4.1 Function

A function is a part of program that performs a specific task and can be invoked from any part of program.

A function is a set of statements that take inputs, do some specific computation and produces output.

The basic philosophy of function is divide and conquer, by which a complicated task is successively divided into simpler and more manageable tasks which can be easily handled.

There are some situations when we need to write a particular block of code for more than once in our program.

A program can be divided into smaller subprograms that can be developed and tested successfully.

A function is a complete and independent program which is used (or invoked) by the main program or other subprograms.

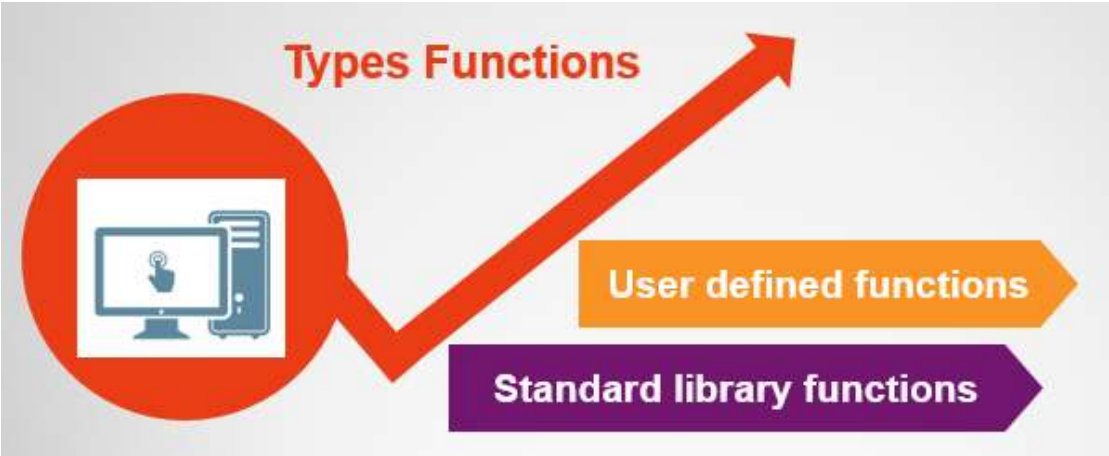
A subprogram receives values called arguments from a calling program, performs calculations and returns the results to the calling program.

A function is a self-contained block of statements that perform a task of some kind.

Every C program should have at least one function, which is main(), from where the execution of program begins.

Function allows us Top down approach i.e. functions are executed from top to bottom.

There are 2 types of functions



Advantages of Functions

- It provides modularity to the program.
- Program development is made easy due to modularity.
- Modular programming makes C program more readable.

- Avoid repetition of codes, due to function we can reuse code again and again hence it supports code reusability. We just have to call the function by its name to use it.
- It makes the program easier to design and understand
- Program testing becomes easy
- In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.

Characteristics of a function.

- Any C program should contain at least one function.(main())
- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calling in main( ).
- After each function has done its work, control returns to main( ).
- It facilitates top-down modular programming.
- A function can be called more than once.
- The length of a source program can be reduced by using functions.
- Any function can call any other function.
- Function can call itself.
- Default return type of any function is int
- Function can return only one value.
- We can specify as many parameters as are specified in the function definition.

4.1.1 Standard Library Function or Inbuilt Function

These are the readymade functions available for use and they resides in their respective header files. You do not need to write the code for it. Examples of such functions are printf(), scanf(), gets(), puts(), strcpy(), strcmp(), sqrt() etc.

Some Useful function and its header files,

stdio.h: I/O functions

Function Name	Function Description
getchar()	returns the next character typed on the console.
putchar()	outputs a single character to the screen.
printf()	as previously described
scanf()	as previously described

string.h: String functions

Function Name	Function Description
strcat()	concatenates a copy of str2 to str1
strcmp()	compares two strings
strcpy()	copies contents of str2 to str1
strlen()	return length of the string

ctype.h: Character functions

Function Name	Function Description

isdigit()	returns 1 if argument is digit i.e. 0 to 9 else return 0
isalpha()	returns 1 if argument is a letter of the alphabet else return 0
isalnum()	returns 1 if argument is a letter or digit else return 0
islower()	returns 1 if argument is lowercase letter else return 0
isupper()	returns 1 if argument is uppercase letter else return 0
tolower()	converts letter into lowercase
toupeer()	converts letter into uppercase

math.h: Mathematics functions

Function Name	Function Description
acos()	returns arc cosine of argument
asin()	returns arc sine of argument
atan()	returns arc tangent of argument
cos()	returns cosine of argument
exp()	returns natural log
fabs()	returns absolute value of argument
sqrt()	returns square root of number passed as argument

stdlib.h: Miscellaneous functions

Function Name	Function Description
malloc()	provides dynamic memory allocation, covered in future sections
rand()	as already described previously
srand()	used to set the starting point for rand()
exit()	used to exit from the main() function

4.1.2 What is User Defined Function (UDF)?

We can divide our code into separate functions.

