Pointer

So before knowing what pointer is let us know about memories

- All computer have primary memory, also known as **RAM**(Random Access Memory)

- RAM holds the programs that the computer is currently running along with the data(i.e. variables) they are currently manipulating.

- All the variables used in a program reside in the memory when the program is executed.

- RAM is divided into a number of small units or locations and each location is represented by some unique number known as memory address.

So on…

- Each memory location is capable of storing small number, which is known as byte.

- A char data is one byte is size and hence needs one memory location of the memory.

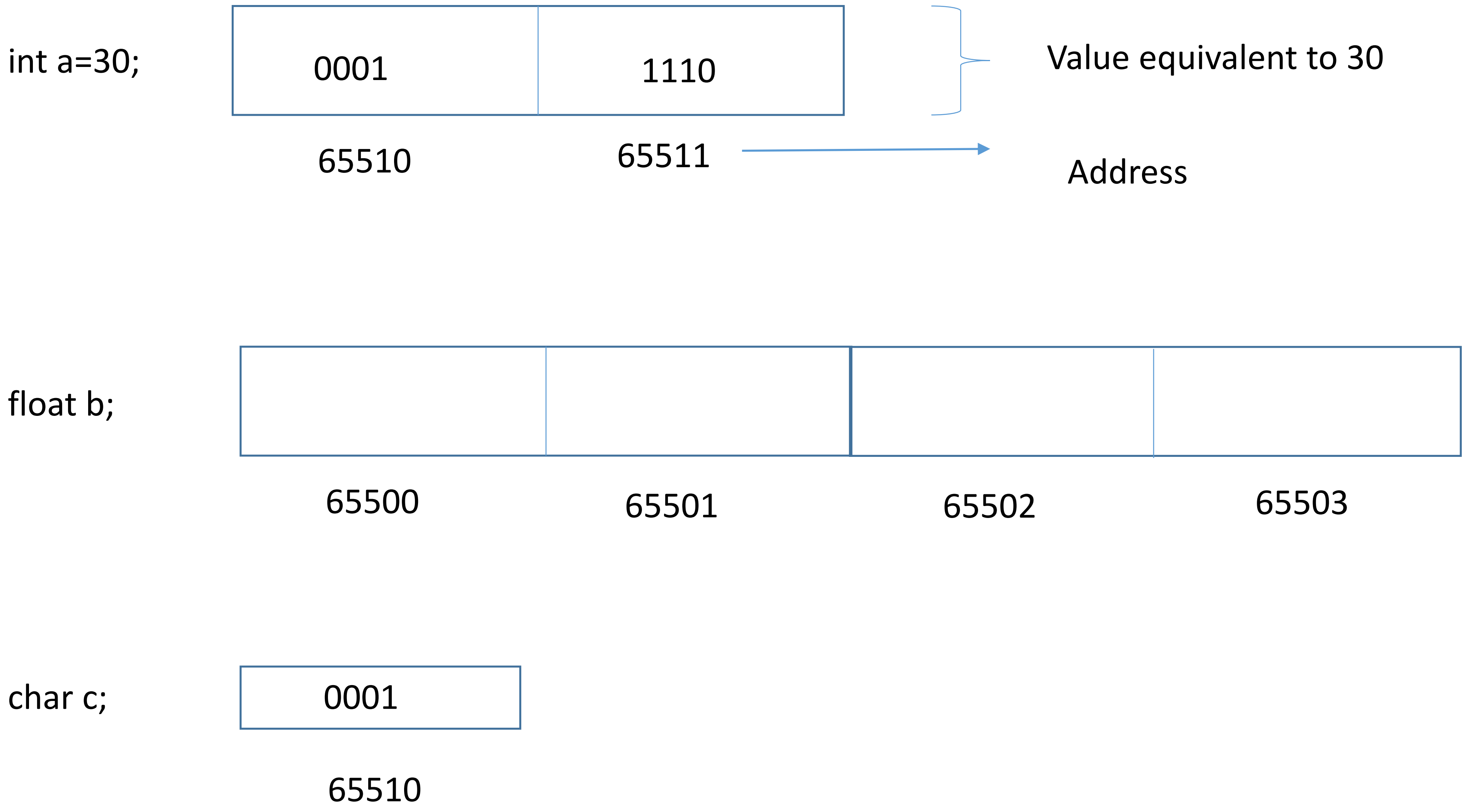
- Similarly, integer data is two byte in size and hence needs two memory locations of the memory.

Key concept

8 bits	1 byte ex. 1001 1111 is one byte
1024 bytes	1 kilo byte (KB)
1024 KB	1 Mega Bytes (MB)
1024 MB	1 Giga Bytes (GB)

- A computer having 1 GB RAM has 1024*1024*1024 i.e. 1073741824 bytes and these 1073741824 bytes are represented by 1073741824 different address.
- For ex. The memory address 65524 represents a byte in memory and it can store data of one byte.

- Every variable in C program is assigned a space in memory.
- When a variable is declared, it tells computer the type of variable and name of the variable.
- According to the type of variable declared, the required memory locations are reserved.
- For ex. int requires two bytes, float requires four bytes and char requires one byte.


int a=30;
0001
1110
Value equivalent to 30
65510
65511
Address
float b;
65500
65501
65502
65503
char c;
0001
65510

A program to display memory location reserved by a variable

```
#include <stdio.h>
int main(){
        int a=20;
        printf("The address of a is: %u\n", &a);
        printf("The value of a is: %d\n", a);
        return 0;
}
```

<u>OUTPUT:</u>

```
The address of a is: 65524
The value of a is: 20
```

A program to illustrate address reserved by different data types

```
#include <stdio.h>
int main()
{
        /* code */
        int a=20, b=50;
        float c=50.4;
        char d='A';
        printf("The Base Address of a is: %u\n", &a);
        printf("The Base Address of b is: %u\n", &b);
        printf("The Base Address of c is: %u\n", &c);
        printf("The Base Address of d is: %u\n", &d);
        return 0;
}
```

Pointer

- A pointer is a variable that contains a memory address of variable.
- A pointer variable is declared to some type, like any other variable.
- Each pointer variable can point only to one specific type
- Pointer is declared in the same fashion like other variables but is always preceded by '*' (asterisk operator.)

