one because of the overhead for the extra function calls. In many instances, however,, recursion enables us to specify a natural, simple solution to a problem that otherwise would be difficult to solve. For this reason, recursion is an important and powerful tool in problem solving and programming. Recursion is used widely in solving problems. Those are not numeric, such as proving mathematical theorems, writing compilers, and in searching and sorting algorithms.

In iterative method we use for, while, do-while loops for achieving iteration for problem solving.

In recursive method of problem solving we replace for, while, do-while statement by if statement that selects between the recursive case and the base case (i.e terminating condition).

In recursion successive function call values are stored in stack and accessed in LIFO.

Examples:

Example#1:

/* compute n! using a recursive definition */	**/* iterative solutions computes n! for n is >=0 */**

int factorial (int n)
{
if (n == 0)
return 1;
else
return(n * factorial (n-1));
}
	int factorial (int n)
	{
	int i, product=1;
	for (i=n; i>1; --i)
	product=product *i;
	return (product);
	}

Example#2:

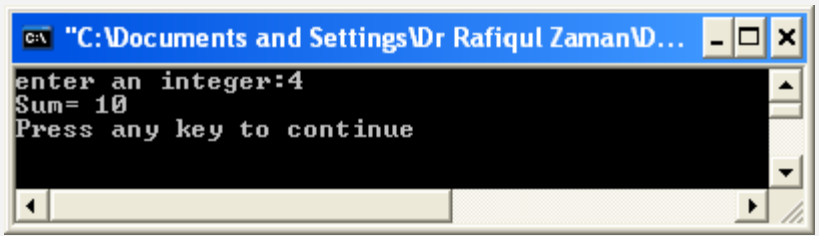
Recursive Program:

// Write recursive function to find sum of first n natural numbers:

```
#include <stdio.h>
int main( ) // calling program
 {
 int sum(int n);
 int num, s;
printf("enter an integer:");
```

```
scanf("%d",&num);
s=sum(num);
printf("%d",s);
return 0;
 }

int sum(int n)  // recursive function
  {
    if (n <= 1)
    return    n;
    else
    return n + sum(n-1);
    }
```

Sample output:



How above recursive function works inside computer? Please see below:

Function call	n	n+ sum(n-1)	Return value
sum(4)	**4**	**4 + ~~sum(3)~~** **4 + 6**	**4+ 6=10**
sum(3)	**3**	**3 + ~~sum(2)~~** **3 + 3**	**3+3=6**
sum(2)	**2**	**2 + ~~sum(1)~~** **2 + 1**	**2+1=3**
sum(1)	**1**	**1 + ~~sum(0)~~** **1 + 0**	**1+0=1**

The Iterative Program.

// Write iterative function to find sum of first n natural numbers:

```
#include <stdio.h>
int main()    // calling main program
 {
```

```
 int sum(int n);
 int num,s;
printf("enter an integer:");
scanf("%d",&num);
s=sum(num);
printf("%d",s);
return 0;
}

int sum( int n)  //User defined function
{
 int i, total=0;
for (i=1; i<=n ; i++)
total +=i;
return total;
}
```

Example#3:

Power function:

base ^{exp}. One way to do the calculation is:
base ^{exp}= base * base * base * … * base

For exp repetitions. A more compact way is:

base ^{exp} = base * base $^{exp-1}$

In this case the stopping condition (the base case) occurs when exp=0. Since any number raised to the 0 power is 1, we have the following situation:

$$\text{base}^{exp} = \begin{cases} 1 & \text{for exp=0 (base case)} \\ \text{base} * \text{base}^{exp-1} & \text{for exp>0 (General Recursive case).} \end{cases}$$

Implementation of Power Function:

Recursive	*Iterative*

int power(int base, int exp)
{
if (exp = = 0)
return (1);
else
return (base * power(bae, exp –1));
}
	int power_it(int base, int exp)
	{
	int count, product;
	product=base;
	for (count = exp-1; count >0; count --)
	product = product * base;
	return (product);
	} **OR**

	int power(int base, int exp)
	{ int p=1;
	while(exp !=0)
	{ p *= base;
	exp --;
	}
	return p;
	}

Example#4:

Fibonacci sequence:

The Fibonacci numbers are a sequence of numbers that have many varied uses
the Fibonacci sequence 0, 1, 1,2,3,5,8,13,21,34

Fibonacci number Begins with the term 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms)

- **Fibonacci(0) is 0**
- **Fibonacci (1) is 1**
- Fibonacci n is Fibonacci n-2 + Fibonacci n-1, for n >2

This is defined recursively as:

$f_0 = 0$, $f_1 = 1$ $f_{i+1} = f_i + f_{i-1}$ for i=1,2, ..
except for f0 and f1, every element is the sum of its previous two elements:
0, 1, 1, 3, 5, . . .

