UNIT-II

Pointers:-

A pointer is a variable that contains an address which is a location of another variable in memory.

* Pointer enables us to access a variable that is defined outside the function.
* Pointers are more efficient in handling the data tables.
* Pointers reduce the length and complexity of a program.
* They increase the execution speed.
* The use of a pointer array to character strings results in saving of data storage space in memory.

Pointer variables:

* To differentiate ordinary variables from pointer variables, the pointer variable should proceed by "value at address operator".
* It returns the value stored at particular address.
* Pointer variables must be declared with its type in the program as ordinary variables.
* Pointer variables points to starting byte address of any datatype.
* Whenever we declare a variable, the system allocates somewhere in the memory, an appropriate location to hold the value of the variable.

  Since, every byte has a unique address number, this location will have its own address number.

Syntax:

```
datatype    *pointername;
```

This tells the compiler three things about variable pointername.

① The asterisk (*) tells that the variable pointername is a pointer variable

② pointername needs a memory location.

③ pointername points to a variable of type datatype.

Ex

```
int *p, v;
v = 20;
```

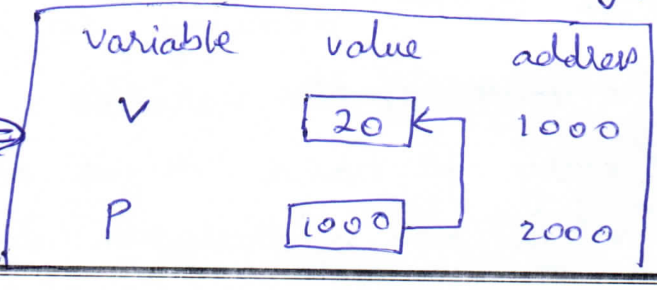
Here v is a variable of type integer.
p is a pointer variable.
v stores value and p stores address.

Pointer variables are initialized by $p = \& v$,
it indicates p holds the starting address of integer variable v.

Let us assume address of v is 1000. This address is stored in p.

Now p holds address of v, and
*p gives value stored at the address pointed by p.
*p = 20.

Pointer as a variable ⟹



Pointer operators:

There are two pointer operators: * and &.

① Operator (*):

→ It is a unary operator that returns the value located at the address that follows.

→ It is called as "value at address" operator.

→ It is also called as "indirection operator" and "de-reference" operator.

Ex:-

q = *m;

Here q will have the value stored at memory address of m.

② Operator (&):

→ It is a unary operator the returns the memory address of its operand. This address is computer's internal location of the variable.

→ It is called as "address operator"

→ It is also called as "reference operator".

Ex:-

m = &x;

Here memory address of x is placed into variable m.

⇒ Here &x is pronounced as "address of x".

⇒ The address may not be a fixed location and is randomly given by the computer.

Program:

```
#include <stdio.h>
int main()
{
 int n=20;
 printf(" The address of n is %u", &n);
 printf(" \n The ~~address~~ value of n is %d ", n);
 printf(" \n The value of n is %d ", *(&n));
}
```

output:

```
The address of n is 1000
The value of n is 20
The value of n is 20
```

How *(&n) is same as printing the value of n?

&n => gives address of the memory location whose name is 'n'.

* => means value at operator gives value at address specified by &n.

m = &n => Address of n is stored in m, but remember that m is not ordinary variable like 'n'.

Ex:

m = *(&n)
 = *(Address of variable 'n')
 = *(1000)
 = value at address 1000
 =20

Pointer Expressions

Like other variables, pointer variables can be used in expressions.

Ex: If p1 & p2 are two pointers, then the following statements are valid.

```
① y = *p1 * *p2;        (or)  y=(*p1) * (*p2);
② Sum = sum + *p1;
③ z = 5* - *p2 / *p1;   (or)  z=(5 *(-(*p2)))/(*p1);
④ *p2 = *p2 + 10;
```

In the expression ③, the symbol /* is considered as the beginning of a comment.
So, the statement is wrong.

⇒ C allows to add integers or to subtract integers from pointers.

Ex: p1 + 4 , p2 - 2 and p1 - p2

⇒ We may also use short-hand operators with the pointers.

Ex:
```
p1 ++;
-p2;
sum + = *p2;
```

⇒ Pointers can be compared using relational operators.

Ex: p1 > p2 , p1 = = p2 and p1 ! = p2 .

⇒ We may not use pointers in division or multiplication or addition.

Ex: p1/p2 or p1 * p2 or p1+p2 or p1/3 are not allowed.

