CP MODULE 3

Pointers: Array of pointers, structures and pointers. Example programs using pointers and structures.

POINTERS

ABOUT MEMORY LOCATION AND STORAGE SPACE

Within the computer's memory, every stored data item occupies one or more contiguous memory cells (i.e., adjacent words or bytes).

Computer's memory is a sequential collection of locations where each location can store 1 byte of data.

The number of memory cells required to store a data item depends on the type of data item.

For example,

- an **integer** usually requires***two contiguous bytes***;
- a single ***character*** will typically be stored in ***one byte*** (8 bits) of memory;
- a **floating-point** number may require ***four contiguous bytes***; and
- a **double-precision** quantity may require ***eight contiguous bytes***.

E.g.

int a;

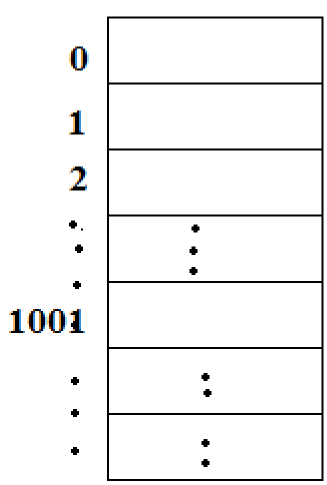
float b;

char c;

Suppose integer variable **a** is stored in memory location 2000 .(It will occupy(2 bytes) locations 2000 and 2001. Address of a is represented by &a

Suppose floating point variable **b** is stored in memory location 4100.(It will occupy(4 bytes) locations 4100 to 4103. Address of b is represented by &b

Suppose character variable **c** is stored in memory location 3105.(It will occupy(1byte) locations 3105. Address of c is represented by &c



Computer memory showing location 0,1……...1001……

Pointers

A pointer is a variable that ***represents the location*** (rather than the value) ***of a data item***s such as a variable or an array element.

- The data item can then be accessed if we know the location (i.e., the address) of the first memory cell.
- The address of a variable v 's memory location is &v, where **&** is a unary operator, called the **address operator**, that evaluates the address of its operand.
- let us assign the address of v to another variable, pv. Thus,
  int v=10;
  int *pv;
  pv = &v;
- This new variable pv is called a <u>pointer to v, since it "points" to the location where v is stored in memory.</u>
- pv represents v's address, not its value.
- Thus, pv is referred to as a **pointer variable.**

Relationship between pv and v (where pv = &v and v = *pv)

The data item represented by v (i.e., the data item stored in v's memory cells) can be accessed using pointer variable pv using expression *pv, where * is a unary operator, called the **indirection operator,** that operates only on a pointer.

Therefore, *pv and v both represent the same data item (i.e., the contents of the same memory cells).

POINTER DECLARATIONS

- Pointer variables, like all other variables, must be declared before they may be used in a C program.
- When a pointer variable **variable name must be preceded by an asterisk** (*)is declared, the .
  - This identifies the fact that the variable is a pointer.
- <u>**A pointer declaration**</u> may be written in general terms as
  data- type *ptvar;
  - where ptvar is the name of the pointer variable, and data-type refers to the data type of the pointer's object

float *p;

This declares pv to be a pointer variable whose object is a floating-point quantity; i.e., pv points to a floating-point quantity. pv represents an address, not a floating-point quantity.

POINTER INITIALIZATION

Within a variable declaration, a pointer variable can be initialized by assigning it the address of another variable.

E.g;

int a=10;

int *p=&a;

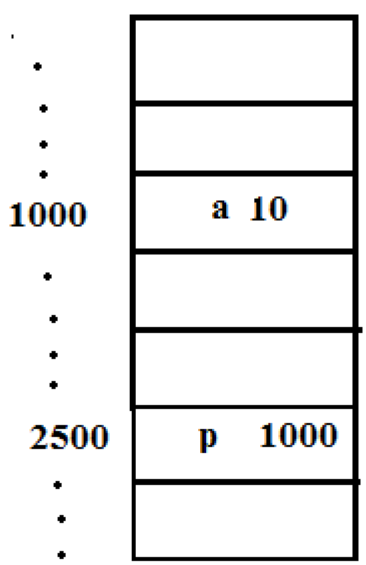
This can also be written as

int a=10;

int *p;

p=&a;

Here p is an integer pointer variable in location 2500 that stores the address of integer variable a. So integer variable a should be declared before declaring the pointer variable p.



