Pointers and Dynamic Memory Allocation

What is Pointer,

Declaration and Initialization Pointer Variables,

Accessing a Variable using Pointer ,

Chain of Pointers (Pointer to pointer),

Pointer Expressions (Pointer Arithmetic),

Pointer Increments and Scale Factor,

Arrays and Strings using Pointers,

Arrays of Pointers,

Concept of Dynamic Memory Allocation,

Allocating Single and Multiple Block of Memory,
Altering the Size of a Block .

INTRODUCTION TO POINTERS

- **Pointer:** It is a derived data type in C.
  - It is built from one of the fundamental data types.
  - Contains memory addresses as their value. Used to access and manipulate data stored in the memory.

- **Pointer variable :** Variable that hold memory addresses as their values.

- Consider the following statement:

```
int  i = 125;

   OR

int  i;

i = 125;
```

This statement says the compiler to find a address for the integer variable "i" and puts the value 125 in that location. Assume that the address is 100. We represent a variable as follows:

i ----------> Name of a variable

125	----------> Value of a variable

100 ----------> Address of a variable

Now, we may access to the value 125 by two ways,

- A name of variable itself (here i)
- An address of a variable (here 100)

BENEFITS OF POINTERS

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

ACCESSING THE ADDRESS OF A VARIABLE

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately.

The '**address of operator' &** immediately preceding a variable returns the address of a variable associated with it .

for example,

p = **&quantity;**

would assign the address 5000 (the location of **quantity)** to the variable p.

The **&** operator can be used only with a simple variable or an array element.
The following are **illegal (invalid)** use of address operator:
1. &125 (pointing at constants).
2. int x[10];
   &x (pointing at array names) .
3. &(x+y) (pointing at expressions).

If x is an array, then expressions such as

&x[0] and &x[i+3]

are **valid** and represent the addresses of 0th and (i+3)th elements of x.

DECLARATION OF POINTER VARIABLES

The declaration of a pointer variable (General form):

data_type *pt_name;

This tells the compiler three things about the variable pt_name.
1. The asterisk (*) tells that the variable pt_ name is a pointer variable.
2. pt_name needs a memory location.
3. pt_name points to a variable of type *data_type.*

integer pointer

```
int *p;
```

declares the variable p as a pointer variable that points to an integer data type. The type **int** refers to the data type of the variable being pointed to by p and not the type of the value of the pointer.

float pointer

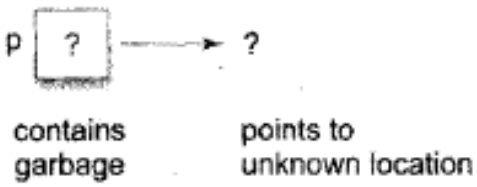
```
float *x ;
```

declares x as a pointer to a floating point variable.

The declarations cause the compiler to allocate memory locations for the pointer variables p and x.
Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown: .

```
int *p;
```

int *p;



Pointer variables are declared similar to normal variables except for the addition of the unary * operator. This symbol can appear anywhere between the type name and the printer variable name. Programmers use the following styles:

```
int*    p;    /* style 1 */
int     *p;   /* style 2 */
int   * p;    /* style 3 */
```

However the style2 is becoming increasingly popular due to the following reasons:

1. This style is convenient to have multiple declarations in the same statement. Example:

   ```
   int *p, x, *q;
   ```

2. This style matches with the format used for accessing the target values. Example:

   ```
   int x, *p, y;
   x = 10;
   p = & x;
   y = *p;          /* accessing x through p */
   *p = 20;         /* assigning 20 to x */
   ```

   We use in this book the style 2, namely,

   int *p;

INITILIZATION OF POINTER VARIABLES

The process of assigning the address of a variable to a pointer variable is known as ***initialization.***

All uninitialized pointers will have some unknown values that will be interpreted as memory addresses. They may not be valid addresses or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous results. It is therefore important to initialize pointer variables carefully before they are used in the program.

Once the pointer variable has been declared :
The assignment operator to initialize the pointer variable. (Valid)
Example:

```
int quantity;
int *p;                    /* declaration */

p = &quantity;             /* initialization */
```

We can also combine the initialization with the declaration. **(Valid)**
Example,

```
int quantity;
int *p = &quantity;
```

The only requirement here is that the variable **quantity** must be declared before the initialization takes place. This is an initialization of p and not *p.

