UNIT-3

* Strings

Introduction:-

⇒ A string is a sequence of characters that is treated as a single data item.

Any group of characters defined between double quotation marks is a string constant

Ex: "hello world".

* The common operations performed on character strings include:

⇒ Reading and writing strings
⇒ combining strings together
⇒ copying one strings to another
⇒ comparing strings for equality
⇒ Extracting a portion of a string

* Declaration of a string:-

```
char string-name[size]
```

Ex: char city[10];
    char name[30];

* Initialisation of a string:-

⇒ String can be initialised in 2 ways

1) Compile time
2) Run time..

1) Compile time:-

Compile time initialisation can be done in different ways

```
char city[9] = "NEW YORK";
```

char city[9] = {'N', 'E', 'W', 'Y', 'O', 'R', 'K', '\0'};

char string[] = {'G', 'O', 'O', 'D', '\0'};

char str[10] = "GOOD";

2) Run-time initialisation:-

scanf("%s", s);

⇒ while reading the string using scanf ampersand(&) is not required

*NOTE:- we cannot seperate the initialization from declaration.

That is, char str[5];

str = "GOOD"

is not allowed.

* String Handling functions:-

⇒ These are also called as string manipulation functions.

The different string handling functions are:-

1) strlen
2) strcpy
3) strcat
4) strcmp
5) strrev

All the string handling functions are defined in <string.h>

* Strlen:-

This function is used to find the length of a string, the returned type is integer.

Syntax:

strlen(stringname);

* Write a C program to find the length of the string with and without using string handling functions.

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s[20], i, l=0;
    Printf("Enter a string");
    scanf("%s",s);
    Printf("The length of a string with string
            handling function is %ld", strlen(s));
    for(i=0; s[i]!='\0'; i++)
    l++;
    Printf("\n The length of a string without
        string handling function is %d", l);

    return 0;
}
```

* Strcpy:

This function is used to copy the contents of one string to another.

Syntax:-

```
strcpy(s2, s1);
```

In the above function s_1 is copied into s_2

*
```
#include<stdio.h>
#include<string.h>
int main()
{
  char s1[20], s2[20], i, j=0;
  Printf("Enter a string");
  scanf("%s", s1);
  strcpy(s2, s1);
  Printf("\n The copied string with string handling
                function is %s", s2);

  for (i=0; s1[i]!='\0'; i++)
  {
    s2[j]=s1[i];
    j++;
  }
  Printf("\n The copied string without string
          handling function is %s", s2);

  return 0;
}
```

* Strcat:-

This function is used to combine 2 strings

Syntax:

```
strcat(s1, s2);
```

```
#include <stdio.h>
#include<string.h>
int main()
{
    char s1[20], s2[20], s3[50], i, j=0, k;
    Printf("Enter 2 strings");
    scanf("%s %s", s1, s2);
    strcat(s1, s2);
    Printf("\n The combined string with string
            handling function is %s", s1);
    for(i=0; s1[i]!='\0'; i++)
    {
        s3[j]= s1[i];
        j++;
    }
    for(k=0; s2[k]!='\0'; k++)
    {
        s3[j]= s2[i];
        i++;
    }
    s3[j]='\0';
    Printf("\n The combined string without
    string handling function is %s", s3);
    return 0;
}
```

* Strcmp:-

This function is used to compare two strings

It returns 0 if both the strings are equal

otherwise it returns as ASCII difference of first unmatched character.

Syntax:-

Strcmp(s1,

* write a C

and without

#include<st

int main(

{ char s1

int i, x

Printf(

scanf("

Printf("

x = st

if (x =

Print

else

Print

Printf

for

if(s

Prin

els

Priv

re

}

Syntax:-

Strcmp (s1, s2);

* write a C program to compare two strings with and without using string handling functions.