Program:

```
main( )
{
 int a, b, *p1 , *p2 , x, y, z;
  a=12;
  b=4;
  p1 = &a;
  p2 = &b;
  x = *p1 * *p2-6;
  y= 4* - *p2 / *p1 +10;
 printf (" Address of a = %u \n", p1);
 printf (" Address of b= %u \n", p2);
 printf (" a= %d , b= %d \n", a, b);
 printf (" x= %d , y= %d \n", x, y);
*p2 = *p2 + 3;
*p1 = *p2-5;
 z = *p1 * *p2 -6;
 printf (" a= %d , b= %d , z= %d ", a, b, z);
}
```

output:

```
Address of a = 4020
Address of b = 4016
 a = 12 ,   b = 4
 x = 42,    y = 9
 a = 2 ,    b = 7,  z = 8
```

Pointers and Arrays

⇒ When an array is declared, the compiler allocates a base address & sufficient amount of storage to contain all elements of the array in memory.

⇒ The base address is the location of the first element (index 0) of the array.

⇒ The compiler defines the array name as a constant pointer to the first element.

Ex: int x[5] = {1, 2, 3, 4, 5};

Suppose the base address is 1000 & each integer requires 2 bytes, then five elements are stored as:

Elements →	x[0]	x[1]	x[2]	x[3]	x[4]
Value →	1	2	3	4	5
Address →	1000	1002	1004	1006	1008

The name **x** is defined as a constant pointer pointing to the first element, **x[0]**.

∴ The value of x is 1000, the location where **x[0]** is stored.

$$x = \& x[0] = 1000$$

If we declare **p** as an integer pointer, then pointer **p** is pointed to the array **x** as follows:

p = x;   (or)

p = &x[0];

Program:

```
main( )
{
 int *p, i ;
 int x[5] = {5, 9, 6, 3, 7};
 i = 0;
 p = x;
 printf(" Element   Value   Address \n ");
 while (i < 5)
  {
   printf (" x[%d]   %d   %u \n", i, *p, p);
   i++ , p++;
  }
  printf ("\n &x[0] = %u \n", &x[0]);
 printf ("\n p = %u \n", p);
}
```

Output:

```
   Element        Value      Address
    x[0]            5          166
    x[1]            9          168
    x[2]            6          170
    x[3]            3          172
    x[4]            7          174
 &x[0] = 166
   p   = 176
```

Indexing pointers

An array name without an index generates a pointer. A pointer can be indexed as if it were declared to be an array.

Ex: int x[5] = {1, 2, 3, 4, 5};

Elements →	x[0]	x[1]	x[2]	x[3]	x[4]
value →	1	2	3	4	5
Address →	1000	1002	1004	1006	1008

Lets us assume p is a pointer.

p = x;

p = & x[0];

Now, we can access every value of x using p++ to move from one element to another.

p = & x[0] = 1000

p+1 = & x[1] = 1002

p+2 = & x[2] = 1004

p+3 = & x[3] = 1006

p+4 = & x[4] = 1008

The address of an element is calculated using its index & the scale factor of the datatype.

address of x[3] = base address + (3 * scale factor of int)

= 1000 + (3 x 2)

= 1006

⇒ When handling arrays, instead of using array indexing, we can use pointers to access array elements.

· $*(p+3)$ gives the value of $x[3]$.

The pointer accessing method is much faster than array indexing.

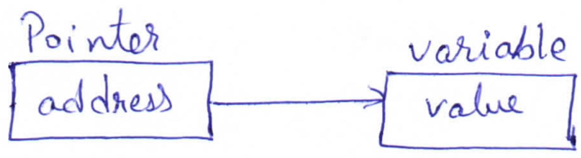
⇒ The same concept also applies to arrays of two or more dimensions.

for two-dimensional array:

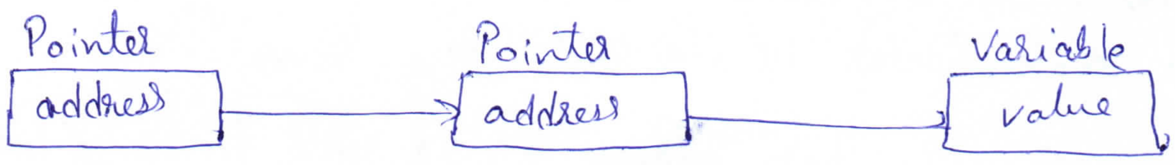
$a[i][j] = *((\text{basetype}*)\, a + (i * \text{rowlength}) + j)$

Multiple indirection

When a pointer is pointed to another pointer, thus creating a chain of pointers, then it is called as multiple indirection or pointers to pointers.