- integer variable declared as:
  - int a;
  - Here a is considered as integer variable.
- Similarly,
  - int * a;
  - Now variable a is a pointer variable, it now can store address of integer variable.
  - The address of float variable can not be stored in it.

- Valid Example

```
int  *p;
int num;
p=&num;
```

- Invalid Example

```
int  *p;
float num;
p=&num;
```

Pointer Declaration

- Pointer variable can be declared as follows:

SYNTAX:

```
data_type  *  variable_name;
```

- Example:

```
int *x;          //x integer pointer, holds address of any int variable
float *y;        //y integer pointer, holds address of any float variable
char *z;         //z integer pointer, holds address of any char variable
```

```
#include <stdio.h>
int main()
{
        /* code */
        int v=10, *p;
        p=&v;
        printf("address of v=%u \n",&v );

        printf("address of v=%u \n",p );

        printf("value of v=%d \n",v );
        printf("value of v=%d \n",*p );

        printf("address of p=%u \n",&p );

}
```

Explanation

- v is an integer variable with 10.
- p is a pointer variable
- p=&v assigns address of v to p variable. i.e ‘p’ is the pointer to variable ‘v’
- To access the address and value of v, pointer p can be used.
- The value of p is nothing but address of the variable v.
- *p can be used to display value stored at a location pointed by pointer variable p.

Indirection or Dereference Operator

- The operator *, used in front of a variable, is called pointer or indirection or dereference operator.

- Normal variable provides direct access to their own values whereas a pointer provides indirect access to the values of the variable whose address it stores.

- When the pointer is declared, the star indicates that it is a pointer, not a normal variable.

- The indirection operator indicates "the value at the memory location stored in the pointer" or "the content of the location pointed by pointer variable"

Address Operator

- The operator & is known as address operator.
- &a denotes the address of variable a.

- Ex.

```
int a=10, *p;
p=&a;
```

Initializing Pointer

- Address of some variable can be assigned to a pointer variable at the time of declaration of the pointer variable.

- For ex:

```
int num;
int *ptr=&num;
```

```
int num;
int *p;
p=&num;
```

- These two statements above are equivalent to following statements.

So what actually is bad pointer?

- When a pointer is first declared, it doesn't have a valid address.
- Each pointer must be assigned a valid address before it can support dereference operators. Before that, the pointer is bad and must not be used.
- Every pointer contains garbage value before assignment of some valid address.
- Correct code overwrites the garbage value with a correct reference to an address and thereafter the pointer works fine.
- We have to program carefully.

Void Pointer

- Void pointer is a special type of pointer.
- It can point to any data type, from an integer value to a float to a string of characters.
- Using void pointer, the pointed data can not be referenced directly (i.e. * operator can not be used on them)
- Type casting or assignment must be used to change the void pointer to a concrete data type to which we can refer.

Void Pointers in C : Definition

- Suppose we have to declare integer pointer, character pointer and float pointer then we need to declare 3 pointer variables.
- Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as integer pointer, character pointer.
- Declaration of Void Pointer :

```
void * pointer_name;
```

Void Pointer Basics

- In C General Purpose Pointer is called as void Pointer.
- It does not have any data type associated with it
- It can store address of any type of variable
- A void pointer is a C convention for a raw address.
- The compiler has no idea what type of object a void Pointer really points to

example

```
void *ptr;    // ptr is declared as Void pointer


char cnum;

int inum;

float fnum;


ptr = &cnum;  // ptr has address of character data

ptr = &inum;  // ptr has address of integer data

ptr = &fnum;  // ptr has address of float data
```

Explanation

- **Void pointer** declaration is shown above.
- We have declared 3 variables of integer, character and float type.
- When we assign **address of integer** to the void pointer, pointer will become Integer Pointer.
- When we assign **address of Character** Data type to void pointer it will become Character Pointer.
- Similarly we can assign address of any data type to the void pointer.
- It is capable of storing address of any data type

Summary : Void Pointer

Scenario	Behavior
When We assign address of integer variable to void pointer	Void Pointer Becomes Integer Pointer
When We assign address of character variable to void pointer	Void Pointer Becomes Character Pointer
When We assign address of floating variable to void pointer	Void Pointer Becomes Floating Pointer

example

```
#include <stdio.h>
int main(void){
	int a=10;
	double b=4.5;
	void *vptr;

	vptr=&a;
	printf("a=%d\n", *((int *)vptr)); /* not just simply *vptr */
	vptr=&b;
	printf("\nb=%lf\n", *((double *)vptr));
}
```

NULL POINTER

- A null pointer is a special pointer value that points nowhere or nothing. i.e. no other valid pointer to any variable or array cell or anything else will ever be equal to a null pointer.

- We can define a null pointer using predefined constant NULL which is defined in header files such as stdio.h, stdlib.h,string.h

- Ex:

```
int *ptr=NULL;
```

POINTER TO POINTER (Double Pointer)

- C allows the use of pointers that point to other pointers and these in turn, point to data.