```
#include<string.h>
int main()
{
  char s1[20], s2[20], s3[50];
  int i, x;
  Printf("Enter 2 strings");
  scanf("%s %s", s1, s2);
  Printf("\n Comparing using string handling
                                  function");

  x = strcmp(s1, s2);
  if (x == 0)
  Printf("\nStrings are equal");
  else
  Printf("\n Strings are not equal");
  Printf("\n Comparing without using string handlin
                                  function");

  for(i=0; s1[i] != '\0' && s2[i] != '\0'; i++);
  if(s1[i] == '\0' && s2[i] == '\0')
  Printf("\n Strings are equal");
  else
  Printf("\n Strings are not equal");
  return 0;
}
```

* Strrev:

This function is used to reverse the given string.

Syntax: Strev stringname

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s1[20], s2[20];
    int i,j=0;
    Printf("Enter a string");
    scanf("%.s", s1)
    Printf("\n The reverse of a string with string
            handling function is %.s", strrev(s1));
    for(i=strlen(s1)-1; i>=0; i--)
    {
      s2[j]=s1[i];
      j++;
    }
    s2[j]='\0';
    Printf("\n The reverse of a string without
     string handling function is %.s", s2);
     return 0;
}
```

* Array of Strings :-

It is also known as 2D-string

We often A list of names can be treated as a table of strings and a two-dimensional character array can be used to store the entire list.

Ex: a character array city[5][15] may be used to store a list of 5 city names, each of length not more than 15 characters.

* Declaration :-

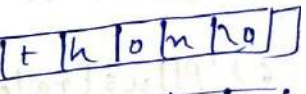
Char string-array-name [row-size] [column-size];

Ex: char student[5][20]

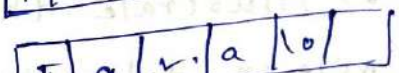
* Initialization :-

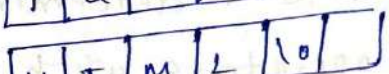
char language[5][10] = {"Java", "Python", "C++", "HTML", "SQL"}

char language[][10] = {"Java", "Python", "C++", "HTML", "SQL"}



the following declarations are invalid





char language[][] = {"Java", "Python", "C++", "HTML", "SQL"};

char language[5][] = {"Java", "Python", "C++", "HTML", "SQL"};

* Run-time :-

Scanf("%s", s[i]);

* Functions:-

* Definition:

⇒ A function is a block of code that performs a specific task. It has a name and it is reusable
⇒ It can be executed from as many different parts in a program as required, it can also return a value to calling program
⇒ All executable code resides within a function.
⇒ A C program consists of one or more functions.
⇒ A computer program cannot handle all the tasks by itself.
- It requests other program like entities called functions in C.

* Types of functions:-

* Predefined → It is not written by the user
e.g: Printf, scanf, sqrt, pow, strlen, strcpy
* User-defined → It is written by the user as per the neccesity

⇒ To create and use user-defined function we have to know these 3 elements.
* Function declaration
* function call
* function Definition.

* function Declaration:-

⇒ Any function should be declared when it is used in the c program
→ It has 4 parts:
* function return-type
* Function name
* Parameter list

* Semicolon

Syntax:

function-return type fun-name (formal parameter list);

*NOTE:

1) The parameter list must be seperated by commas
2) Use of parameter names in the declaration is optional
3) The parameter names may not be same in the function declaration and function definition but type and number should watch.
4) The return type must be void if no value is returned
5) If the function has no parameters, the list is written void or blank.

Ex:

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

* function call:-

→ A function can be called by simply using the function name followed by a list of actual Parameters if any.

Syntax:

function-name (parameter list);

* Function Definition:-

The syntax for the function definition:-

return-value-type function-name (parameter list)

{

declarations and statements

}.

* Declaration and statements: function body (block).

→ Variables can be declared inside blocks (can be nested)

→ functions can not be defined inside other functions.

* Returning control

- If nothing returned

o return;

o or, until reaches right brace.