Single indirection



Multiple indirection

⇒ Here, the first pointer contains the address of the second pointer, which points to the object that contains the desired value.

⇒ A variable that is a pointer to a pointer must be declared using additional indirection operator symbol in front of the name.

Ex: int **p ;

This tells that p is a pointer to pointer of int type. Here p is not a pointer to integer, but a pointer to an integer pointer.

Program:

```
main()
{
 int x , *p1 , **p2;
 x=100;
 p1=&x;
 p2=&p1;
 printf(" %d ", **p2);
}
```

output

```
100
```

Problems with pointers

⇒ When a pointer is used incorrectly, or contains the wrong value, it can be a very difficult bug to find.

⇒ An erroneous pointer is difficult to find because the pointer, by itself, is not the problem, the trouble starts when you access an object through that pointer.

① The classic example of a pointer error is the uninitialized pointer. Consider this program:

```
int main(void)
{
 int x, *p;
  x=10;
 *p=x;        /* error, p is not initialized */
 return 0;
}
```

This program assigns the value 10 to some unknown memory location.

② A second common error is caused by a simple misunderstanding of how to use a pointer.
Consider the program:

```
#include <stdio.h>        /* This program is wrong */
 int main (void)
 {
  int x, *p;
   x = 10;
   p = x;
   printf (" %d", *p);
   return 0;
 }
```

The call to printf() doesnot print the value of x, i.e, 10.
It prints some unknown value because ....
p = x;   is wrong.
That statement assigns the value 10 to the pointer p.
But p should contain the address, not a value.
p = &x; is right..

③ Another error that sometimes occurs is caused by incorrect assumptions about the placement of variables in memory. For example,

```
char  s[80], y[80];
char  *p1, *p2;
 p1 = s;
 p2 = y;
 if (p1 < p2) ....
   ------
  ---------
```

is generally an invalid concept.

Making any comparisons between pointers that donot point to a common object may yield to unexpected results.

④ A related error results when you assume that two adjacent arrays may be indexed as one by simply incrementing a pointer accross the array boundaries.
For example,

```
int  first [10], second (10);
int *p, t;
 p= first ;
 for (t=0; t<20; ++t)
   *p++=t;
```

This is not good way to initialize the arrays first & second with the numbers 0 through 19.
It may work on some compilers, but it assumes that both arrays will be placed back to back in memory with first. This may not always be the case.

⑤ Other problems with pointers are like
* dangling pointers
* memory leaks
* Double deallocation / heap corrupting
* Aliasing.

Functions :-

A function is a block of code that performs a particular task.

(or)

A function is a group of statements that together perform a specific task.

Types of functions :-

C functions can be classified into two categories;

① Library functions

② User-defined functions

⇒ Library functions are pre-defined functions which are already defined in C library.

Ex: printf(), scanf(), strcat() etc.

You need to include appropriate header files to use these functions.

⇒ User-defined functions are those functions which are defined by the user at the time of writing program.

These functions are made for code reusability and for saving time & space.

Benefits of using functions:

* It provides modularity to your program's structure.
* It makes your code reusable. You just have to call the function by its name to use it, wherever required.
* In case of large programs with thousands of code lines, debugging & editing becomes easier if you use functions.
* It makes the program more readable & easy to understand.

Function Declaration:-

Syntax:

```
returntype funcname (parameter list);

              (or)

returntype funcname ( type1 parameter1, type2 parameter2, ....);
```

⇒ A function must be declared before its used.

⇒ Function declaration consists of 4 parts:

① return type
② function name
③ Parameter list
④ Terminating semicolon.

Returntype: When a function is declared to perform some sort of calculation or any operation & is expected to provide with some result at the end, then, a return statement is added at the end of function body.

⇒ Return type specifies the type of value like int, float, char, double etc.

⇒ If your function doesn't return any value, the return type would be **void**:

functionname: Function name is an identifier & it specifies the name of the function.

⇒ The function name is any valid C identifier & must follow same rules like other variables in C.

Parameter list: The parameter list declares the type & number of arguments that the function expects when it is called.

⇒ The parameters in the parameter list receives the argument values when the function is called.

⇒ They are often referred as formal parameters.

Terminating semicolon: It is mandatory to end a function declaration with a semicolon. It shows the difference between function declaration & function definition.