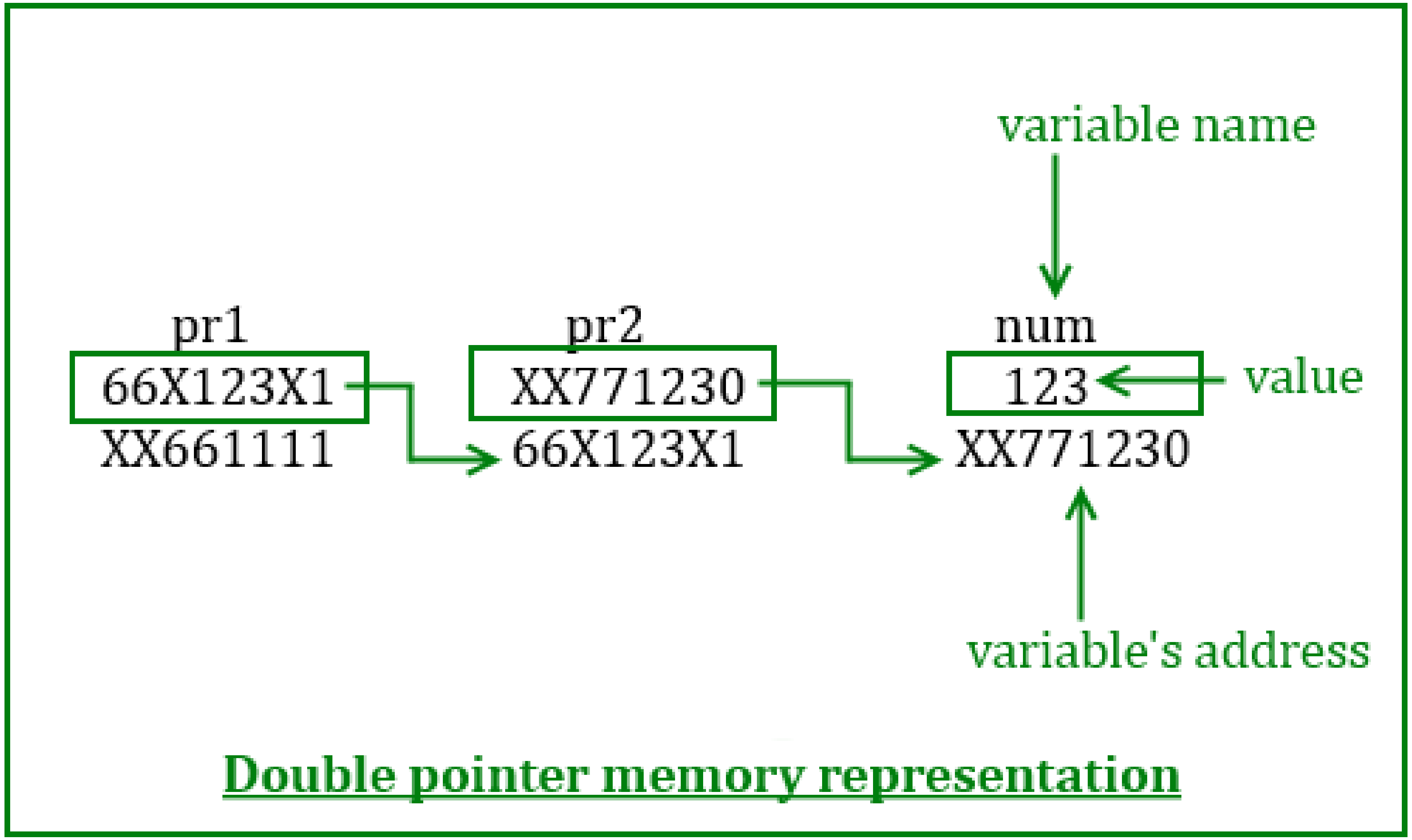
- For pointers to do that, we only need to add asterisk (*) for each level of reference. For example:

```
int a=20;
int *p;
int **q; → pointer to “a pointer to an integer”
p=&a;
q=&p;
```

- To refer to variable ‘a’ using pointer ‘q’, dereference it once i.e. *p

- To refer to variable ‘a’ using pointer ‘q’, dereference it twice because there are two levels of indirection involved.

- Both *p and **q displays 20 if they are printed with a printf statement.


variable name
pr1
66X123X1
XX661111
pr2
XX771230
66X123X1
num
123
XX771230
value
variable's address
Double pointer memory representation

Double Pointer

```
int main () {
    int  var;
   int  *ptr;
   int  **pptr;
   var = 3000;
   ptr = &var;
   /* take the address of ptr using address of
operator & */
   pptr = &ptr;
   /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);
   return 0;
}
```

OUTPUT

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

Array Of Pointers

- An array of pointers can be declared as

```
data_type *pointer_name[size];
```

- For ex:

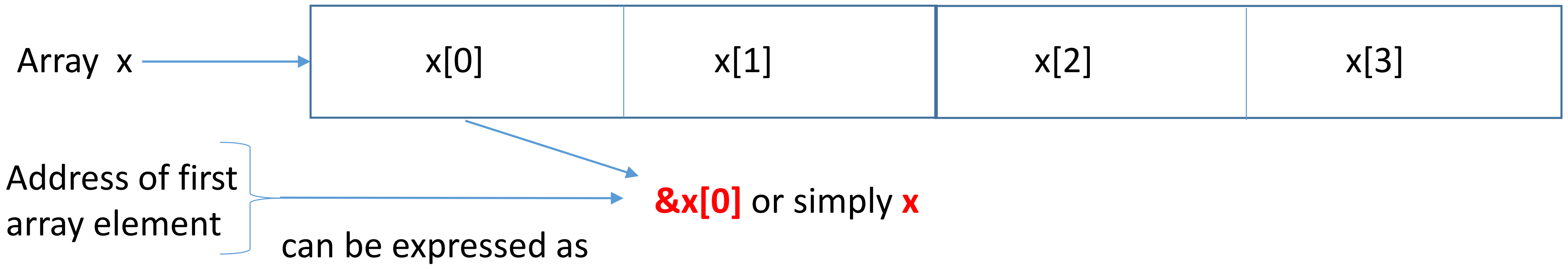
```
int *p[10];
```

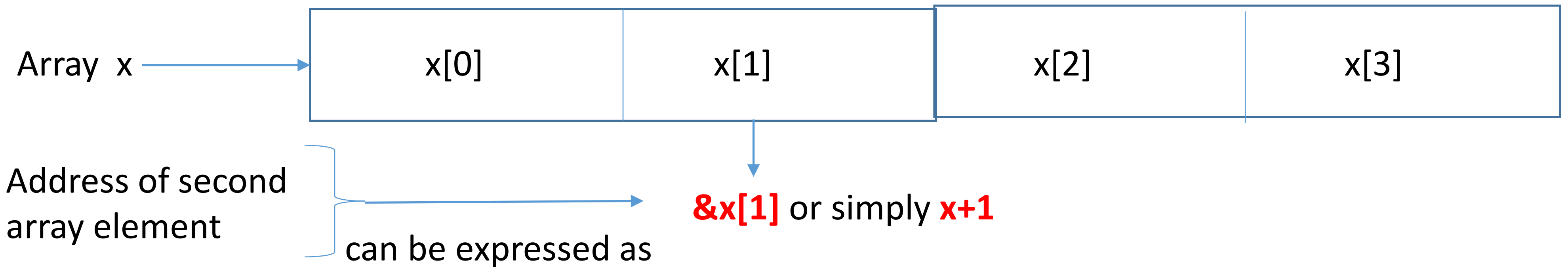
  - This declares an array of 10 pointers, each of which points to an integer. The first pointer is called p[0], the second is p[1] and so on up to p[9].
  - Initially, these pointers are uninitialized and they can be used as below.

```
int a=10, b=100, c=1000;
p[0]=&a;
p[1]=&b;
p[2]=&c; and so on.
```

Relationship between 1-D array and pointer

- Array name by itself is an address or pointer.
- It points to the address of the first element(0th element of an array)
- If x is 1D array, the address of the first element can be expressed as &x[0] or as x.
- Similarly address of second array element can be written as &x[1] or x+1.
- In general, address on an array element i can be expressed as &x[i] or x+i





- In general address of array element **i** can be expressed as &x[i] or **x+i**
- **x[i]** and ***(x+i)** both represents represents the content of the address.

OUTPUT

array element	elements value	address
x[0]	20	0x7fff5bb0bbb0
x[1]	40	0x7fff5bb0bbb4
x[2]	60	0x7fff5bb0bbb8
x[3]	80	0x7fff5bb0bbbc
x[4]	100	0x7fff5bb0bbc0

To display array element with their address using array name as a pointer

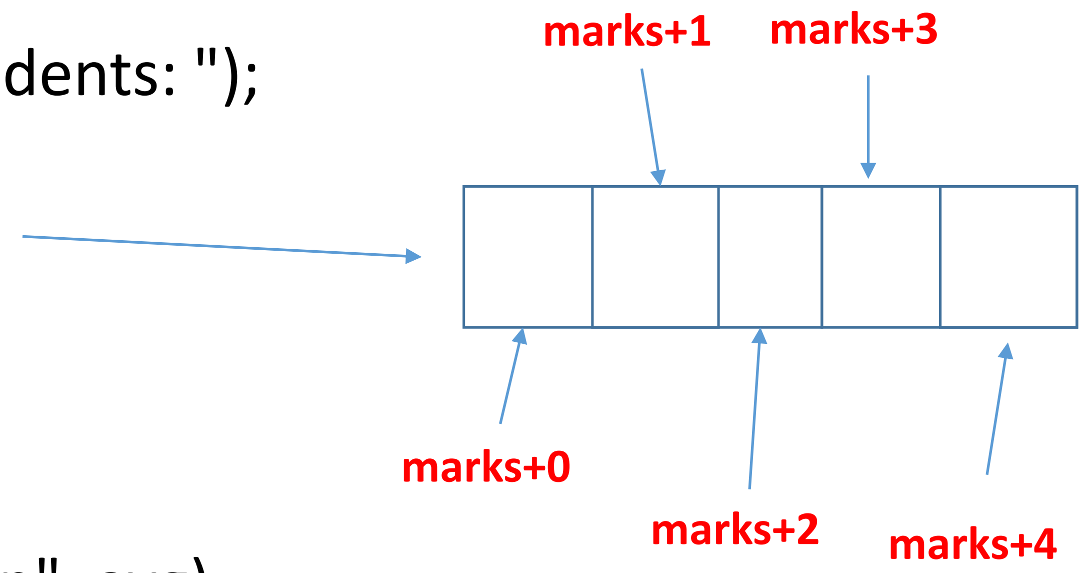
```
#include <stdio.h>

int main(){

        int x[5]={20,40,60,80,100},k;
        printf("\narray element \t\telements value \t\taddress\n");
        for(k=0;k<5;k++){
                printf("x[%d]\t\t\t%d\t\t\t%p\n",k,*(x+k),x+k );
        }
}
```

- Element k acts as the element number(0,1,2,3,4)
- x acting array is added with k i.e k is added with the address of first element, it points to the consecutive memory location.
- Thus **&x[k]** is same as ***(x+k)**

```
/* WAP to calculate average marks of 10 students in a subject using pointer*/
#include <stdio.h>
int main(void){
        float marks[10],sum=0;
        int i;
        float avg;
        printf("Enter marks of 10 students: ");
        for(i=0;i<10;i++){
                scanf("%f",marks+i);
                sum+=*(marks+i);
        }
        avg=sum/10;
        printf("\nThe average is=%f\n", avg);
}
```



Pointers and 2-D Arrays

- A two dimensional array is actually a collection of one dimensional arrays, each indicating a row (i.e. 2-D array can be thought as one dimensional array of rows).