User-defined functions are those functions which are defined by the user.

It is also known as sub program.

Functions are made for code reusability and for saving time and space.

Advantages of UDF

- We can avoid writing redundant program code of some instructions again and again.
- Reduce program size.
- Programs, by using functions become compact & easy to understand.
- Testing and correcting errors is easy because errors are localized and hence can be corrected easily.
- It provides top to down model which is easy to understand.
- Reusability

Disadvantages of UDF

- Execution is slow

Difference between UDF and Library function

Library function	UDF
It is provided by 'c' libray	User create it for reusability
We have to add header file	There is no need to add any extra header file
There are no error because they are pre-compiled	Possibility of errors
No types	There are 5 types of UDF
Library Functions are part of header file (i.e. MATH.h) which is called runtime.	UDF are part of the program which compile runtime

Example# printf(), scanf()	Example#
	void add(int a,int b)
	{
	printf("\nAdd = %d",a+b);
	}

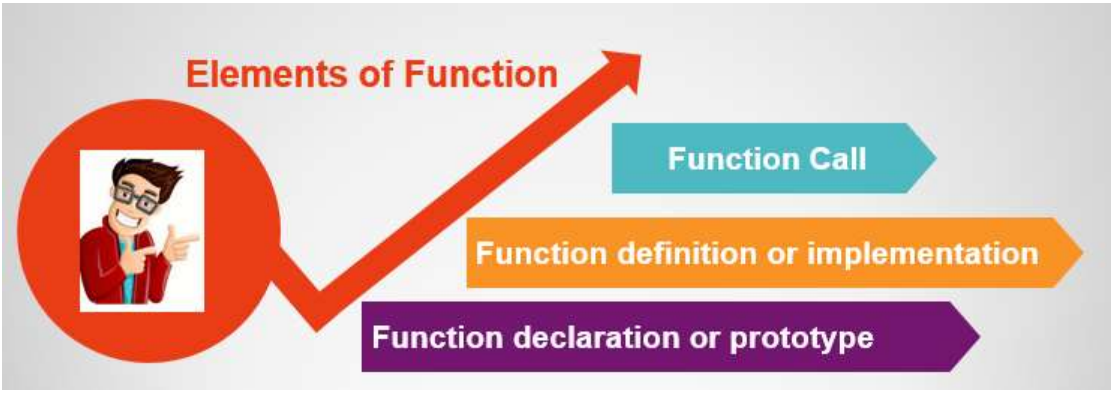
Syntax#

```
return-type function-name ( argument-list-if-necessary )
{
   ...local-declarations...

   ...statements...

   return return-value;
}
```

Examples:

```
void print()
{
printf("Ankit Thakkar");
}

int div(int x,int y)
{
return x/y;
}
```

4.1.2.1 Elements of User-defined function



4.1.2.1.1 Function Prototype/Declaration

The calling function that is to be used later in the program is known as function declaration or function prototype.

Function declaration is a statement that informs the compiler about

1. Name of the function
2. Type of arguments
3. Number of arguments
4. Data Type of Return value

The arguments declared at the time of definition of program are called formal argument.

It's not necessary that the variable name of actual arguments and formal arguments are same.

Syntax#

returntype function_name ([arguments type]);

Examples#

void square();

function name = square, without any arguments and without any return value.

void add(int a,int b);

function name = add, receives two integer values of integer as argument and returns nothing

char max(int a,int b);

function name = max, receives two integers as argument and returns an character

4.1.2.1.2 Function Definition

Function definition consists of the body of function.

Function definition consists actual code i.e. programmer's logic.

The body consists of block of statements that specify what task is to be performed.