The pointer variables always point to the corresponding type of data.
For example,

```
float a, b;
int x, *p;
p = &a;                    / * invalid */
b = *p;
```

will result in erroneous output because we are trying to assign the address of a **float** variable to an integer pointer.

When we declare a pointer to be of **int** type, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and initilization of the pointer variable in one step. (Valid)
For example,

```
int x, *p = &x;
```

is **valid.**

It declares x as an integer variable and p as a pointer variable and then initializes to the address of x. The variable x is declared first.
The following statement is **invalid :**

```
int *p = &x, x;
```

We could also define a pointer variable with an initial value of NULL or 0 (zero).
That is, the following statements are valid

```
int *p = NULL;
int *p = 0;
```

With the exception of NULL and 0, **no other constant value can be assigned to a pointer variable.**
For example, the following is invalid:

```
int *p = 5360;
```

ACCESSING A VARIABLE THROUGH ITS POINTER

Indirection operator : Value of the variable can be accessed through pointer by using another unary operator *** (asterisk)** known as **indirection operator** or **dereferencing operator**.

Consider the following statements:

```
int quantity, *p, n;
quantity = 179;
p = &quantity;
n = *p;
```

The following statements are valid:

```
p = &quantity;
n = *p;
```

are equivalent to

```
n = *&quantity;
```

which in turn is equivalent to

```
n = quantity;
```

Pointers and function arguments

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of the variables is known as "call by reference". The process of calling a function using actual values of the variable is known as "call by value".

Example : program to change value of variable using concept of pointer in functions.

```
main( )
{
    int x;
    x=20;
   change(&x);
   printf("%d",x);
}
```

```
change(int *p)
 {
       *p=*p + 10;
 }
```

The above code will add value 10 to the value stored at address p. Thus x will be 30.

Returning multiple values through pointers

Example : Program to interchange value of two integer variables using pointers.

```
void change (int *, int *)                /* prototype */
main( )
{
     int x,y;
     x=100;
     y=200;
     printf("Before interchange : x = %d y = %d", x , y);
     change(&x,&y);
     printf("After interchange : x = %d y = %d", x , y);
}

change(int *a, int *b)

{

    int t;

     t = *a;

    *a = *b;

    *b = t;

}
```

Output :

Before interchange : x = 100 y = 200

After interchange : x =200 y = 100

DYNAMIC MEMORY ALLOCATION

Compile time allocation : The process of allocation memory at compile time is known as a compile time memory allocation.

If the judgment of allocated size is wrong, then it may cause failure of program or wastage of storage space.

Dynamic memory allocation

Definition: "The process of allocation memory at runtime is known as a dynamic memory allocation ". It is the ability to calculate and assign memory space required by the variables in the program during execution time.

This can be done by memory management / allocation functions.

They are:

1. malloc
2. calloc
3. free
4. realloc

1. malloc : It allocates single block of storage space. A block of memory can be allocated using this function. It reserves a block of memory of specified size and returns a pointer of type **void**. Initial value is **garbage.**

The general form of malloc is :

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

where, **ptr** is a pointer of specified data type.

malloc is a function that allocates memory

size is an integer number

Example : **To allocate memory for 100 integers.**

ptr = (int *) malloc (100 * sizeof (int));

On successful execution of this statement, a memory space equivalent to 100 times the size of int(2) byte is reserved, and address of the first byte of the memory allocated is assigned to the pointer **x** of type integer.

Example :

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

allocates 10 bytes of space for the pointer **cptr** of type **character.**

The malloc function is also used to allocate the memory for complex data types like structure.

If the space does not satisfy the requirement, it fails and returns NULL value.

2. calloc : It allocates **multiple** block of storage, each of the same size, and then set all bytes to size. It is used for requesting memory space at runtime for storing derived data types such as array and structure. It initilizes all memory to **zero.**

The general form of calloc is :

ptr = (cast - type *) calloc (n, elem - size);

where **ptr** is a pointer of specified data type.

It allocates contiguous space for n blocks, each of size 'elem-size' bytes.

Example : to allocate memory for 10 characters

```
cptr = (char * ) calloc (10,1);
```

Example : to allocate memory for 100 integers

```
cptr = (int * ) calloc (100,2);
```

Example : to allocate memory for 100 integers

```
cptr = (int * ) calloc (100,sizeof(int));
```

If the space does not satisfy the requirement, it fails and returns NULL values.

3. free :

It is used to release the block of memory, which has been created by malloc or calloc function.

It is user responsibility to release the space, when it is not required (if created by malloc / calloc). The release of memory space becomes important when the storage is limited.