- If something returned

o return expression;

* Return-type:

⇒ Return type is type of value returned by function

→ Return Type may be "void" if function is not going to return a value.

* function-name:

→ It is unique name that identifies function.

→ All variable naming conversions are applicable for declaring valid function name.

* Parameters:

→ comma seperated list of types and names of parameters

- Parameter injects external values into function on which function is going to operate.

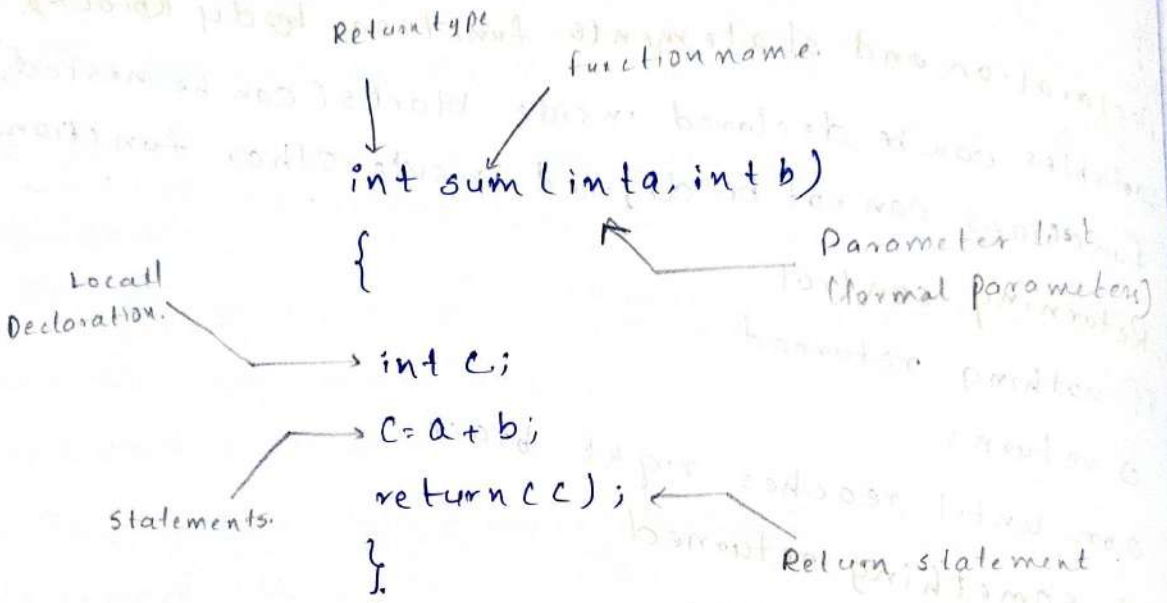
→ Parameter field is optional

→ If no parameter is passed then no need to write this field

* Value:

→ It is value returned by function upon termination.

→ function will not be return a value if return-type is void.



* Terminology associated with functions:

Parameter: The term parameter refers to any declaration within the parenthesis following the function name in a function declaration or definition

Ex: void sum (int a, int b); // a, b are parameters

Argument: The term argument refers to any expression within the parentheses of a function call

Ex: sum (num 1, num 2); // call function sum with two Parameters.

Calling function: The function which is making a call to any function is called calling function

Called function: The function which is being executed due to function call is known as called function

* Categories of functions :-

There are 5 categories of functions.

1) functions with no arguments and no return values.
2) functions with arguments and no return values
3) functions with arguments and return values.
4) functions with no arguments and return values.
5) functions returning multiple values.

* Write a C program to add 2 numbers using all the categories of functions.

(i) with arguments and with return type

```
#include<stdio.h>
int sum(int,int);
void main()
{
   int a,b,c;
   scanf("%d%d",&a,&b);
   c=sum(a,b);
   Printf("The sum = %d",c);
}
int sum(int x, int y)
{
   int z;
   z=x+y;
   return z;
}
```

Output:
10 20
The sum = 30.