Suppose a is stored in memory location 1000. Pointer variable p is in memory location 2500.

int a=10;

int * p=&a;

or

int *p;

p=&a;

This means that the pointer variable p stores the address of variable a.

Here the value of :

address of a **&a is 1000**

Since p=&a **p is also 1000**

&p is the address of p . Here **&p is 2500.**

a is 10. So the value of ***p is also 10**

E.g.

```
#include<stdio.h>
main()
{
	inta,*p;
	a=45;
	p=&a;

	printf("The value of a is %d *p is %d\n",a,*p);
	*p=25;
	printf("The value of *p now is %d\n",*p)
	printf("The value of a now is %d\n",a);
}
```

Output is:
The value of a is 45 *p is 45
The value of *p now is 25
The value of a now is 25

E.g.

Pointer Examples

```
#include<stdio.h>
main()
{
int a=10, b=20, *p, *j;
p=&a;
j=&b;
printf("\nAddition a + b = %d", a + b);

printf("\nAddition *p + b = %d", *p + b);
printf("\nAddition *p + *j = %d", *p + *j);
printf("\nAddition *(p) + *(j) = %d", *(p) + *(j));
printf("\nAddition *(&a) + *(&b) = %d", *(&a) + *(&b));
}
```

OUTPUT
Addition a + b=30

Addition *p + b=30
Addition *p + *j =30
Addition *(p) + *(j) =30
Addition *(&a) + *(&b) = 30

NULL POINTER

- Null pointer does not currently point to a valid memory location. It is given the value null (which is **zero**).
- **Null pointer** does not hold the address of any element.
- A null pointer does not point to anywhere.
- One way to give a pointer a null value is to assign zero to it.

E.g.

```
#define NULL 0
main()
{
   float *pv = NULL;
}
```

Here pv is declared as a pointer variable that points to a floating-point quantity. In addition, pv is initially assigned a value of 0 to indicate some special condition. pv is null pointer.
null pointer is a value whereas void pointer is a type.

VOID POINTER

- A void pointer is a pointer that *has no associated data type* with it.
- Void pointer is a generic pointer
  - It can hold address of any type of variable.
- A void pointer is declared like a normal pointer, using the **void** keyword as the pointer's type.

  void *vptr;
- pointers of one data type cannot hold the address of a variable of some other type.

- void pointer **cannot be dereferenced** *because the void pointer does not know what type of object it is pointing to*
  - To solve this type casting can be done.
  - When we assign *address of integer to the void pointer*, pointer will become ***integer pointer***. To print the value using that void pointer it has to be ***typecasted***to **(int *)**
  - When we assign *address of character to the void pointer*, pointer will become ***character* pointer.** To print the value using that void pointer it has to be ***typecasted***to **(char *)**

- If the void pointer contains the address of a float variable, then we need to **typecast** the pointer to (float*) before printing the variable's value.

E.g.
main()
{
inti=2;
void * ptr;
ptr=&i;
printf("%d ",*(**(int*)**ptr));
}
ptr is a void pointer. It is type casted into integer

- **no arithmetic operations** can be performed on void pointer.

void *ptr = 0; // Here ptr is a void pointer that is currently a null pointer

CONSTANT POINTER

A **constant pointer** is a pointer that cannot change the address it is holding. In other words, once a constant pointer points to a variable, then it cannot point to any other variable. Trying to do so will result in error.

