**Functions:**

A function in C is a small “sub-program” that performs a particular task, and thus supports the concept of modular programming design techniques. In modular programming the various tasks that your entire program must accomplish are assigned to individual functions and the main program basically calls these functions in a certain order.

The main body of a C program, identified by the keyword main, and enclosed by left and right braces is a function. It is called by the operating system when the program run, and when terminated, returns to the operating system. We have also seen examples of library functions which can be used for I/O, and character/string handling.

**There are two types of functions.**

**Pre/System Defined functions,**

**User Defined functions.**

**Pre/System Defined Functions:**

The Functions we use like printf, scanf, and gets, etc are the predefined functions used from the library.

**User Defined functions:**

These Factions defined by the programmer to perform a task. main is also a User Defined function.

In order to use their own functions, the programmer must do three things (not necessarily in this order):

**– Define the function**

**– Declare the function**

**– Use the function in the main code.**

Function Definition:

syntax:

return\_type function\_name(function arguments)

{

/\* Declaration part;

Executable part;

\*/

return (expression);

}

**return\_type:**

it is data type which indicates the type of the returning value(default is int). Some functions will not actually return a value or need any arguments. For these functions the keyword void is used.

**function\_name:**

It is a valid function name to identify it uniquly.

**Function arguments:**

Tells what arguments the function needs when it is called (and what their types are). if you need any values to perform a task, to represent those values we use these declarations.

return Statement:

A function returns a value to the calling program with the use of the keyword return, followed by a data variable or constant value. The return statement can even contain an expression. Some examples

return 3;

return n;

return ++a;

return (a\*b);

• When a return is encountered the following events occur:

1) Execution of the function is terminated and control is passed back to the calling program,

2) The function reference evaluates to the value of the return expression.

• If there is no return statement control is passed back when the closing brace of the function is encountered (“falling off the end”).

**Function arguments are two types:**

Actual arguments: The arguments that are passed to the function at the time of calling.

Formal arguments: The arguments that are defined to store actual arguments are called as formal arguments/alias arguments.

Formal Argument declarations can be done as follows,

(data\_type var1,data\_type var2,...)

Function definition example:

int addition(int a1,int b1)

{

int result;

result = a1 + b1;

return result;

}

main()

{

int a,b,result;

scanf("%d%d",&a,&b);

result = addition(a,b);

printf("%d is the addition",result);

}

**Functions can be declared as variables:**

Function prototypes are used to declare a function so that it can be used in a program before the function is actually defined.

A function prototype is simply the function header from the function definition with a semi-colon attached to the end. The prototype tells the compiler the number and type of the arguments to the function and the type of the return value. Function prototypes should be placed before the start of the main program.

Ex:

int addition(int a, int b);

function calls:

function calls are used to call the

function with actual parameters.

main()

{

int a1=10,b1=10;

int add;

add = addition(a1,b1);

/\* this is the function call \*/

}

int addition(int a1,int b1)

{

contents...

}

To invoke a function, just type its name in your program and be sure to supply actual arguments (if necessary). A statement using our addition function it would look like

add = addition(a1, b1);

When your program encounters a function reference, control passes to the function. When the function is completed, control passes back to the main program. In addition, if a value was returned, the function reference evaluates to that return value.

**Functions types:**

**1. Pass/Call by value**

This method copies the value of an argument into the formal parameter of the functions. In this case, changes made to the parameter have no effect on the argument.

Based on Arguments and return types,

a. Function with return type and with parameters

eg:

int addition(int a1,int b1)

{

return...;

}

b. Function with return type and without parameters

eg:

int addition()

{

return...;

}

c. Function without return type and with parameters

eg:

void addition(int a1,int b1)

{

}

d. Function without return type and without parameters

eg:

void addition(void)

{

}

**2. Pass/Call by reference**

In this method, the address of an argument is copied into the parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

You can create a call by reference by passing a pointer to an argument, instead of passing the argument itself. Since the address of the argument is passed to the function, code within the function can change the value of the argument outside the function.

Pointers are passed to functions just like any other argument. Of course, you need to declare the parameters as pointer types.

**Recursion:**

The Process of calling the function by itself. Recursion is the process in which a function repeatedly calls itself to perform calculations. Typical applications are in mathematics, sorting trees and lists.

main()

{

addition();

}

addition()

{

addition();

}

factorial(5) = 5\*4\*3\*2\*1

5! = 120

0! = 1

int factorial(int n)

{

int result;

if (n<=1)

result=1;

else

result = n \* factorial(n-1);

return result;

}

**Storage classes:**

Every variable in C actually has two attributes: its data type and its storage class. The storage class refers to the manner in which memory is allocated for the variable. The storage class also determines the scope of the variable, that is, what parts of a program the variable’s name has meaning. In C, the four possible Storage classes are

**– auto**

**– extern**

**– static**

**– register**

- auto:

• This is the default classification for all variables declared within a function body [including main()] .

• Automatic variables are truly local.

• They exist and their names have meaning only while the function is being executed.

• They are unknown to other functions.

• When the function is exited, the values of automatic variables are not retained.

• They are normally implemented on a stack.

• They are recreated each time the function is called.

- extern:

In contrast, extern variables are global.