Syntax#

```
returntype function_name ([arguments])
{
   Logic…
   … … …
}
```

Examples#

```
void square()
{
int a;

printf("\nenter value =>");
scanf("%d",&a);

printf("\nSquare = %d",a*a);
}
```

Here void is a return type that means function is not returning any value. Function does not have any arguments.

```
void add(int a,int b)
{
printf("\nAdd = %d",a+b);
}
```

Here void is return type means function is not returning any value. Function have two arguments.

```
char max(int a,int b)
{
        if(a>b)
        reutrn 'a';
        else
        return 'b';
}
```

Here function is returning char value and function have two arguments.

4.1.2.1.3 Function calling

To use the function we need to call it.

A function can be called by using the function name followed by a list of actual parameters.

The arguments that we pass from the calling function are called actual arguments.

Syntax#
function_name ([actual arguments]);

Example#

```
Square();
add(5,6);
ans = max(a,b);
```

Example: Function for odd even

```
#include<stdio.h>
#include<conio.h>

void OE();      //function prototype

main()
{
        clrscr();
        OE();   //function calling here no any argument
        getch();
}

void OE()       //function defintion
{
        int no;
        printf("\nEnter no ->");
        scanf("%d",&no);

        if(no%2==0)
                printf("\nNo is even");
        else
                printf("\nNo is odd");
}
```

Output:
Enter no -> 5
No is odd

What is global and local prototype?

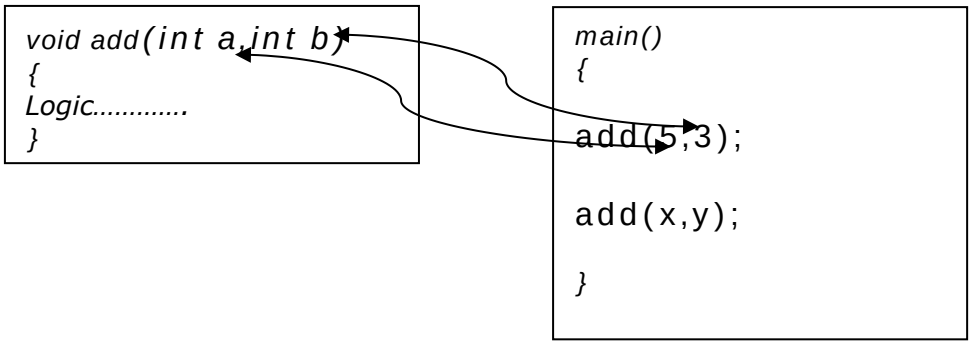
Prototype declaration may be placed into main() or before main() .

When we place Prototype before main() then it's called global prototype.(Good)

When we place Prototype in the main() then it's called local prototype.

4.3 What is argument or parameter?

- Parameter = Argument
- Function may or may not have argument.
- When a function is defined, its parameters are listed in parentheses next to its name.
- If a function does not accept any parameters, leave the parentheses empty () or type (void)
- When we want to pass values from the main() function to user define function then we need to create function with arguments.
- The values of arguments which we send from the main() should be match with the sequence of the argument which we define in the prototype and logic of the function.
- In below image function have 2 arguments a and b.



- In C, by default, all function arguments are passed by value, and the called function receives a copy of the value passed to it.

4.4 What is return value?

- Function can return only one value.
- A C function returns a value of type int as the default data type when no other type is specified explicitly.
- It's not compulsory that every function should return some value to the main() function.
- Function may have more than one return statement.
- If function does not return anything then return type of function is void.

If function returns a value then the function calling statement will change as given below:

var-name=function-name(arguments);

If not, function-name(arguments)

Example:

```
#include<stdio.h>
#include<conio.h>

void add(int a,int b)
{
```

```
        printf("Addition = %d",(a+b));
}

int sub(int a,int b)
{
        return a-b;
}

main()
 {
        int x,y,z;
        clrscr();
        x=20;
        y=10;
        add(x,y);       // does not return value
        z=sub(x,y); //return value
        printf("\n\nSubtraction = %d",z);

        getch();
 }
```

Output:

Addition = 30
Subtraction = 10

Some examples of function returning value.