A **constant pointer** is declared as follows :<type of **pointer**> * **const**<name of **pointer**>
E.g. int* constptr;

#include<stdio.h>
main()
{
int var1 = 0, var2 = 0;
int *constptr = &var1;***//ptr is const pointer. It contains the address of var1.***
ptr = &var2; ***//ERROR, ptr is a const pointer. The address it holds cannot be changed.***
printf("%d\n", *ptr);
}

OPERATIONS ON POINTERS / POINTER ARITHMETIC

int v=2,u=5;
int *pv;
int *pu

1. A pointer variable can be assigned the address of an ordinary variable (e.g., pv = &v).
2. A pointer variable can be assigned the value of another pointer variable (e.g., pv = px) provided both pointers point to objects of the same data type .
3. A pointer variable can be **assigned a null (zero) value** (e.g., pv = NULL, where NULL is a symbolic constant that represents the value 0).
4. An*integer* quantity can be **added to or subtracted** from a pointer variable (e.g., pv + 3, ++pv, etc.)

5. One pointer variable can be **subtracted** from another pointer *so that both pointers point to elements of the same array.*
6. Two pointer variables can be **compared** provided *both pointers point to objects of the same data type.*

- ***Other arithmetic operations on pointers are not allowed.***
  E.g.
  - a pointer variable cannot be multiplied by a constant;
  - two pointer variables cannot be added;
- Ordinary variable cannot be assigned an arbitrary address (i.e., an expression such as &x cannot appear on the left side of an assignment statement).
  E.g

```
int a=2;
int b=3;
&a=b;  // ERROR
```

POINTERS AND ONE-DIMENSIONAL ARRAYS

If x is a one dimensional array, then

- the *address of the first array element* can be expressed as either &x [0] or simply as x
- The address of the second array element can be written as either &x [1] or as (x + 1), and so on.
- In general, the address of array element (i + 1) can be expressed as either &x [i] or as (x+i).
- Since &x[i] and (x + i)both represent the address of the ith element of x,
  - x[i]and * (x + i)both represent the contents of that address, i.e., the value of the ith element of x.

static intx[10]={10,20,25}; // x is integer array of size 10 with initial value 10,11,12
//x[0] is 10 x[1] is 20 x[2] is 25
//remaining spaces filled with 0	

x x+0	Address of 0th element(starting element) in array &x[0]
*x *(x+0)	Value Of 0th element(starting element) in array x[0]
x+i	Address of ith element in array &x[i]

*(x+i)	Value Ofith element in array x[i]

Example Display contents in array using pointer

```
# include <stdio.h>
main( )
{
static  int  x[10] = {10, 11 , 12, 13, 14, 15, 16, 17, 18, 19};
inti ;
        for ( i = 0; i<= 9; ++i)
        {
        printf( " %d\t”, * ( x+i ) ) ;
        }
}
```

Or

```
#include<stdio.h>
main()
{
static  int  x[10] = {10, 11 , 12, 13, 14, 15, 16, 17, 18, 19};
inti ;
int *p;
p=x;
for( i = 0; i<= 9; ++i)
{
printf( "%d\t", *p ) ;
p++;
}
}
```

OUTPUT

10 11 12 13 14 15 16 17 18 19

- x is an integer array of size 10
  intx[10];
- x, (x + i) and &x [i] cannot appear on the left side of an assignment statement.
  - Because the address of an array cannot arbitrarily be altered
  - So expressions such as ++x are not permitted.
  - Note that the address of one array element cannot be assigned to some other array element. Thus we cannot write a statement such as
    &line[2] = &line[l];

- we can assign the value of one array element to another through a pointer

```
intline[80];
int *pl;
/*To  assign(store)  value of line[1] in line[2]*/
line[2] = line[l];
or
line[2] = *(line + 1);              // line+1 is  &line[1]
                                    //*(line+1) is line[1]
or
pl = &line[l];
line[2] = *pl;
Or
pl = line + 1;
*(line + 2) = *pl;
```

- If a numerical array is defined as a pointer variable, the array elements cannot be assigned initial values.

STRINGS and POINTERS

- A character-type pointer variable can be assigned an entire string when it is declared.
    - So string can conveniently be represented by either a one-dimensional character array or a character pointer.

```
#include <stdio.h>
char x[] = "This string is declared externally\n";
main( )
{
static char y[] = "This string is declared within main";
printf( " % s " , x);
printf( "%s", y) ;
}
```

Here definition of y occurs within a function; therefore y[ ] must be defined as static array so that it can be initialized.

```
#include <stdio.h>
char *x = "This string is declared externally\n";
main( )
{
char *y = "This string is declared within main";
printf( " %s" , x);
```

```
printf( "%s", y) ;
printf( " *x= %c " , *x);
printf( " *(x+1)=%c", *(x+1)) ;

}
```

OUTPUT

This string is declared externally

This string is declared within main

*x=T

*(x+1)=h

- Although x is a pointer here, no need to write *x to print the string because x is also the name of the string.
- *x will actually print the zeroth character of the string.]

