Functions

- Suppose you have a task that is always performed exactly in the same way- say a bimonthly servicing of your motorbike.When you want it to done ,you go to service station and say “it’s time ,do it now”.You don’t need to give istructions , because he knows his job.You don’t need to be told when the job is done. You assume the bike would be serviced in the usual way, the mechanic does it.

Simple example

```
#include<stdio.h>
#include<conio.h>
void service();
void main()
{
service();
printf (“I am clear”);
getch();
}
void service()
{
printf(“\n smile,and the world smiles with you”);
}
```

- Functions is a self-contained subprogram that is meant to do specific, well-defined task.
- A C program consist of one or more functions.
- If a program has only one function then it must be the main() function.

Advantages

- Generally a difficult program is divided into sub-problem and then solved. This divide and conquer technique is implemented in C through functions.
- A program can be divided into functions, each of which performs some specific task. So the use of C functions modularizes and divides the work of a program.
- When some specific code is to be used more than once and at different places in the program, the use of functions avoids repetition of that code.
- The program becomes easily understandable, modifiable and easy to debug and test. It becomes simple to write the program and understand what work is done by each part of the program

Types of functions

- Library function
- User-defined function

Library Function

- C has facility to provide library function for performing some operations.
- These functions are present in C library and they are predefined.
- For eg-sqrt() is a mathematical library function which is used for finding out the square root of any number.
- printf() and scanf() are input output library functions.
- To use a library function we have to include corresponding header files using the preprocessor directives ***#include***.

WAP to find square root of any number.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
int num,s;
printf(“enter the number”);
scanf(“%d”, &num);
s=sqrt(num);
printf(“%d”,s);
getch();
}
```

User-defined functions

- User can create their own functions for performing any specific task of the program.These types of functions are called user-defined functions.
- To create and use these functions ,we should know about these things-

  function declaration and prototype

  function definition

  function call

WAP to see how functions are applied

```
#include<conio.h>
#include<stdio.h>
void  italy();
void brazil();
void argentina();
void main()
{
printf("I am in main");
italy();
brazil();
argentina();
getch();
}
void italy()
{
printf("\n I am in italy");
}
void brazil()
{
printf("\n I am in brazil");
}
void argentina()
{
printf("\n I am in argentina");
}
```

Function Definition

- Function definition consists of whole description and code of a function.
- It tells what the function is doing and what are its inputs and outputs.
- Function definition consists of two parts-function header and function body.

```
return_type func_name(type arg1,type arg2,…)
{
local variable declaration;
statement;
…
return(expression);
}
```

- The return_type denotes the type of the value that will be returned by the function.A function can return either one value or no value.
- If the function does not return any value than void should be written in place of return_type.
- The function definition can be placed anywhere in the program. But generally all definitions are placed after the main() function.

Passing values between functions

- Like our mechanic who always services the motorbike in exactly the same way, we haven't been able to influence the function in the way they carry out their tasks.
- It would be nice to have a little more control over what function do,in the same way it would be nice to be able to tell the mechanic.
- The mechanism used to convey information to the function is the 'argument'.
- The list of variables used inside the parantheses in these functions are arguments.
- These arguments are sometimes called as parameters.
- A function can take any number of arguments or even no argument at all.
- If there are no arguments then either the parentheses can be left empty or void can be written inside the parantheses.

WAP to find sum of two numbers

```
#include<stdio.h>
#include<conio.h>
int sum(int x,int y);          /*function declaration*/
main()
{
int a,b,c;
scanf("%d%d",&a,&b);
c=sum(a,b);                    /*function call*/
printf("%d",c);
getch();
}
int sum(int x,int y)           /*function definition */
{
int s;
s=x+y;
return s;
}
```

- In this program the value of a & b are passed on to the function sum().

  s=sum(a,b);

Here ‘a’ and ‘b’ are ‘actual arguments’ whereas x and y are formal arguments.

Any number of arguments can be passed to a function being called. However,the type ,order and number of actual and formal arguments must be same.

Function Declaration

- The calling function needs information about the called function.
- If definition of the called function is placed before the calling function then declaration is not needed.

WAP to find sum of two numbers

```
#include<stdio.h>
#include<conio.h>
int sum(int x,int y)          /*function definition */
{
int s;
s=x+y;
return s;
}
main()
{
int a,b,s;
scanf("%d%d",&a,&b);
s=sum(a,b);                   /*function call*/
printf("%d",s);
getch();
}
```

WAP to find whether the number is even or odd

```
#include<conio.h>
#include<stdio.h>
void even(int num);
void main()
{
int num;
scanf("%d",&num);
even(num);
getch();
}
void even(int num)
{
If(num%2==0)
    printf("num is even");
else
    printf("num is odd");
}
```

WAP to find the larger among two numbers

```
#include<stdio.h>
#include<conio.h>
int larger(int x,int y);
void main()
{
int a,b,l;
printf("%d%d",&a,&b);
l=larger(a,b);
getch();
}
int larger(int x,int y)
{
if(x>y)
    return x;
else
    return y;
}
```

Return statement

- Return statement is used in a function to return a value to the calling function.
- A function can return only one value at a time.
- If no value is to be returned from the function then it should be declared as void.
- All functions which are not of void type ,return a value
  - » return(expression);

Category of functions

- Function with no argument and no return values.
- Function with argument and no return values.
- Function with argument and return value.
- Function with no argument but return value.

Function with no argument and no return values