- It is stored in memory in the row form. For ex.

$$a=\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

- Is stored in the row major order in memory as illustrated below

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	a[2][0]	a[2][1]	a[2][2]
1	2	3	4	5	6	7	8	9
65500	65502	65504	65506	65508	65510	65512	65514	65516

Syntax for declaration of 2-D array

- data_type (*ptr_var)[size2];

- Instead of data_type array[size1][size2];

- Ex: Suppose x is a two dimensional integer array having 4 rows and 5 columns. We declare x as

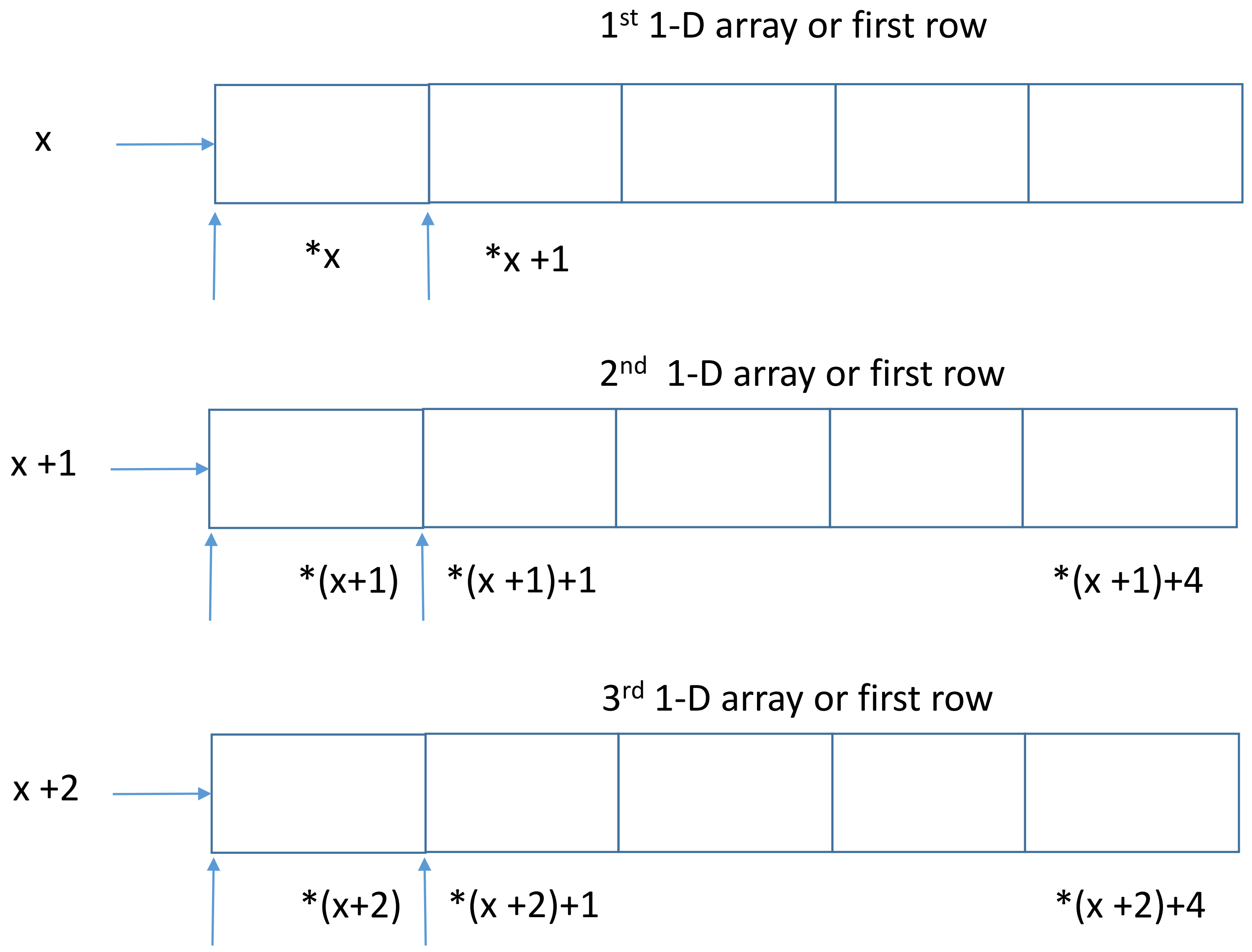
  int (*x)[5];

  rather than

  int x[4][5];

- x points to the first 5 element array, which is actually first row of the two dimensional array.