The external pointer variable x points to the beginning of the first string, whereas the pointer variable y, declared within main, points to the beginning of the second string. Since y is a pointer, y can be initialized without being declared static.

e.G.input string as pointer

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

intmain(){
    char *str;
str=(char *) malloc(sizeof(char)*12);

printf("enter the string : ");
scanf("%s", str);
printf("you entered %s\n", s);
}
```

E.g. **Input 3 strings using pointers**

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

intmain(){
        char *s[3];
        intn,len,i;

        for(i=0;i<3;i++)
        {
        s[i]=(char *) malloc(sizeof(char)*12);
        }
        printf("Enter  strings");
```

```
        for(i=0;i<3;i++)
        {
        scanf(" %s",s[i]);
        }
for(i=0;i<3;i++)
        {
        printf("\n %s",s[i]);
        }

}
```

E.g. **The program to finds the length of a string using pointers.**

```
#include<stdio.h>
main()
{
char *cptr,str[10];
int length=0;
printf("Enter the string\n");
scanf("%s",str);
cptr=str;                       //pointer to first(0th) character in the string
while(*cptr!='\0')
{
length++;
cptr++;         /*incrementing the pointer so that it points to next character in the string*/
}
printf("The length of the string %s is %d\n",str,length);
}
```

OUTPUT

Enter the string
Hello
The length of the string Hello is 5

***p++ is same as *(p++)**

It will first compute *p (value stored in content of p(address of an item)). Then p will be incremented to point to next location.

Eg.

x=*p++;

Or

x=*(p++);

is equivalent to writing

x=*p;

p=p+1;

But (*p)++ will first compute *p. Then the value of *p is incremented by 1.

x=(*p)++;
is equivalent to
x=*p;
***p=*p+1;**

```
#include <stdio.h>
main()
{
intarr[] = {10, 20};
int *p = arr;
printf("arr[0] = %d, arr[1] = %d, *p = %d",  arr[0], arr[1], *p);
printf("\narr[0] = %d, arr[1] = %d, (*p)++ = %d",  arr[0], arr[1], (*p)++);
printf("\narr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
  }
```

Output

```
arr[0] = 10, arr[1] = 20, *p = 10
arr[0] = 11, arr[1] = 20, (*p)++ = 10
arr[0] = 11, arr[1] = 20, *p = 11
```

```
#include <stdio.h>
main(void)
{
intarr[] = {10, 20};
int *p = arr;
printf("arr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
printf("\narr[0] = %d, arr[1] = %d, *p++ = %d",arr[0], arr[1], *p++);

printf("\narr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
}
```

Output

```
arr[0] = 10, arr[1] = 20, *p = 10
arr[0] = 10, arr[1] = 20, *p++ = 10
arr[0] = 10, arr[1] = 20, *p = 20
```

```
#include <stdio.h>
main(void)
{
intarr[] = {10, 20};
int *p = arr;
```

```
printf("arr[0] = %d, arr[1] = %d, *p = %d",arr[0], arr[1], *p);
printf("\narr[0] = %d, arr[1] = %d, *(p++) = %d", arr[0], arr[1], *(p++));
printf("\narr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);    }
```