The general form of free function is

free(ptr);

where **ptr** is a pointer to a memory block which is created by malloc or calloc.

For Example : **Free 20 bytes of allocated memory, for 10 integers.**

```
main()
{
    int *ptr;
    ptr = (int *) malloc (20);
    printf("memory size : %d", ptr);
    free(ptr);
}
```

4. realloc :

It is used to change (increase / decrease) the memory size already allocated.

Syntax: allocation statement

```
ptr = malloc (size);
```

Syntax: reallocation statement

ptr = realloc (ptr, newsize);

For example,

If the original allocation is done by the following statement:

ptr = (int *) malloc (10 * sizeof (int));

then, the reallocation of space can be done as:

ptr = realloc (ptr, 300);

This function allocates a new memory space of size 300 to the pointer variable ptr, and returns a pointer to the first byte of the new memory block. The new memory block may or may not be same as old block.

If the function is unsuccessful in allocating space, it returns the Null value, and the original block is lost.

POINTERS AND ARRAYS (one dimensional integer & float)

When array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.

The base address is the location of the first element (index 0) of the array. The compiler also defines array name as a constant pointer to the first element.

Suppose we declare an array x as follows:

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

Suppose the base address of x is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows:

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 to the first element. Therefore, value of **x** is 1000 accessed by **x[0].** That is,

x = &x[0]=1000

If p is an integer pointer,

p = x;

This is equivalent to

p = &x[0] ;

Suppose p is integer pointer

p=x;

p=&x[0]=1000

p+1=&x[1]=1002

p+2=&x[2]=1004

Program : To print elements of an array using pointer.

```
main()
{
   int  a[10], *pa , i;

   for(i=0 ; i<10 ; i++)
   {
      printf("Enter array element: ");
      scanf(%d",&a);
   }
   pa=a;
   for(i=0 ; i<10 ; i++)
   {
      printf("\n Value of array Element %d is : %d",i+1,*pa);
      pa++;
   }
```

POINTER ARITHMETIC

Pointer variables can be used in expressions.
For example, if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
y = *p1  *  *p2;
sum = sum + *p1;
*p2 = *p2  + 10;
```

***z* = 5* - *p2/ *p1** is **invalid.**

The symbol /* is considered as the beginning of a comment and therefore the statement fails.

Integers can be added or subtracted from pointers
following statements are **valid.**

```
p1= p1 + 4 ;
p2 = p1 – 1 ;
```

pointer can be subtracted from another pointer

p1 – p2 is **valid.**

If p1 and p2 are both pointers to the same array, then p2 – p1 gives the number of elements between p1 and p2.

We may also use short-hand operators with the pointers.
following statements are **valid.**

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

pointers can also be compared using the relational operators, **[Valid]**

p1 > p2, p1==p2 and p1!=p2 are are allowed.

Following statements are invalid :

Pointer cannot be divided by another pointer : p1/p2

Pointer cannot be multiplied by another pointer : p1 * p2

Pointer cannot be divided by a constant : p1/3

Pointer cannot be added to another pointer : p1 + p2

POINTER INCREMENTS AND SCALE FACTOR

Following statements are valid:

```
p1 = p2 + 3 ;
p1= p1 + 1 ;
p1++;
```

all the above statements will cause pointer p1 to point to the next value of its type.

For example , if **p1** is an integer pointer with initial value say **1000**, then after execution of statement p1=p1+1 , value of p1 will be **1002** and not **1001** because the statement will cause pointer to increment its value by length of data type it points to. This length is called scale factor.

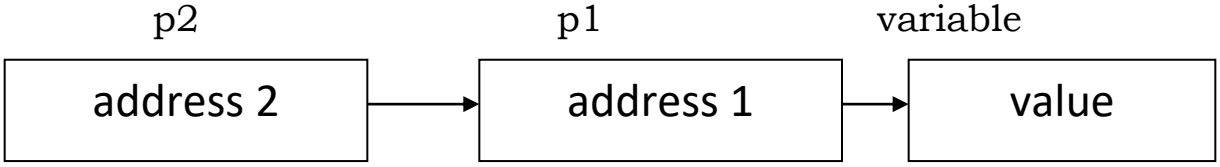
Length of various datatypes are:

characters	1 byte
integers	2 bytes
floats	4 bytes
long integers	4 bytes
doubles	8 bytes

The number of bytes used to store various data types depends on the system and can be found by making use of the **sizeof** operator. For example, if x is a variable, then sizeof(x) returns the number of bytes needed for the variable.

POINTER TO POINTER (CHAIN OF POINTERS)

It is possible to make pointer point to another pointer, thus creating chain of pointers.



The pointer variable **p2** contains the address of the pointer variable **p1**, which points to the location that contains the desired value. This is known as ***multiple indirections.***

A variable that is pointer to pointer is declared using additional indirection operator symbols in front of the name.

Example : int **p2;

It tell that p2 is the pointer to pointer of **int** datatype. p2 is not a pointer to an integer, but rather a pointer to an integer pointer.

Example :

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

Output : 100
p1 is pointer to an integer and p2 is pointer to an integer pointer.

Differentiate between "*" and "&" Operator.

	* Operator		& Operator
1.	It is used to access the value of the variable using pointer. This operator is * known as the indirection operator.	1.	It is used to access the address of a variable.
2.	The * can be remembered as "value at address" operator.	2.	The & can be remembered as "address of" operator.
3.	It is also known as dereferencing operator.	3.	It is known as Address of operator.

4.	Example:
	int x,*p;
	x=10;
	p=&x;
	printf("value of x is : %d",*p);
	Output:
	value of x is : 10
		4.	Example:
			int x,*p;
			x=10;
			p=&x;
			printf("Address value of x is %u",p);
			
			Address value of x is : 4001

Differentiate between malloc and calloc

	malloc		calloc
1.	It allocates single block of memory.	1.	It allocates multiple block of memory.
2.	Does not initilizes memory allocated. So value is garbage value.	2.	Initilizes memory allocated to zero value.
3.	Requires one agument. 1. Total memory required in bytes.	3.	Requires two aguments. 1. Total elements to be stored. 2. No. of bytes.
4.	General Form : ptr = (cast - type *) malloc (byte - size);	4.	General Form : ptr = (cast - type *) calloc (n, elem - size);
5.	Example: to allocate memory for 100 integers. px = (int *) malloc (100 * 2);	5.	Example: to allocate memory for 100 integers. px = (int *) calloc (100 , 2);

RULES OF POINTER OPERATIONS [Extra Reading]

The following rules apply when performing operations on pointer variables.

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
5. An integer value may be added or subtracted from a pointer variable.
6. When two pointers point to the same array, one pointer variable can be subtracted from another.
7. When two pointers point to the objects of the same data types, they can be compared using relational operators.
8. A pointer variable cannot be multiplied by a constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an arbitrary address (i.e &x = 10; is illegal).

Programs and Examples for reference : [Extra Reading]

1. **Write a program to. print the address of a variable along with its value**

```
#include <stdio.h>

main ()
{
  int a;
  int *pa;
  a = 10;
  pa = &a;

   printf (" The address of a is %u \n", pa);
   printf (" The value at that location is %d\n", a);
   printf (" The value at that location is %d\n", *pa);
   *pa = 50;
   printf ("The value of a is %d\n", a);
}
```

```
The address of a is   631672
 The value at that location is 10
 The value at that location is 10
The value of a is 50
```

2. Write a program to illustrate the use of indirection operator,*, to access the value pointed to by a printer. [Extra Reading]

Program

```
main()
{
   int   x, y , *ptr;
   x = 10;
   ptr = &x;
   y = *ptr;
   printf("Value of x is %d\n\n",x);
   printf("%d is stored at addr %u\n", x, &x);
   printf("%d is stored at addr %u\n", *&x, &x);
   printf("%d is stored at addr %u\n", *ptr, ptr);
   printf("%d is stored at addr %u\n", y, &*ptr);
   printf("%d is stored at addr %u\n", ptr, &ptr);
   printf("%d is stored at addr %u\n", y, &y);
   *ptr = 25;
   printf("\nNow x = %d\n",x);

}
```

Output

```
Value of x is 10
```

10 is stored at addr 4104

10 is stored at addr 4104

10 is stored at addr 4104

10 is stored at addr 4104

4104 is stored at addr 4106

10 is stored at addr 4108

Now x = 25

The statement ptr = &x assigns be address of x to ptr and y = *ptr assigns the value pointed to by the pointer ptr to y.

Note the use of the assignment statement

*ptr = 25;

This statement puts the value of 25 at the memory location whose address is the value of ptr.

3. ILLUSTRATION OF POINTER EXPRESSIONS

(pointer Arithmetic) [Extra Reading]

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("\n");
    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("\na = %d, b = %d,", a, b);
    printf(" z = %d\n", z);
}
```

Output

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