Power function	Square function

int power(int a,int b)
{
int i,ans=1;
for(i=1;i<b;i++)
ans=ans*a;
 	
return ans;
}
	int square(int a)
	{
	return a*a;
	}

Lowercase function	Simple Interest function

char Lowercase(char a)
{
if(islower(a))
return 'y';
else
return 'n';
}
	float Intereset(float p,float r,int n)
	{
	float ans;
	ans=(p*r*n)/100;
	return ans;
	}

Maximum between 2 values function int max(int a,int b)	Return Ascii value int retAs(char value) {

```
{
        return (a>b)?a:b;
}
```

```
    return value;
}
```

4.5 Categories of UDF

1. Function with no arguments and no return values.(NANR)
2. Function with arguments and no return values.(WANR)
3. Function with arguments and return value.(WAWR)
4. Function with no arguments and return a value.(NAWR)

Function with no arguments and no return values.

In this type, function does not contain arguments and does not return any value.

Syntax#

void function-name();

Example#

```
#include<stdio.h>
#include<conio.h>

void add()
{
        int a,b;
        a=22;
        b=5;
        printf("\nAdd = %d",a+b);
}

main()
{
        clrscr();
        add();
        getch();
}
```

Output:
Add = 27

In above program,

add() function is defined without any return type and arguments list. When a function does not return any value it return type is void.

From the main() function, add() function is called. When execution of add() function is finished then the execution gets return back to the main() because it was called from main() and program execution is finish when all the statements of main() are executed.

4.5.1 Function with arguments and no return values.

In this type, function does not return a value but it contains arguments.

Syntax#

void function-name(arg1, agr2,……..argN);
Where, arg1,arg2,……argN are the list of variables with data type.

Example#

```
#include<stdio.h>
#include<conio.h>

void add(int a,int b)
{
        int c=a+b;
        printf("\n\nAns = %d",c);
}

main()
{
        int x,y;
        clrscr();
        printf("\n\nEnter x and y ->");
        scanf("%d %d",&x,&y);
        add(x,y);
        getch();
}
```

Output:
Enter x and y ->10 5
Ans = 15

In above program,

add(int a,int b) function is defined without return type and with arguments list. As we discussed earlier when a function does not return any value it return type is void.

From the main() function, add(int a,int b) function requires two values from the main() function, so two arguments x and y are passed and then further the values of x and y are copied into arguments a and b respectively. When add(int a,int b) function finish execution, execution return back to the main() because it was called from main() and program finish the execution.

4.5.2 Function with arguments and return value.

In this type, functions have both, arguments and return type.

Syntax#
Data-type function-name(arg1, agr2,……..argN);

Example#

```
#include<stdio.h>
#include<conio.h>

int cube(int a)
{
        return a*a*a;
}

main()
{
        int ans;
        clrscr();
        ans=cube(3);
        printf("\n\nCube= %d",ans);
        getch();
}
```

Output:
Cube = 9

In above program,

add(int a,int b) function is defined with return type and also with arguments list. With return type int.

From the main() function, cube(int a) is called by passing 3 as argument, int a gets the value 3 and the cube() function returns its cube i.e. 9 which is stored in ans variable and then the program execution gets finished.

4.5.3 Function with no arguments but return a value.

In this type, function does not contain arguments but it returns a value.

Syntax#
Data-type function-name();

Where, data-type is a C data type such as int, float, char etc.

Example#

```
#include<stdio.h>
#include<conio.h>

float CircleArea()
{
        int r;
        printf("Enter r ->");
```

```
        scanf("%d",&r);

        return r*r*3.14;
}

main()
{
        float ans;
        clrscr();
        ans=CircleArea();
        printf("\n\nans= %f",ans);
        getch();
}
```

Output:

```
Enter r ->10
314
```

In above program,

CircleArea() function is defined without any argument list and it returns a float value.

From the main() function, CircleArea() is called as it does not have any argument list so there is no need to pass actual arguments while calling. When return statement of CircleArea() is called execution comes back to the main function and it also return the area and its value is stored in ans variable.

4.6 Nesting of functions

C permits nesting of two functions.

There is no limit how deeply functions can be nested.

Suppose a function a() calls function b() and function b() calls function c() and so on.

```
#include<stdio.h>
#include<conio.h>

void sub(int a,int b)
{
        printf("\nSub = %d",a-b);
}

void add(int a,int b)
{
        printf("\nAdd = %d",a+b);
        sub(a,b);
}
```

```
void mathop(int a,int b)
{
        add(a,b);
}

main()
{
        clrscr();

        mathop(22,2);

        getch();
}
```

Output:

```
Add = 24
Sub = 20
```

4.7 Recursion Function

Recursive function is a function that calls itself.