Output

arr[0] = 10, arr[1] = 20, *p = 10
arr[0] = 10, arr[1] = 20, *(p++) = 10
arr[0] = 10, arr[1] = 20, *p = 20

```
#include <stdio.h>
int main(void)
{
intarr[] = {10, 20,30};
int *p = arr;
   *++p;
printf("arr[0] = %d, arr[1] = %d, arr[2]=%d, *p = %d", arr[0], arr[1],arr[2],  *p);

printf("\narr[0] = %d, arr[1] = %d,arr[2] = %d, *(p++) = %d", arr[0],arr[1], arr[2], *(p++));

printf("\narr[0] = %d, arr[1] = %d,arr[2] = %d, (*p) = %d", arr[0], arr[1], arr[2], *p);
printf("\narr[0] = %d, arr[1] = %d,arr[2] = %d, (*p)++ = %d", arr[0], arr[1],arr[2],  (*p)++);
printf("\narr[0] = %d, arr[1] = %d,arr[2] = %d, (*p) = %d", arr[0], arr[1], arr[2], *p);
  return 0;
}
```

arr[0] = 10, arr[1] = 20,arr[2] = 30, *p = 20
arr[0] = 10, arr[1] = 20,arr[2] = 30, *(p++) = 20
arr[0] = 10, arr[1] = 20,arr[2] = 30, (*p) = 30
arr[0] = 10, arr[1] = 20,arr[2] = 31, (*p)++ = 30
arr[0] = 10, arr[1] = 20,arr[2] = 31, (*p) = 31

DYNAMIC MEMORY ALLOCATION

A conventional array definition results in a fixed block of memory being reserved at the beginning of program execution, whereas this does not occur if the array is represented in terms of a pointer variable.

Example:-

inta[10];

This will allot 10 spaces for storing 10 numbers in array a.

The mechanism by which ***storage***/memory/cells *can be allocated to variables during the* ***run time***is called **dynamic memory allocation.**

Dynamic memory allocation methods are

- malloc()

- free()
- calloc()

If we use pointer variable to represent an array, we have to assign required memory(storage) before processing the array elements. This is known as dynamic memory allocation.
Generally, the **malloc**library function is used for allocation required space

Suppose x is a one-dimensional, 10-element array of integers. It is possible to define x as a pointer variable rather than an array. Thus, we can write
int *x;
instead of writing the following:-
intx[10];
Or
#define SIZE 10
int x[SIZE];
When x is defined as a pointer variable, x is not automatically assigned a memory block, but a block of memory large enough to store 10 integer quantities will be reserved in advance when x is defined as an array.
<u>To assign sufficient memory for pointer variable x</u>, we can make use of the library function **malloc**, as follows.
x = (int *) malloc(l0 * sizeof(int));
This function reserves a block of memory whose size (in bytes) is equivalent to 10 integer quantities.
Consider double pointer y to store 10 doublenumbers:-
double *y;
y = (double *) malloc(l0 * sizeof(doub1e));
Here y is a pointer to double-precision quantity and we have can enough memory to store 10 double-precision quantities.

- If the declaration is to <u>***include the assignment of initial values***</u>, then <u>x must be defined as an array</u> rather than a pointer variable. For example,

int x[l0] = {1, 2, 3, 4, 5 , 6, 7, 8, 9, 10};
or
intx[] = {1, 2, 3, 4, 5 , 6, 7, 8, 9, 10};

- **malloc** and calloc() are library functions that allocate memory dynamically. It means that memory is allocated during runtime(execution of the program) from heap segment.

• **malloc()** allocates memory block of given size (in bytes) and returns a	• **calloc() allocates the memory and also initializes the allocates memory block to zero.**

pointer to the beginning of the block. malloc() doesn’t initialize the allocated memory. ○ If we try to access the content of memory block then we’ll get garbage values. void * malloc(size_t size); ptr = (cast-type*) malloc(byte-size)	○ If we try to access the content of these blocks then we’ll get 0. void * calloc(size_tnum, size_t size); ● calloc() takes two arguments: 1) Number of blocks to be allocated. 2) Size of each block.

free(pointervariable)

free(pointer variable) is a dynamic memory management function that help to release the memory allocated using malloc() function.

free method is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() are not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

```
#include <stdio.h>
main()
{
    printf("Enter elements\n");
    int *x;
    x=(int*)malloc(10*sizeof(int));
    inti;
    for(i=0;i<10;i++)
    {
    scanf("%d",x+i);
    }
    printf("Elements are \n");
```

```
for(i=0;i<10;i++)

{

printf("%d\t",*(x+i));

}

free(p);  // This statement frees the space allocated in the memory pointed by p.

}
```