- Similarly x+1 points to the second 5 element array, which is the second row of the two dimensional array


1st 1-D array or first row
x
*x
*x +1
2nd 1-D array or first row
x +1
*(x+1)
*(x +1)+1
*(x +1)+4
3rd 1-D array or first row
x +2
*(x+2)
*(x +2)+1
*(x +2)+4

It can be concluded that	
x	pointer to 1st row
x+i	pointer to ith row
*(x+i)	Pointer to first element in the ith row
*(x+i)+j	Pointer to jth element in the ith row
((x+i)+j)	Value stored in the cell i,j

Thus in 2-D array,		
&x[0][0]	Is same as	*x or *(x+0)+0
&x[0][1]	Is same as	*x +1 or *(x+0)+1
&x[2][0]	Is same as	*x +2 or *(x+2)+0
&x[2][4]	Is same as	*(x+2)+4

and

Thus in 2-D array,		
&x[0][0]	Is same as	**x or *(*(x+0)+0)
&x[0][1]	Is same as	*(*x +1) or*(*(x+0)+1)
&x[2][0]	Is same as	*(*x +2) or *(*(x+2)+0)
&x[2][4]	Is same as	*(*(x+2)+4)

```
/* Illustration of 2D array representation in Memory */

#include <stdio.h>

int main(void){
        int p[2][3]={{1,2,3},{4,5,6}};

        printf("p=%p p+1=%p",p,p+1);

        printf("*p=%p *(p+1)=%p ",*p,*(p+1));

        printf("*(p+0)+1=%p  *(p+1)+1=%p",*(p+0)+1,*(p+1)+1);

        printf("*(*(p+0)+1)=%d  *(*(p+1)+1)=%d",*(*(p+0)+1),
            *(*(p+1)+1));

}
```

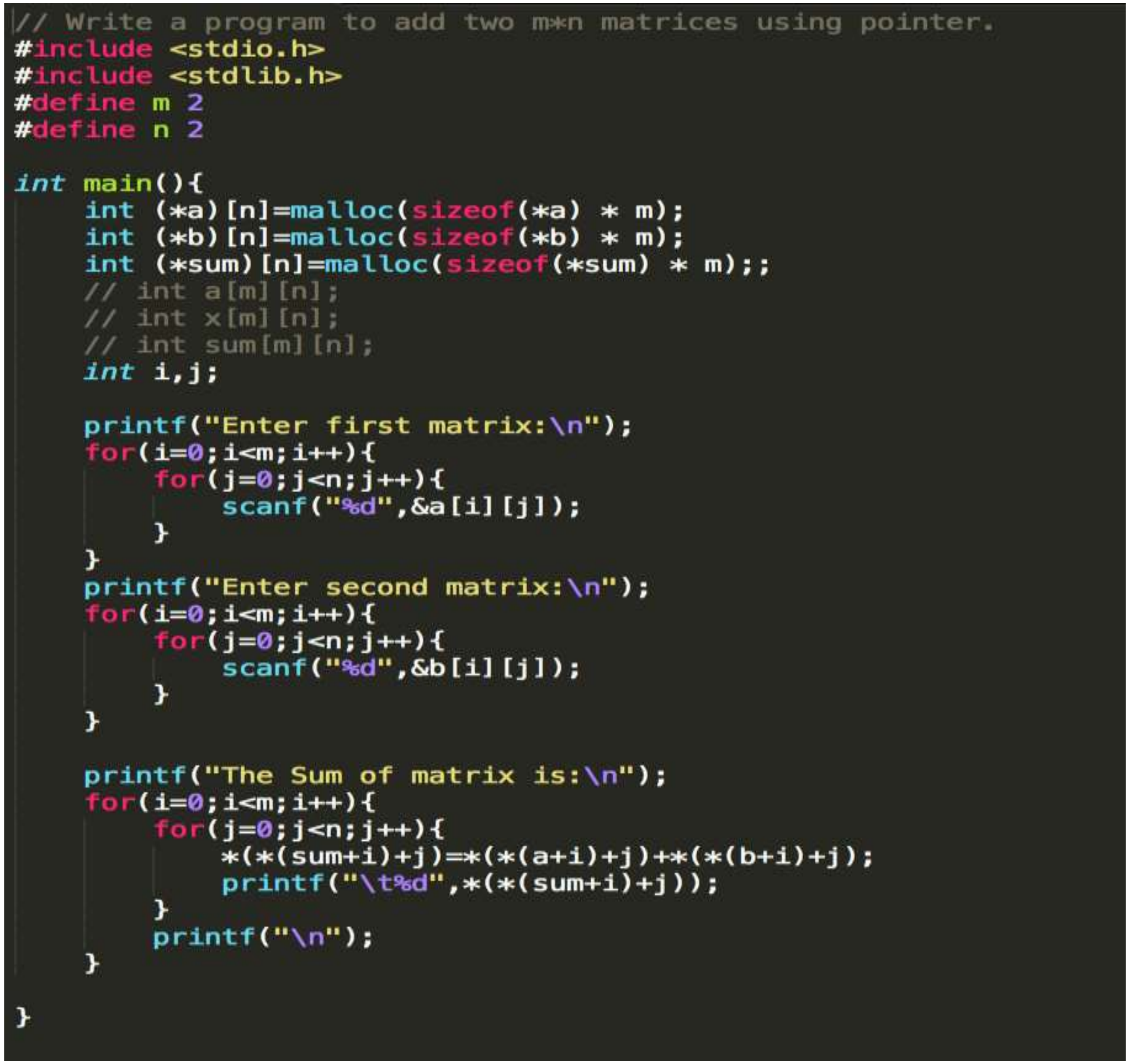
OUTPUT

```
p=0x7fff51c9dbd0            p+1=0x7fff51c9dbdc

*p=0x7fff51c9dbd0           *(p+1)=0x7fff51c9dbdc

*(p+0)+1=0x7fff51c9dbd4     *(p+1)+1=0x7fff51c9dbe0

*(*(p+0)+1)=2               *(*(p+1)+1)=5
```



```c
// Write a program to add two m*n matrices using pointer.
#include <stdio.h>
#include <stdlib.h>
#define m 2
#define n 2

int main(){
    int (*a)[n]=malloc(sizeof(*a) * m);
    int (*b)[n]=malloc(sizeof(*b) * m);
    int (*sum)[n]=malloc(sizeof(*sum) * m);;
    // int a[m][n];
    // int x[m][n];
    // int sum[m][n];
    int i,j;

    printf("Enter first matrix:\n");
    for(i=0;i<m;i++){
        for(j=0;j<n;j++){
            scanf("%d",&a[i][j]);
        }
    }
    printf("Enter second matrix:\n");
    for(i=0;i<m;i++){
        for(j=0;j<n;j++){
            scanf("%d",&b[i][j]);
        }
    }

    printf("The Sum of matrix is:\n");
    for(i=0;i<m;i++){
        for(j=0;j<n;j++){
            *(*(sum+i)+j)=*(*(a+i)+j)+*(*(b+i)+j);
            printf("\t%d",*(*(sum+i)+j));
        }
        printf("\n");
    }

}
```