When a function calls another function and that second function calls the third function then this kind of a function is called nesting of functions.

But a recursive function is the function that calls itself repeatedly.

Recursion means function calls itself.

When we write recursive function, we must have add if statement somewhere to force the function to return, without the recursive call being executed. Otherwise the function will never return.

A simple example:

```
#include<stdio.h>
main()
{
        printf("Hi all");
        main();
}
```

Output:

```
Infinite.....
```

When this program is executed. The line is printed repeatedly and indefinitely. We might have to abruptly terminate the execution.

Example# Factorial using Recursion

```
#include<stdio.h>
#include<conio.h>

int fact(int a)
 {
        int ans;
        if(a==1)
        {
                return 1;
        }
         else
        {
                ans=a*fact(a-1);
        }

        return ans;

 }

main()
{
        int a,ans;
        clrscr();
        printf("\nEnter value ->");
        scanf("%d",&a);
        ans=fact(a);
        printf("\n\nans = %d",ans);

        getch();
}
```

Output:

```
Enter value ->5
120
```

Example# Fibonacci Series using Recursion

```
#include<stdio.h>
#include<conio.h>

void printFibonacci(int);

void main(){

        int k,n;
        long int i=0,j=1,f;
```

```
        clrscr();

        printf("Enter the range of the Fibonacci series =>");
        scanf("%d",&n);

        printf("\nFibonacci Series: ");
        printf("%d %d ",0,1);
        printFibonacci(n);

        getch();
}

void printFibonacci(int n){

        static long int first=0,second=1,sum;

        if(n>0)
        {
                sum = first + second;
                first = second;
                second = sum;
                printf("%ld ",sum);
                printFibonacci(n-1);
        }

}
```

Output:

```
Enter the range of the Fibonacci series =>10

Fibonacci Series: 0 1 1 2 3 5 8 13 21 34 55 89
```

Example# Prime number using Recursion

```
#include<stdio.h>
#include<conio.h>

int primeno(int, int);

void main()
{
         int num, check;
        clrscr();
        printf("Enter a number =>");
        scanf("%d", &num);
```

```
        check = primeno(num, num / 2);

        if (check == 1)
        {
                printf("%d is a prime number\n", num);
         }
        else
        {
                printf("%d is not a prime number\n", num);
        }
        getch();
}

int primeno(int num, int i)
{

        if (i == 1)
        {
                return 1;
        }
        else
        {
                if (num % i == 0)
                {
                        return 0;
                }
                else
                {
                        return primeno(num, i - 1);
                }
        }
}
```

Output

```
Enter a number =>13
13 is a prime number
```

4.8 Functions and arrays

We can pass an entire array of values into a function same as we pass individual variables. In this task it is essential to list the name of the array along with the function arguments without any subscripts and the size of the array is declared in the function definition.

Syntax#

```
FunctionName(arrayName);
```

Will pass all the elements that are in that array 'a' of size n. the called function expecting this call must be appropriately defined.

Example# Array as a function argument

```
#include<stdio.h>
#include<conio.h>

void Largest(int a[],int n);
void setArray(int a[],int n);
void printArray(int a[],int n);

void main()
{

	int a[100],n;

	clrscr();

	printf("Enter the limit =>");
	scanf("%d",&n);

	setArray(a,n);
	printArray(a,n);
	Largest(a,n);

	getch();
}

void setArray(int a[],int n)
{
	int i;
	for(i=0;i<n;i++)
	{
		printf("\nEnter value of a[%d] =>",i);
		scanf("%d",&a[i]);
	}

}

void printArray(int a[],int n)
{
	int i;
	printf("\nValues");
	for(i=0;i<n;i++)
	{
		printf("\n%d ",a[i]);
	}
}

void Largest(int a[],int n)
```

```
{
        int max=a[0];
        int i;
        for(i=0;i<n;i++)
        {
                if(max<a[i])
                {
                        max=a[i];
                }
        }
        printf("\nLargest value is = %d",max);
}
```

Output

```
Enter the limit =>5

Enter value of a[0] =>11

Enter value of a[1] =>22

Enter value of a[2] =>3

Enter value of a[3] =>4

Enter value of a[4] =>6

Values
11
22
3
4
6
Largest value is = 22
```

4.9 Scope and lifetime of variables in functions.

Scope

The scope actually determines that over which part or parts of the program the variable is available i.e. it can be excessed.

Variable scope is a region of a program in which a variable is available for use.