POINTERS AND MULTIDIMENSIONAL ARRAYS

- A two-dimensional array, for example, is actually a ***collection of one-dimensional arrays***.
- Therefore, we can define a two-dimensional array as a **pointer to a group of contiguous one-dimensional arrays.**
- A two-dimensional array declaration can be written as

  data- type (*ptvar) [expression2] ;

instead of writing

data- type array[expression I] [expression 21;

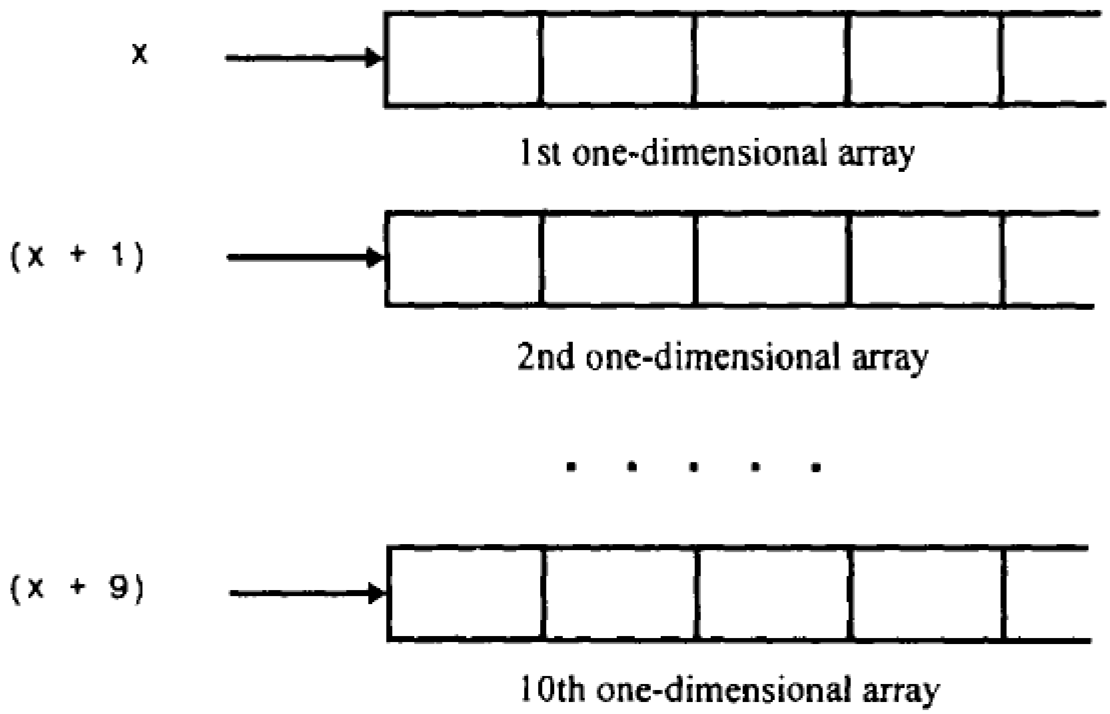
E.gSuppose x is a two-dimensional integer array having 10 rows and 20 columns. Instead of writing as

intx[10][20];

We can declare x as

int (*x)[20];

- Herex is defined to be a pointer to a group of contiguous, one-dimensional, 20-element integer arrays.
- Thus, x points to the first 20-element array, which is actually the first row (i.e., row 0) of the original two-dimensional array.
- Similarly, (x + 1) points to the second 20-element array, which is the second row (row 1) of the original two dimensional array,