Function definition

Syntax:

```
returntype funcname (type1 parameter1, type2 parameter2, ----)
 {
   Body of the function;
 }
```

Function definition consists of 4 parts:

① return type
② function name
③ Parameter list
④ Body of the function.

Function body: It contains the declarations & the statements necessary for performing the required task. The body is enclosed within curly braces {----} & consists of 3 parts.

* local variable declaration (if required)
* function statements to perform the task inside function.
* a return statement to return the result evaluated by the function. (if it is void, then no return statement).

Note: In function definition, there is no semicolon(;) after the parenthesis.

Understanding the scope of functions:-

A scope is a region of a program. Variable scope is a region in a program where a variable is declared & used. So, we can have 3 types of scopes depending on the region.

(i) Local variables
(ii) Global variables
(iii) Formal parameters.

Local variables : Variables that are declared inside a function or a block are called local variables and are said to have local scope.
These are created when the control reaches the block or function contaning the local variables & then they get destroyed after that.

Program:

```
#include <stdio.h>
void func1()
{
  int x = 4;          /* local variable of func1 */
  printf("%d \n", x);
}
```

```
int main()
{
 int x=10;        /* local variable of main() */
  {
   int x=5;   /* local variable of ~~this~~block */
   printf("%d \n", x);
  }
  printf("%d \n", x);
  func1();
}
```

```
output
5
10
4
```

Global variables: Variables that are defined outside of all the functions & are accessible throughout the program are global variables & are said to have global scope. Once, declared, these can be accessed & modified by any function in the program.

Program:

```
# include <stdio.h>
 int x= 10;          /* global variable */
 void func1()
 {
   int x=5;          /* local variable */
   printf("%d \n", x);
 }
```

```
int main()
{
  printf("%d \n", x);
  func1();
}
```

```
output
10
5
```

Formal parameters: Formal parameters are the parameters which are used inside the body of a function. These are treated as local variables in that function and get a priority over the global variables.

Program:

```
#include <stdio.h>
 int x=10;              /* global variable */
 void func1(int x)
 {                  /* x is a formal parameter */
  printf("%d \n", x);
  }
 int main()
 {
    func1(5);
 }
```

```
output
5
```

Scope rules:

Standard c defines four scopes that determine the visibility of an identifier. They are as below.

(i) file scope
(ii) Block scope
(iii) function prototype scope
(iv) function scope.

File scope:-

* It starts at the begining of the file & ends with the end of the file.
* It refers only to those identifiers that are declared outside of all functions.
* File scope identifiers are visible throughout the entire file.
* Variables that have file scope are global.

Block scope:-

* Begins with the opening { of a block & ends with its associated closing }.
* Block scope extends to function parameters in a function definition.
* Function parameters are included in a function's block scope.
* Variables with block scope are local to their block.

Function prototype scope:-

* Identifiers declared in a function prototype; visible within the prototype.

Function scope:

* Begins with the opening { of a function & ends with its closing }. Function scope applies only to labels.
* A label is used as the target of a goto statement, & that label must be within the same function as the goto.

Type qualifiers:-

The keywords which are used to modify the properties of a variable are called type qualifiers.

There are two types of qualifiers available in C language.

They are:
(i) Const
(ii) Volatile

Const:

* Constants are like normal variables. But, only difference is, their values can't be modified by the program once they are defined.
* They refer to fixed values. They are also called as literals.
* They may belong to any of the datatype.

Syntax:

```
const datatype variablename;   (or)
const datatype *variablename;
```

Ex: `const int classsize = 40;`

Volatile :-

* When a variable is defined as volatile, the program may not change the value of the variable explicitly.
* But, these variable values might keep on changing without any explicit assignment by the program. These types of qualifiers are called volatile.

Syntax:

```
volatile datatype variablename;     (or)
volatile datatype *variablename;
```

Ex: volatile int date;

Here, the value of date may be altered by some external factors. But when we declare as volatile, the compiler will examine the value of the variable each time it is encountered.

⇒ The value of a variable declared as volatile can be modified by its own program as well.
If the value must not be modified by the program while it may be altered by some other process, then we may declare the variable as both const and volatile.

Ex: volatile const int location = 100;

Function arguments

Functions are called by their names. If function doesnot have any arguments, then to call a function, its name is directly used.
But for functions with arguments, its called in two different ways, based on how we specify the arguments.
They are: (i) Call by value
(ii) Call by reference.