* with arguments and with no return type.

```
#include<stdio.h>
void sum(int,int);
void main()
{
   int a,b;
```

```
    scanf("%d %d", &a, &b);
    sum(a, b);
  }
  void sum (int x, int y)
  {
    Printf("The sum = %d", x+y);
  }
```

* without arguments and with return type:-

```
#include <stdio.h>
int sum();
void main()
{
  Printf("The sum = %d", sum());
}
int sum()
{
   int a, b;
   scanf("%d%d", &a, &b);
   return a+b;
}
```

* without arguments and without return type:-

```
#include<stdio.h>
void sum();
void main()
{
  sum()
}
void sum()
{
   int a, b;
   Scanf("%d%d", &a, &b);
   Printf("The sum = %d", a+b);
}
```

*Write a C program to find largest of two numbers using all the categories of functions.

*without arguments and without return type

```
#include<stdio.h>
void big();
void main()
{
 big();
}
void big()
{
 int a,b;
 scanf("%d %d",&a,&b);
 if(a>b)
 Printf("%d is largest",a);
 else
 Printf("%d is largest",b);
}
```

*without arguments and with return type.

```
#include<stdio.h>
int big();
void main();
{
 Print("%d is largest",big());
}
int big()
{
 int a,b;
 scanf("%d %d",&a,&b);
 if(a>b)
 return a;
 else
 return b;
}
```

* with arguments and with return type:-

```
#include<stdio.h>
int big(int,int);
{
  int a,b;
  scanf("%d %d",&a,&b);
  Printf("%d is largest", big(a,b));
}
int big(int x, int y)
{
  if (x>y)
   return x;
  else
  return y;
}
```

~~(* with arguments and without return type:-)~~

* Parameters passing techniques:-

* There are 2 types:

1). Call by value

2) Call by reference

* Call by value:

→ In this method calling function sends a copy of actual values to called function, but the changes in called function does not reflect the original values of calling function.

* Call by reference:

* with arguments and without return type :-

```
#include<stdio.h>
void big(int x, int y)
{
  if (x>y)
  Printf("%d is largest", x);
  else
  Printf("%d is loogest", y);
}
void main()
{
  int a, b;
  scanf("%d%d", &a, &b);
  sum(a,b);
}
```

* call by value program:-

```
#include<stdio.h>
void swap(int, int);
void main()
{
  int a,b;
  scanf("%d%d", &a, &b);
  Printf("\n Before swapping a=%d, b=%d", a, b);
  swap(a, b);
  Printf("\n After swapping a=%d, b=%d", a, b);
}
void swap(int x, int y)
{
  int z;
  z=x;
  x=y;
  y=z;
  return;
}
```

* Call by reference:

→ In this method calling function sends address actual values as a parameter to called function, called function performs its task and sends the result back to calling function

→ Thus, the changes in called function reflect the original values of calling function.

→ To return multiple values from called to calling function we use pointer variables.

→ Calling function needs to pass "&" operator along with actual arguments and called function need to use "*" operator along with formal arguments.

→ Changing data through an address variable is known as direct access and "*" is represented as indirection operator.

*Write a C program to implement call by reference.

```
# include <stdio.h>
void swap (int *, int *);
void main ()
{
   int a, b;
   scanf("%d %d", &a, &b);
   Printf("\n Before swapping a = %d, b = %d", a, b);
   swap (&a, &b);
   Printf("\n After swapping a = %d, b = %d", a, b);
}
void swap(int *x; int *y)
{
   int z;
   z = *x;
   *x = *y;
   *y = z;
} return;
```