Pointer Operations

- To illustrate the pointer operations, let us consider following declaration of ordinary variables and pointer variables:

```
int a,b;    float c;    int *p1,*p2,    float *f;
```

- A pointer variable can be assigned the address of an ordinary variable.

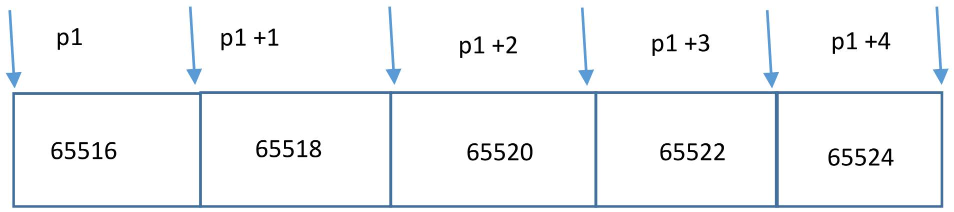
  For ex: p1=&a; p2=&b; f=&c

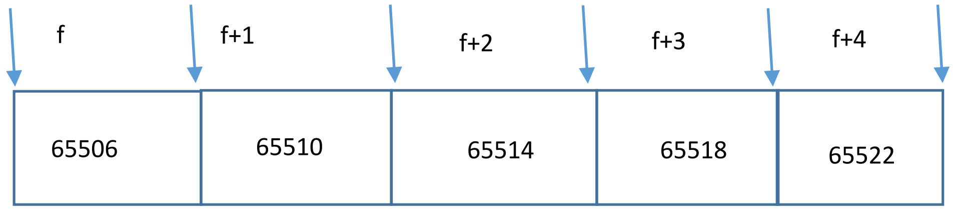
- Content of one Pointer can be assigned to other pointer provided they point to same data type. For ex.

```
p1=p2;      /* valid */
f=p1;       /* invalid */
```

- Integer data can be added to or subtracted from pointer variables. Eg.

p1+2	/* specifies an address which is two memory blocks beyond the address pointed by p1 */
f+1	/* address which is one memory block beyond address pointed by f */





If **p1** points or stores address 65516, then p1+1 means address pointed by **p1+ size in bytes of data type of pointer.**

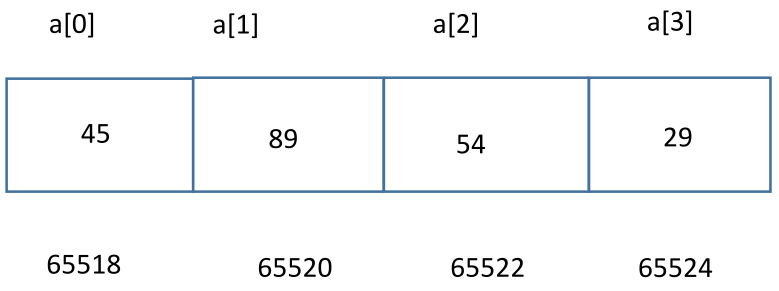
i.e 65516+2 =65518. thus p1+1 represent address 65518 instead of 65517

Here f if float type. If **f** points or stores address 65506, then **f+1** means address pointed by **f+ size in bytes of data type of pointer.**

i.e 65506+1 =65510. thus **f+1** represent address 65510 instead of 65507

- One pointer can be subtracted from other pointer provided they point to elements of same array. For ex:

```
int main(void){
        int a[]={45,89,54,29},*pf,*p1;
        p1=a;
        pf=a+2;
        printf("So pf-p1=%ld\n",pf-p1 );
}
OUTPUT: ???
```



- Here **p1**, points address of a[0] i.e. 65518 and **pf** points address to next second block i.e. 65522.

- The difference between pf and p1 means number of memory blocks for their type between addresses pointed by them i.e. **2** in this case

- Two pointer variables can be compared provided both pointers point to objects of the same data type:

```
if(p1>pf){
        /* is a valid comparison */
}
```

- There is no sense in assigning an integer to a pointer variable.

```
p1=100;       /* non sense */
p2=65516;   /* invalid because cannot be directly assigned
               like integer*/
```

- Two pointer variables cannot be multiplied and added together.

```
p1+p2;          /* invalid */
```

- A pointer variable cannot be multiplied by a constant

```
p1*2;           /* invalid */
```

- Null value can be assigned to pointer variable.

```
p1=NULL;
```

Passing Pointer to a Function

- A pointer can be passed to a function as an argument.

- Passing a pointer means passing address of a variable instead of value of the variable.