Call by value:-

* Calling a function by value means, we pass the value of the arguments which are stored or copied into the formal parameters of the function.
* The original values are unchanged only the parameters inside the function changes.

Program:

```
#include <stdio.h>
void swap (int x, int y)
int main()
{
 int a = 100;
 int b = 200;
printf("before swapping  a = %d ", a);
printf ("Before swapping  b = %d ", b);
swap (a, b)                    /* call by value */
printf (" after swapping a = %d ", a);
printf ("after swapping b = %d ", b);
return 0;
}
void swap (int x, int y)
{
int temp;
temp = x;
 x = y;
 y = temp;
return;
}
```

output

```
Before swapping  a=100
Before swapping  b=200
After swapping  a=100
After swapping  b=200
```

Call by reference:-

* In call by reference we pass the address (reference) of a variable as argument to any function.
* When we pass the address of any variable as argument, then the function will have access to our variable, as it now knows where it is stored & hence can easily update its value.

* In this case the formal parameter can be taken as a reference or a pointer, in both the cases they will changes the values of the original variable.

Program:-

```
#include <stdio.h>
 void swap(int *x, int *y);
 int main()
 {
 int a=100;
 int b= 200;
 printf (" Before swapping  a=%d", a);
 printf (" Before swapping  b=%d", b);
 swap (&a, &b);           /* call by reference */
 printf (" After swapping  a=%d", a);
 printf (" After swapping  b=%d", b);
 return 0;
}
void swap (int *x, int *y)
{
 int temp;
 temp= *x;
 *x = *y;
 *y= temp;
 return 0;
}
```

output

```
Before swapping a=100
Before swapping b=200
After swapping a=200
After swapping b=100
```

Passing single dimensional arrays to functions:-

In C programming, a single array element or an entire array can be passed to a function.

Passing single array element in function:-

Single element of ~~an~~ array can be passed in similar manner as passing variable to a function.

Program:

```
# include <stdio.h>
void display (int age)
{
 printf ("%d", age);
}
int main()
{
  int ageArray[]= { 2, 3, 4 };
  display (ageArray [2]);       /* passing array element
                                     ageArray [2] only */
 return 0;
}
```

```
output

4
```

Passing an entire one-dimensional array to a function:-

While passing arrays as arguments to the function, only the name of the array is passed (i.e., starting address of memory area is passed as argument.)

Program:

```
#include <stdio.h>
float average (float age[]);
int main()
{
  float avg, age[] = {23.4 , 55 , 22.6 , 3 ,40.5, 18};
  avg = average (age);
  printf (" average age = %f ", avg);
  return 0;
}
float average (float age[])
{
  int i;
  float avg, sum = 0.0;
  for (i=0 ; i< 6 ; ++i)
  {
     sum + = age [i];
  }
  avg = (sum /6);
  return avg;
}
```

Output

```
Average age = 27.08
```

Variable length arrays:-

You can declare an array whose dimensions are specified by any valid expression, including those whose value is known only at run time. This is called a variable-length array.
However, only local arrays (i.e., those with block scope or prototype scope) can be of variable length.
Here is an example of a variable length array:

```
void f (int dim)
{
   char str [dim];        /* a variable length character array*/
     /* ------ */
}
```

Here, the size of str is determined by the value passed to f() in dim. Thus, each call to f() can result in str being created with a different length.
Variable-length arrays supports numeric processing.

C's Dynamic Allocation functions :-

The process of allocating memory at run time is known as dynamic memory

There are four memory allocation functions. They are :

(i) malloc
(ii) calloc
(iii) free
(iv) realloc

(i) malloc :- It allocates requested size of bytes and returns a pointer to the first byte of the allocated space.

⇒ A block of memory may be allocated using the function malloc.

⇒ It reserves a block of memory of specified size & returns a pointer of type void. This means we can assign it to any type of pointer.