There are two types of Scope: Local Scope and Global Scope.

Example#

```
#include<stdio.h>
#include<conio.h>

int x=20;  //Global scope
int y=30; //Global scope

void main(){

   int a=40;  //Local scope
   int y=50; //Local scope

   clrscr();

   printf("\nX = %d Y = %d",x,y);
   printf("\nA = %d",a);

   getch();
}
```

Output:

```
X = 20 Y = 50
A = 50
```

Lifetime

The lifetime of the variable retains a given value.

The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Lifetime of local variables starts when control enters the function in which it is declared and it is destroyed when control exists from the function.

Global variables exist in the memory as long as the program is running. These variables are destroyed from the memory when the program terminates. These variables occupy memory longer than local variables.

4.10 Storage class of a variable

The program's ability to access a variable from the memory.

The storage class determines the scope and lifetime of a variable.

The scope and lifetime depends on the storage class of the variable in c language the variables can be any one of the four storage classes:

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

Automatic variables:

Automatic variables are declared inside a particular function and they are created when the function is called and destroyed when the function exits.

Automatic variables are local or private to a function in which they are defined by default, all variable declared without any storage specification is automatic.

The values of variable remains unchanged to the changes that may happen in other functions in the same program and by doing this no error occurs.

```
void function1()
{
	int m=10;
	printf("\nInside fun1 no = %d",m);
}

void function2()
{
	int m=100;
	function1();
	printf("\nInside fun2 no = %d",m);
}

void main()
{
	int m=1000;
	clrscr();
	function2();
	printf("\nInside main no = %d",m);
	getch();
}
```

Output:

```
Inside fun1 no = 10
Inside fun2 no = 100
Inside main no = 1000
```

A local variable lives throughout the whole program although it is accessible only in the main function.

A program with two subprograms function1 and function2 with m as automatic variable and is initialized by values 10, 100, 1000 in function1, function2 and main respectively.

When program is executed main calls function2 which in turns calls function1. When main is active m is equal to 1000, but when function2 is called, the main m is temporarily put on the shelf and the new local m=100 becomes active.

Similarly when function1 is called both previous values of m are put on shelf and latest value (m=10) become active, a soon as it is done main (m=1000) takes over. The output clearly shows that value assigned to m in one function does not affect its value in the other function. The local value of m is destroyed when it leaves a function.

External variables:

Variables which are common to all functions and accessible by all functions of a program are called external variables.

External variables can be declared outside a function.

Example:

```
int m=10;

void function1()
{
	printf("\nInside fun1 no = %d",m);
}

void function2()
{
	m=m+20;
	printf("\nInside fun2 no = %d",m);
}

void main()
{
	clrscr();
	printf("\nInside main M = %d",m);
	function2();
	function1();
	getch();
}
```

Output:

```
Inside main M = 10
Inside fun2 no = 30
Inside fun1 no = 30
```

A global value can be used in any function i.e. all the functions in a program can access the global variable and change its value. The subsequent functions get the

new value of the global variable, it will be inconvenient to use a variable as global because due to this factor every function can change the value of the variable on its own and it will be difficult to get back the original value of the variable if it is required.

Global variables are usually declared in the beginning of the program i.e. before the main program however c provides a facility to declare any variable as global this is possible by using the keyword storage class extern.

Static variables:

The value given to a variable declared by using keyword static persists until the end of the program.

A static variable is initialized only once, when the program is compiled.

It is never initialized again.

During the first call to stat in the example shown below x is incremented to 1. because x is static, this value persists and therefore the next call adds another 1 to x giving it a value of 22. The value of x becomes 25 when 5th time call is made. If we had declared x as an auto then output would here been x=21 all the 5 times.

```
void stat()
{
        static int x=20;
        x=x+1;
        printf("\n x=%d",x);
}

main()
{

        int j;
        clrscr();
        for(j=1;j<=5;j++)
        stat();
        getch();
}
```

Output:

```
 x=21
 x=22
 x=23
 x=24
 x=25
```

Register variables:

A variable is usually stored in the memory but it is also possible to store a variable in the compilers register by defining it as register variable.

The registers access is much faster than a memory access, keeping the frequently accessed variables in the register will make the execution of the program faster.

Example:

register int no;

Since only a few variables can be placed in a register, it is important to carefully select the variables for this purpose. However c will automatically convert registered variables into normal variables once the limit is exceeded.