* Differences b/w call by value & call by reference

Call By value	Call By reference
1) It is also known as Pass by value.	1) It is also known as Pass by reference
2) values are passed as inputs	2) Addresses are passed as inputs
3) The actual arguments are variables, constants, expressions	3) The actual arguments are addresses of variable
4) The formal arguments are also variables.	4) The formal arguments are Pointers
5) Changes made in the called function are not reflected back in the calling function.	5) Changes made in the called function are reflected back in the calling function
6) The calling and the called function maintains seperate copies of data	6) The calling and the called function maintains or shares a common location.
7) A copy of contents is maintained by calling function.	7) The called functions shares common data
8) Example of function proto type void swap(int a, int b);	8) The ex of function prototype void swap(int *p, int *q);
9) Ex of function call: swap (a, b);	9) Example of function call: swap (&a, &b);

Actual Parameters	Formal Parameters.
1) These are the variable found in the function call.	1) These are the variable found in the function definition or declaration
2) They can be variables, constants or expressions or another function calls.	2) They can only be variables.

3) Example:
func1(12,23); // constants
func1(a,b); // variables
func1(a+b, b+a); // expression
	3) Ex:
	void func1(int x, int y); // always variables

4) They are the original source of information.	4) They get the value after function call occurs.
5) They are supplied by the caller or programmer	5) They are initialised with value of actual arguments

* Nested Functions:-

⇒ Calling the function inside the definition of another function is called as nested functions.

* write a C program to implement nested functions

```
#include <stdio.h>
void add(int, int, int);
int fact(int);
void main()
{
   int a,b,c;
   scanf("%d%d%d", &a, &b, &c);
   add(a,b,c);
}
void add(int x, int y, int z)
{
printf("%d", fact(x) + fact(y) + fact(z));
```

```
}
int fact(int p)
{
  int i, f=1;
  for(i=1; i<=p; i++)
  f = f*i;
  return f;
}
```

* Passing arrays to functions:

It is of two types

1) Passing individual element
2) Passing whole array.

* Passing individual element:

One-dimensional Arrays:

→ We can pass individual elements by either passing their data values or by passing their addresses.

→ We pass data value i.e, individual array elements just like we pass any data value.

→ As long as the array element type the function Parameter type, it can be passed

→ The called function cannot tell whether the value it receives comes from an array, a variable or an function.

```
void func1(int);
void main()
{
  int a[5] = {1,2,3,4,5};
  func1(a[3]);
```

```
}
void func1(int x)
{
 Printf("%d", x+100);
}
```

* Two-dimensional Array;

→ The individual elements of 2-D array can be pass in the same way as the 1-D array.

→ We can pass 2-D array elements either by value or by address

Example:

```
void func1(int);
void main()
{
 int a[2][2]={1,2,3,4};
 func1(a[0][1]);
}
void func1(int x)
{
 Printf("%d", x+10);
}
```

* Passing the whole array:-

→ As array name is a constant pointer pointing to first element of array (string the base address) Passing array name to a function itself indicates that we are passing address of first element of array

→ One dimensional array (passing array name as base address to a function)

→ To pass
array nam
→ In the ca
corresponding
→ we do no

Example:

```
void funl
void main
{
 int a[5]
 funl(a
}
void funl
{
 int i,
 for(i=
 sum=
 Printf
}.
```

* Two-D

→ when w
array na
→ The fo
however n

Rules:

→ The fun

→ To pass the whole array we simply use the array name as the actual parameter

→ In the called function, we declare that the corresponding formal parameter is an array.

→ we do not need to specify the number of elements.

Example:

```
void fun1(int x[], int);
void main()
{
  int a[5] = {1, 2, 3, 4, 5};
  fun1(a, 5);
}
void fun1(int x[], int n)
{
  int i, sum = 0;
  for(i=0; i<n; i++)
  sum = sum + x[i];
  Printf("\n Sum of the elements in an array is %d", sum);
}
```

* Two-Dimensional Array:

→ when we pass a 2-D array to a function, we use the array name as the actual parameter

→ The formal parameter in the called function header, however must indicate that the array has two dimension.

Rules:

→ The function must be called by passing only the

array name.