Syntax:

```
ptr = (cast-type *) malloc (byte-size);
```

Ex:

```
x = (int *) malloc (100 * sizeof (int));

cptr = (char * ) malloc (10);
```

Program:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
int main()
{
 int *ptr;
 ptr = (int *)malloc(sizeof (int));
 if(ptr == NULL)
 {
 printf ("couldnot allocate memory \n");
 exit (-1);
 }
  else
  printf ("Memory allocated successfully ");
  free (ptr);
 }
```

(ii) calloc:- It allocates for an array of elements, initializes them to zero & then returns a pointer to the memory.

⇒ calloc allocates multiple blocks of storage, each of the same size, & then sets all bytes to zero.

⇒ It allocates contiguous space for n blocks, each of size elementsize bytes.

⇒ All bytes are initialized to zero & a pointer to the first byte of the allocated region is returned.

⇒ If there is not enough space, a NULL pointer is returned.

Syntax: ptr = (cast-type *) calloc (n, elementsize);

Program:-

```
#include <stdio.h>
#include <conio.h>
# include <stdlib.h>
int main()
{
 int *ptr;
 ptr = (int *) calloc (10, size of (int));
 if (ptr= =NULL)
 {
 printf("couldnot allocate memory \n");
 exit (-1);
 }
else
 printf(" memory allocated successfully ");
 free (ptr);
}
```

(iii) realloc ;– It modifies the size of previously allocated space.

⇒ Sometimes, the previously allocated memory is not sufficient & we need additional space for more elements.

⇒ It is also possible that the memory allocated is much larger than necessary & we want to reduce it.

⇒ In both the cases, we can change the memory size already allocated with the help of the function realloc.

This process is called the reallocation of memory.

Syntax:

```
ptr = malloc (size);
ptr = realloc (ptr, newsize);
```

Program:–

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main()
{
 int *ptr , *ptr1;
 ptr = (int *) malloc (sizeof (int));
ptr1 = (int *) realloc (ptr ,100 * sizeof (int));
 if (ptr==NULL)
  {
  printf ("\n Exit ");
  free (ptr);
exit (0);
}
else
{
free (ptr);
ptr=ptr1;
}
}
```

(iv) free :- Frees previously allocated space.

⇒ In dynamic run time allocation, it is important to release the space when it is not required.

⇒ The release of storage space becomes important when the storage is limited.

⇒ when we no longer need the data we stored in a block of memory. & we do not inted to use that block for storing any other information, we may release that block of memory for future use, using free function.

Syntax: free (ptr);

Program: Any of the above program.

C-Storage class specifiers

Storage class specifiers in C-language tells the compiler where to store the variable, how to store the variable, what is the initial value of the variable & life time of the variable.

Syntax:-

```
storage-specifier datatype variablename;
```

There are 4 storage class specifiers. They are:

(i) auto
(ii) extern
(iii) static
(iv) register

(i) auto:-

Storage place : CPU memory
initial/default value : Garbage value
Scope : local
Lifetime : within the function only.

⇒ The auto variables are like local variable.
All local variables are auto by default.

Program:-

```
#include <stdio.h>
# include <conio.h>
void increment(void);
int main()
{
  increment();
  increment();
  increment();
  increment();
  return 0;
}
void increment (void)
{
 auto int i=0;
 printf ("%d", i);
 i++;
}
```

output

```
0 0 0 0
```

(ii) static:-

storage space: CPU Memory.
initial/default value: Zero
Scope: local
Life: Retains the value of the variable between different function calls.

Program:-

```
#include <stdio.h>
void increment (void);
int main()
{
 Increment ();
 increment ();
 increment ();
 increment ();
 return 0;
}
 void increment (void)
 {
 static int i=0;
 printf ("%d", i);
 i++;
 }
```

output

```
0 1 2 3
```

(iii) extern:-

storage place:- CPU memory

initial/default value:- zero

Scope:- Global

Lifetime:- Till the end of the main program. Variable definition might be anywhere in the C program.

⇒ Extern variable is equivalent to global variable. Its definition might be anywhere in the C program.

Program:

```
#include <stdio.h>
int x=10;
int main()
 {
  extern int y;
  printf (" The value of x is %d \n", x);
  printf (" The value of y is %d ", y);
  return 0;
 }
 int y=50;
```

output

```
The value of x is 10
The value of y is 50
```

(iv) register:-

storage place: Register memory

initial/default value: Garbage value.

Scope : local

Lifetime : Within the function only.

⇒ Register variables will be accessed very faster than the normal variables since they are stored in register memory rather than main memory.

⇒ But only limited variables can be used as register since register size is very low. (16 bits, 32 bits or 64 bits).

<u>Program:-</u>

```
# include <stdio.h>
int main()
{
 register int i;
  int arr[5];
   arr[0] = 10;
   arr[1] = 20;
   arr[2] = 30;
   arr[3] = 40;
   arr[4] = 50;
   for (i=0; i<5; i++)
    {
      printf ("value of arr[%d] is %d \n", i, arr[i]);
    }
   return 0;
}
```

output

```
value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
```