```
#include<stdio.h>
#include<conio.h>
void message();
void main()
{
message();
printf (“I am clear”);
getch();
}
void message()
{
printf(“\n smile,and the world smiles with you”);
}
```

Function with no argument and return values

```
#include<stdio.h>
#include<conio.h>
int sum();
void main()
{
int s;
s=sum();
printf (“%d”,s);
getch();
}
int sum()
{
int i,sum=0;
for(i=0;i<5;i++)
sum=sum+i;
return sum;
}
```

Function with argument but no return values

```
#include<conio.h>
#include<stdio.h>
void even(int num);
void main()
{
int num;
scanf("%d",&num);
even(num);
getch();
}
void even(int num)
{
If(num%2==0)
    printf("num is even");
else
    printf("num is odd");
}
```

Function with argument and return values

```
#include<stdio.h>
#include<conio.h>
int larger(int x,int y);
void main()
{
int a,b,l;
printf("%d%d",&a,&b);
l=larger(a,b);
getch();
}
int larger(int x,int y)
{
if(x>y)
    return x;
else
    return y;
}
```

Recursion

- Recursion is a process,when a function calls itself.

For eg:-

```
main()
{
printf("this is an example of recursion");
main();
}
```

WAP to find factorial using recursion

```
#include<stdio.h>
#include<conio.h>
int fact(int n);
void main()
{
int num;
scanf("%d",&num);
F=fact(num);
Printf("fact=%d",f);
getch();
}
Int fact(int n)
{
int factorial;
If (n==1)
      return (1);
else
      factorial=n*fact(n-1);
      return (factorial);
}
```

Storage Classes

- Storage classes briefly define four aspects of a variable-
- **Lifetime** refers to the period during which a variable retains a given value during execution of a program(“alive”).
- **Scope** of variable determines over what region of the program a variable is actually available for use(“active”).
- **Initial Value** Default value taken by an uninitialized variable.
- **Place of storage** Place in memory where the variable is stored.
- The variables may also be broadly categorized, depending on the place of their declaration, as internal(local) or external(global).
- Internal variables are those which are declared within the particular function, while external variables are declared outside of any function.

Different storage classes are

1.Automatic variables **2.External Variables**

3.Static Variables **4.Register Variables**

Automatic Variables

- Are declared inside a function in which they are utilized.
- They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic.
- Automatic variables are therefore private(or local) to the function in which they are declared.
- A variable declared inside a function without storage class specification is, by default, an automatic variable.
- Uninitialized automatic variable initially contain garbage value.

```
main()
{
int num;
------
------
}
```

```
main()
{
auto int num;
------
------
}
```

Program to understand automatic variables

```
#include<stdio.h>
main()
{
int x=5;
printf(“%d”,x);
func();
}
func()
{
int x=15;
printf(“%d”,x);
}
```

External Variables

- Variables that are both alive and active throughout the entire program are known as external variables . These are called as global variables.
- Declaration of external variables declare the type and name of variable , while definition reserves space for variable.
- Unlike local variables , global variables can be accessed by any function in the program.
- External Variables are declared outside a function.
- Initial value of an uninitialized external variable is zero.
- Declaration
    - Extern float salary

```
int x=8;
main()
{
-------;
}
function1()
{
----
}
function2()
{
----
}
```

```
#include<stdio.h>
main()
{
-------;
}
function1()
{
----
}
int x=8;
function2()
{
----
}
```

```
#include<stdio.h>
main()
{
extern int x;
…..
}
function1()
{
----
}
int x=8;
function2()
{
----
}
```

Static Variables

- The value of static variables persist until the end of the program.
- A variable can be declared static using the keyword static like

  Static int x;
- The lifetime of static variable is more than that of an automatic variable.
- A static variable is created at the compilation time and remains alive till the end of program.
- It is not created and destroyed each time the control enters a function/block. Hence a static variable is created only once and its value is retained between function calls.

Program to understand the use of static variables

```
#include<stdio.h>
void func(int x,int y);
main()
{
func();
func();
func();
}
func(int x,int y)
{
static int x=2,y=5;
printf("%d%d",x,y);
x++;
y++;
}
```

Output:
X=2,y=5
X=3,y=6
X=4,y=7

Register Variables

- Register storage class can be applied only to local variables.
- The scope, lifetime and initial values are same as that of automatic variables.
- We can tell the compiler that a variable should be kept in one of the machine register, instead of keeping in the memory.
- Since a register access is much faster than a memory access , keeping the frequently accessed variables in the register will lead to faster execution of programs.

register int count;

- A storage class is an attribute that tells us where the variable would be stored, what will be the initial value of the variable if no value is assigned to that variable, life time of the variable and scope of the variable.

Storage class	Keyword	Default value	scope	Place of storage
Automatic	auto	Garbage value	local	Memory
Register	register	Garbage value	local	CPU Registers
Static	static	zero	local	Memory
External	extern	zero	global	Memory

Table 11.1 Scope and Lifetime of Variables

Storage Class	*Where declared*	*Visibility (Active)*	*Lifetime (Alive)*
None	Before all functions in a file (may be initialized)	Entire file plus other files where variable is declared with **extern**	Entire program (Global)
extern	Before all functions in a file (cannot be initialized) **extern** and the file where originally declared as global.	Entire file plus other files where variable is declared	Global
static	Before all functions in a file	Only in that file	Global
None or **auto**	Inside a function (or a block)	Only in that function or block	Until end of function or block
register	Inside a function or block	Only in that function or block	Until end of function or block
static	Inside a function	Only in that function	Global