The following function computes the nth Fibonacci number.

Iterative Solution	Recursive Solution

int fibonacci (int n)
{ int i,sum1=0, sum2=1, sum;
if (n<=1)
return n;
else
{ for (i=2; i<=n; i++)
{ sum=sum1+sum2;
sum1=sum2;
sum2=sum;
}
return sum;
}
}
	int fibonacci(int n)
	{ if (n<=1)
	return n;
	else
	return (fibonacci(n-1)
	+ fibonacci(n-2));
	}

Example#5:

Reversing string:

Reversing string.

An interesting feature of recursion is its ability to easily reverse a process. The following program reads a string from the user and prints it backward.

```
#include <stdio.h>
void reverse_str(void);
main( )
{ printf(“input a line: “);
    reverse_str();
    printf(“\n\n”);
    return 0;
}

void reverse_str(void)
{ char ch;
    scanf(“%c”, &ch);
    if (ch != „\n‟)
      reverse_str();

   printf(“%c”,ch);
}
```

Notice that in the function reverse_str, if the character read is not the new line character, the function is invoked again.
Each call has its own local storage for the variable ch. The function calls are stacked by the system until the new line character is read. Only after the new line character is read that printing begins – starting with the last character entered. Recursion relies on an internal (usually hardware) runt-time stack to store the information for keeping track of the recursive calls.

What happens if the recursive call is placed after printf as shown next?

```
void reverse_str(void)
{ char ch;
    scanf(“%c”, &ch);
    if (ch != „\n‟)
        printf(“%c”,ch);
      reverse_str();
   }
```

In recursive function call Dynamic memory allocation occurs at run time. With each function call, an activation record contains the state of a given function, that is, the address of the current instruction and the values of all the local variables, each time a

function calls itself; an activation record is created and pushed on the run-time stack. The first call thus appears at the bottom of the sack. The last call (the base case) appears at the top of the stack. At that time the top activation record is popped from the stack and the function returns its first value. The process continues until all activation records are popped from the stack, at which time the recursion stops.

Exercises

Problem # 1:

Write a program that reads 2 integers **start** and **end i**n the main program. It passes these 2 integers to a function subprogram that uses for loop to find the sum of all the integers between start and end and returns the sum to the main program. The main program should print the sum.

Problem # 2:

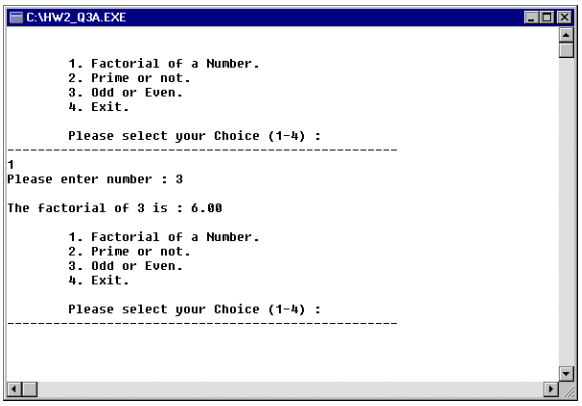
Write a program that reads 3 integers **start, end** and **number** in the main program. It passes these 3 integers to a function subprogram that uses for loop to find those integers between start and end that are divisible by number. If it finds the numbers it prints them in the function only. Here the function is *not returning* anything. Divisible means giving a remainder of 0.

Problem # 3:

Write a menu driven program which has following options :

1. Factorial of a Number.
2. Prime or not.
3. Odd or Even.
4. Exit

It must execute appropriate choice entered by user.
Your output must be in the following format:



Problem # 4:

Write user-defined function to convert a volume in milliliters to fluid ounces.

Write main program to call this function.

Relevant Formula:

fluid_ounces=0.034 (milliliters).

Problem # 5:

Write user-defined function to take a depth (in kilometers) inside the earth as input data; compute and print the temperature at this depth in degree Celsius and degree Fahrenheit. Write main program to call this function.

Relevant Formulas:

Celsius=10(depth)+20.
Fahrenheit=1.8 (Celsius)+32.

Problem # 6:

Write C Program to calculate the net pay of an employee based on values for the wage rate and hours worked. These values are passed as **arguments** to function called **calc_net_pay**. The function then computes the net pay and **returns** the computed value to **main**.

Problem # 7:

Write a program using **for** loop that reads the value of the variable **num** 10 times. It should add 25 to **num** and print the result. If the value of **num** is greater than 50 and less than 100 it should **exit** the loop. If the value of **num** is less than 10 **or** more than 1000 it should not add 25 to the **num**. Use **break** and **continue**.