• If a variable is declared at the beginning of a program outside all functions [including main()] it is classified as an external by default.

• External variables can be accessed and changed by any function in the program.

• Their storage is in permanent memory, and thus never disappears or need to be recreated.

- static:

• A static variable is a local variable that retains its latest value when a function is recalled. Its scope is still local in that it will only be recognized in its own function. Basically, static variables are created and initialized once during the first reference to the function.

- register:

• It takes to fetch a variable from memory and store its value in a register where the CPU can perform some calculation with it. So for performance reasons, it is sometimes advantageous to store variables directly in registers. This strategy is most often used with loop counter variables, as shown below.

register int i;

for (i=0; i<n; ++i)

...

.

.

**/\* program to demonstrate function with return type and with parameters \*/**

int square(int a)

{

return a\*a;

}

int cube(int b)

{

return b\*b\*b;

}

main()

{

int a1,b1,result;

clrscr();

printf("Enter a value to find the Square: ");

scanf("%d",&a1);

result = square(a1);

printf("\nThe Square of %d is : %d",a1,result);

printf("\nEnter a value to find the cube: ");

scanf("%d",&b1);

result = cube(b1);

printf("\nThe Cube of %d is : %d",b1,result);

getch();

}

**/\* program to demonstrate function with return type and without parameters\*/**

int square(void)

{

int a;

printf("Enter a value to find the square: ");

scanf("%d",&a);

return a\*a; /\* We are retruning the square \*/

}

int cube(void)

{

int b;

printf("\nEnter a value to find the cube: ");

scanf("%d",&b);

return b\*b\*b; /\* Here we are about to return cube of a value \*/

}

main()

{

int result;

clrscr();

result = square();

printf("The Square is: %d",result);

result = cube();

printf("The Cube is: %d",result);

getch();

}

**/\* program to demonstrate the functions without return type with parameters \*/**

void square(int a)

{

clrscr();

printf("%d is the square of %d...",a\*a,a);

getch();

}

void cube(int b)

{

clrscr();

printf("%d is the cube of %d...",b\*b\*b,b);

getch();

}

main()

{

int a1,b1;

clrscr();

printf("Enter a value to find square: ");

scanf("%d",&a1);

square(a1);

printf("\nEnter a value to find cube: ");

scanf("%d",&b1);

cube(b1);

}

**/\* program to demonstrate functions without return type and without parameters \*/**

void square(void)

{

int a1;

clrscr();

printf("Enter a value to find squre of it: ");

scanf("%d",&a1);

printf("%d is the Square of %d...",a1\*a1,a1);

getch();

}

void cube(void)

{

int a1;

clrscr();

printf("Enter a value to find cube of it: ");

scanf("%d",&a1);

printf("%d is the Cube of the %d...",a1\*a1\*a1,a1);

getch();

}

void main(void)

{

square();

cube();

}

**/\* program to write a function to find factorial of the given number \*/**

#include<stdio.h>

#include<conio.h>

long fact(long f);

void main()

{

long fac, num;

clrscr();

/\* Reading a value to find factorial of it \*/

printf("Enter a number to find factorial: ");

scanf("%ld", &num);

/\* calling a fact function which perform factorial for the

given number and returns \*/

fac = fact( num );

/\* printing the result \*/

printf("The Factorial of the given number is: %ld", fac);

getch();

}

long fact(long f)

{

long result = 1;

for(; f >= 1 ; f--)

result = result \* f;

return result;

}

**/\* program to perform factorial using function recursion \*/**

#include<stdio.h>

#include<conio.h>

int factrec(int);

void main()

{

int fact;

clrscr();

/\* Reading a number to find factorial \*/

printf("Enter a number to find factorial: ");

scanf("%d", &fact);

/\* calling the recursive factorial fucntion \*/

fact = factrec(fact);

/\* printing the resutl \*/

printf("the Factorial of the given number is: %d", fact);

getch();

}

int factrec(int n)

{

if( (n == 1) || (n == 0) )

return 1;

else

return n\*factrec(n-1);

}

**/\* program to swap two numbers by passing values \*/**

#include<stdio.h>

#include<conio.h>

void swap(int v1, int v2);

void main()

{

int a, b;

clrscr();

/\* Reading two values to swap \*/

printf("Enter two values to swap: \n");

printf("a: ");

scanf("%d",&a);

printf("b: ");

scanf("%d", &b);

/\* Calling the swap function with the values \*/

swap(a, b);

printf("The values in a and b are: \na: %d\nb: %d",a,b);

getch();

}

void swap(int v1, int v2)

{

int temp;

temp = v1;

v1 = v2;

v2 = temp;

}

/\* program to swap two numbers by passing references \*/

#include<stdio.h>

#include<conio.h>

void swap(int \*v1, int \*v2);

void main()

{

int a, b;

clrscr();

/\* Reading two values to swap \*/

printf("Enter two values to swap: \n");

printf("a: ");

scanf("%d",&a);

printf("b: ");

scanf("%d", &b);

/\* Calling Swap function with references \*/

swap(&a, &b);

printf("The values in a and b are: \na: %d\nb: %d",a,b);

getch();

}

void swap(int \*v1, int \*v2)

{

int temp;

temp = \*v1;

\*v1 = \*v2;

\*v2 = temp;

}