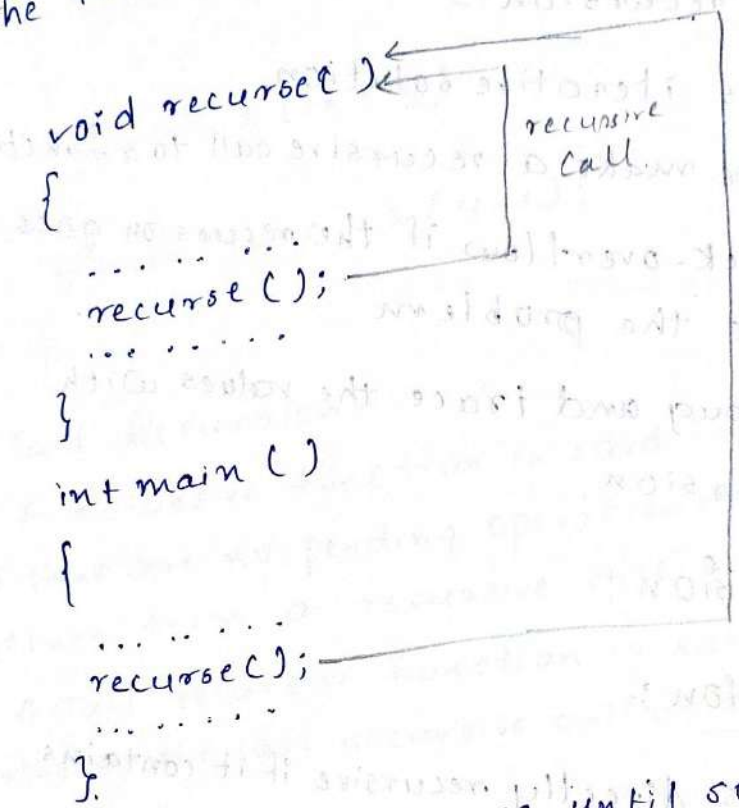
→ In the function definition, the formal parameter is a 2-D array with the size of the second dimension specified.

Ex:

```
void fun1(int a[][2]);
void main()
{
 int a[2][2] = {1, 2, 3, 4};
  fun1(a);
}
void fun1(int x[][2])
{
 int i, j;
 for(i=0; i<2; i++)
 {
 for(j=0; j<2; j++)
 Printf("%3d", x[i][j]);
 Printf("\n");
 }
}
```

*Recursion :-

⇒ Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function. then it is called a recursive call of the function.



⇒ The recursion continues until some condition is met to prevent it.

⇒ To prevent infinite recursion if else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

⇒ When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables

⇒ A recursive call does not make a new copy of the function. Only the values being operated upon are new.

⇒ As each recursive call returns, the old local variables and parameters are removed from the stack, and execution

→ resumes immediately after the recursive call in the function.

2) → The main advantage of recursive functions is that we can use them to create clearer and simpler versions of several programs.

* Limitations of recursion:-

→ Slower than its iterative solution.

→ For each step we make a recursive call to a function

→ May cause stack-overflow if the recursion goes too deep to solve the problem

→ Difficult to debug and trace the values with each step of recursion.

* Types of recursion:

1) Direct Recursion:-

→ A 'c' function is directly recursive if it contains an explicit call to itself.

Example:

```
int function(int a)
 {
   if (a<=0)
     return x;
   else
   return function (x-1);
 }
```

* Indirect Recursion:-

→ A 'c' function is indirectly recursive if it calls to another function which ultimately calls the function itself is called Indirect recursion.

Example:

```
int func 1 (in
{
  if (a<=0)
  return
  else
  return f
}
int func 2
{
  return f
}
```

Tail Recur

→ A Recursiv
if there are
return from

→ A Tail rec
value of the
function.