- As address is passed in this case, this mechanism is also known as **call by address** or **call by reference**.

- As address of variable is passed in this mechanism, if value in the passed address is changed within function definition, the value of actual variable is also changed.

```
#include <stdio.h>
void addGraceMarks(int *m){
                    *m=*m+10;
}
int main(void){
          int marks;
          printf("Enter Actual Marks\n");
          scanf("%d",&marks);
          addGraceMarks(&marks);
          printf("\nThe grace marks is : %d\n",marks);
}
```

```
#include <stdio.h>
void conversion(char *);
int main(){
        char input;
        printf("Enter character of your choice: ");
        scanf("%c",&input);
        conversion(&input);
        printf("The corresponding character is: %c\n",input );
}
void conversion(char *c){
        if (*c>=97 && *c<=122)
        {
                *c=*c-32;
        }
        else if(*c >=65 && *c<=90){
                *c=*c+32;
        }
}
```

Output

```
Enter Character of Your Choice: a
The corresponding Char is: A


Enter Character of Your Choice: b
The corresponding Char is: B
```

String And Pointer

- As strings are arrays and arrays are closely connected with pointers, we can say that string and pointers are closely related.

```
char name[5]="shyam";
```

- As the string variable name is an array of characters, it is a pointer to the first character of the string and can be used to access and manipulate the characters of the string.

- When a pointer to char is printed in the format of a string, it will start to print the pointer character and then successive characters until the end of string is reached.

- Thus name prints “shyam”, name+1 prints “hyam”, name+2 prints “yam” and so on.

Dynamic memory allocation (DMA)

- The process of allocating and freeing memory at run time is known as dynamic memory allocation.

- This reserves the memory required by the program and returns valuable resources to the system once the use of reserved space is utilized.

- Though arrays can be used for data storage, they are of fixed size.

```
#define m 5
int main(void){
            int x[m];
printf("%lu\n",sizeof(x));
}
```

- Consider an array size of 100 to store marks of 100 student.
- If the number of students is less than 100 say 10, only 10 memory locations will be used and rest 90 locations are reserved but will not be used i.e. wastage of memory will occur.
- In such situation DMA will be useful.

- Since an array name is actually a pointer to the first element within the array, it is possible to define the array as a pointer variable rather than as a conventional array.

- While defining conventional array, system reserves fixed block of memory at the beginning of program execution which is inefficient but this does not occur if the array is represented in terms of a pointer variable.

- The use of pointer variable to represent an array requires some type of initial memory assignment before the array elements are processed.

- This is known as Dynamic Memory Allocation (DMA)

- At execution time, a program can request more memory from a free memory pool and frees if not required using DMA.

- Thus, DMA refers allocating and freeing memory at execution time or run time.

- There are four library functions for memory management.
  - malloc()
  - calloc()
  - free()
  - realloc()

- These functions are defined within header file **stdlib.h** and **alloc.h**

1. malloc()

- It allocates requested size of bytes and returns a pointer to the first byte of the allocated space.

```
ptr=(data_type*) malloc(size_of_block);
```

- Here, ptr is a pointer of type *data_type*. The malloc() returns a pointer to an area of memory with size *size_of_block*.

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

- A memory space equivalent to 100 times the size of an integer is reserved and the address of the first byte of the memory allocated is assigned to the pointer x of type int. (i.e. x refers to the first address of allocated memory)

2. calloc()

- The function provides access to the C memory heap, which is available for dynamic allocation of variable-sized blocks of memory.

- Unlike malloc(), it accepts two arguments: *no_of_blocks* and *size_of_each_block* specifies the size of each item.

- The function calloc allocates multiple blocks of storage, each of the same size and then sets all bytes to zero.

- calloc initializes all bytes in the allocated block to zero.

ptr=(*data_type****)calloc(***no_of_blocks,size_of_each_block***);**

- For Ex:

```
x=(int*) calloc(5,10*sizeof(int));
                Or
x=(int*) calloc(5,20);
```

- The above statement allocates contiguous space for 5 blocks, each of size 20bytes. i.e. We can store 5 arrays, each of 10 elements of integer types.

3. free()

- Frees previously allocated space by calloc, malloc or realloc functions.

- The memory dynamically allocated is not returned to the system until the programmer returns the memory explicitly. This can be done using free() function.

- This function is used to release the space when not required.

- Syntax: **free(ptr);**

4. realloc()

- This function is used to modify the size of previously allocated space.
- Sometimes the previously allocated memory is not sufficient; we need additional space and sometime the allocated memory is much larger than necessary.
- We can change memory size already allocated with the help of function realloc().

- If the original allocation is done by the statement

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

- Then, reallocation of space may be done by the statement

  ptr=realloc(ptr,*newsize*);

- This function allocates a new memory space of size *newsize* to the pointer variable **ptr** and returns a pointer to the first byte of the new memory block and on failure the function return NULL

Applications of pointer

- Pointer is widely used in Dynamic Memory Allocation.
- Pointer can be used to pass information back and forth between a function and its reference point.
- Pointers provide an alternative way to access individual array elements.
- Pointers increase the execution speed as they refer address.
- Pointers are used to create complex data structures such as linked list, trees, graphs and so on.

Reference

- https://en.wikipedia.org/wiki/C_dynamic_memory_allocation
- http://stackoverflow.com/questions/21376645/store-string-into-array-in-c
- http://stackoverflow.com/questions/26431147/abort-trap-6-error-in-c
- http://www.gnu.org/software/libc/manual/html_node/String-Length.html
- http://www.tutorialspoint.com/cprogramming/c_strings.htm
- https://www.quora.com/What-is-the-difference-between-runtime-and-compile-time