→ Tail Recu
amount of
the comput
recursive c

* Non-Tail

→ A recurs
recursive
be performed

Example:

```
int func1(int a)
{
  if (a<=0)
    return x;
  else
    return func2(a);
}
int func2(int y)
{
  return func1(y-1);
}
```

Tail Recursion:

→ A Recursive function is said to be Tail recursive if there are no pending operations to be performed on return from a recursive call.

* A Tail recursive function is said to return the value of the last recursive call as the value of the function.

* Tail Recursion is very desirable because the amount of information which must be stored during the computation is independent of the number of recursive calls.

* Non-Tail Recursion:

→ A recursive function is said to be Non-Tail recursive if there are some pending operations to be performed on return from a recursive call

* Storage Classes:

⇒ Variables in C differ in behaviour

→ The behaviour depends on the storage class a variable may assume.

→ From C compilers point of view, a variable name identifies some physical location within the computer where the string of bits representing the variables value is stored

→ Storage classes specify the scope of objects.

* There are four storage classes in C

→ Automatic storage class
→ Register storage class.
→ Static storage class.
→ External storage class.

* Automatic storage class

Keyword.	Auto.
Storage	Memory
Default initial value	An unpredictable value, which is often called a garbage value
Scope	Local to the block in which the variable is defined
Life	Till the control remains within the block in which the variable is defined.

Example:

```
#include<stdi
void main()
{
  int i,x;
  for(i=1; i<
  {
    x=1;
    x++;
    Printf("
  }
}
```

Output:-
value of x
value of x
value of x

* Registe

Keywo

Stora

Default
value

Scope

Life

Example:

```
#include<stdio.h>
void main()
{
  int i,x;
  for(i=1; i<=3; i++)
  {
    x=1;
    x++;
    Printf("value of x in iteration %d is:\n", i, x);
  }
}
```

Output:

value of x in iteration 1 is: 2
value of x in iteration 2 is: 2
value of x in iteration 3 is: 2

* Register storage class.

Keyword	Register
Storage	CPU Registers
Default initial value	An unpredictable value, which is often called a garbage value.
Scope	Local to the block in which the variable is defined.
Life	Till the control remains within the block in which the variable is defined.

Example :-

```
void main()
{
  register int i;
  for(i=1; i<=10; i++)
  Printf("\n%d", i);
}
```

* Static storage class:

Keyword	Static
Storage	Memory
Default intial value	Zero
Scope	Local to the block in which variable is defined
Life	value of the variable persists between different function calls

Example:

```
#include<stdio.h>
void main()
{
  int i;
  static int x;
  for(i=1; i<=3; i++)
  {
    x=1;
    x++;
    Printf("Value of x in iteration %d is \n", i, x);
  }
}
```

Output:
value of x in iteration 1 is: 2
value of x in iteration 2 is: 3.
value of x in iteration 3 is: 4

* External storage class.

Keyword	Extern
Storage	Memory
Default initial value	Zero
Scope	Global
Life	As long as the program execution does not come to end.

Example:

```
#include<stdio.h>
extern int i;
void main()
{
Printf("%d",i);
}
```

* Scope rules:

→ Scope: Scope defines the visibility of object. It defines where an object can be referenced. Scope of a variable is the part of program in which it can be used.

* Scope rules:

The rules are as under:

1. Use static storage class only if you want the value of a variable to persist between different function calls.

2. Use register storage class for only those variables that are being used very often in a program. Reason, is there are very few CPU registers at our disposal and many of them might be busy doing something else. Make careful utilization of the scarce resources. A typical application of register storage class is loop counters, which get used a number of times in a program

3. Use extern storage class for only those variables that are being used by almost all the functions in the program. This would avoid unnecessary passing of these variables as arguments when making a function call. Declaring all the variables as extern would amount to a lot of wastage of memory space because these variables would remain active throughout the life of the program.

4. If we don't have any of the express needs mentioned above, then use the auto storage class. In fact most of the times we end up using the auto variables, because often it so happens that once we have used the variables in a function we don't mind loosing them.