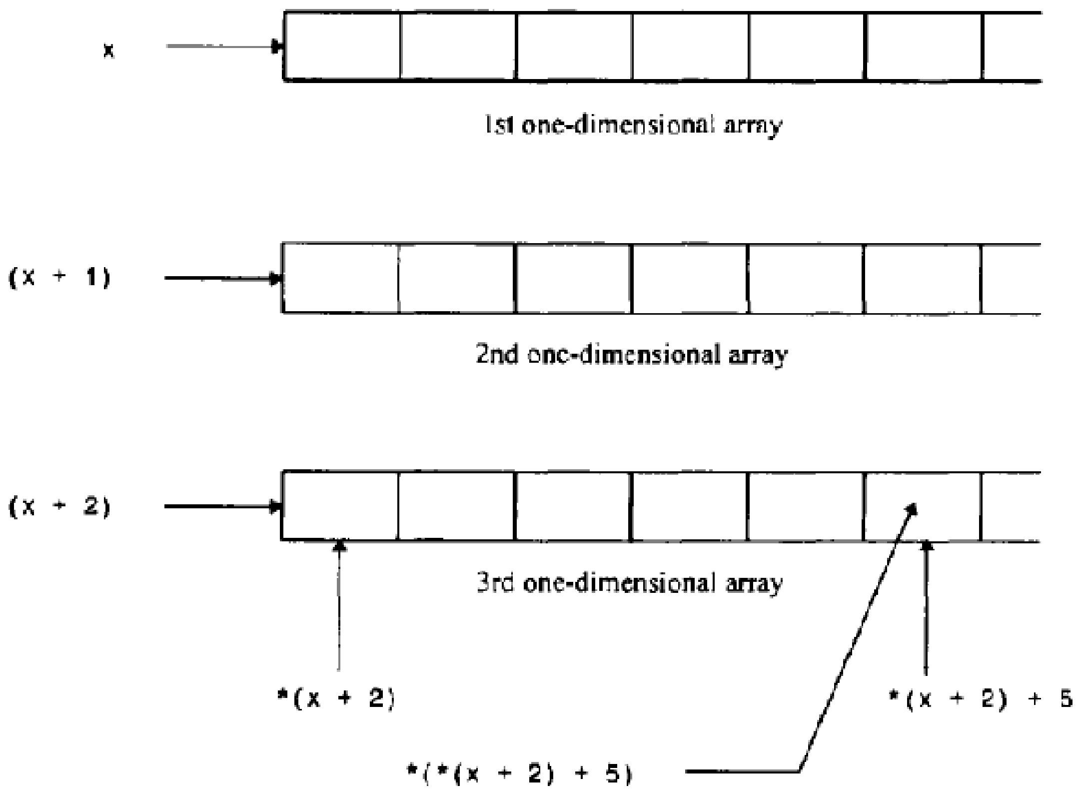
int (*x)[20];
x is the pointerto 0th array
(x+i) is the address of x[i] .
(x+1) is &x[i]
***(x+i) is same as x[i]**
(*(x+i)+j) is &x[i][j]
So *(*(x+i)+j) is same as x[i][j]

.

Row and column number starts from 0.
Suppose x is a two-dimensional integer array having 10 rows and 20 columns, as declared in the previous example. The item in row 2, column 5 can be accessed by writing either
x[2][5]
or
* (* (x + 2) + 5)

- (x + 2) is a pointer to row 2.
- * (x + 2),refers to the entire row.
- Since row 2 is a one-dimensional array, * (x + 2) is actually a pointer to the first element in row 2.
- We now add 5 to this pointer.

- Hence, (* (x + 2) + 5) is a pointer to element 5 (i.e., the sixth element) in row 2.
- * (* (x + 2) + 5),therefore refers to the item in column 5 of row 2, which is x[2][5]



Let row be the row number and ncols is the number of columns, memory can be allocated using:-

x[row] = (int *) malloc(nco1s * sizeof(int)) ;

E.g. allocate space for storing matrix with 10 rows and 20 columns:-

```
int (*x)[20];
Memory can be allocated using
row=10;
col=20
for(i=0;i<row;i++)
{
a[i]= ( int *) malloc(nco1s * sizeof(int ) ) ;
}
```

Each array element can be inputted as

```
for(i = 0;i <row; ++i)
 {
```

```
            for( j= 0; j <col; ++j)
            {
            scanf("%d", (*(a + i) + j ) ) ;
            }
    }
```

Here (*(a + i) + j) is the address of jth column element in ith row

Each array element can be printed as

```
for(i = 0;i <row; ++i)
 {
        for( j= 0; j <col; ++j)
        {
        printf("%d", *(*(a + i) + j ) ) ;
        }
}
```

ARRAYS OF POINTERS

It is better to express a multidimensional array in terms of ***an array of pointers*** rather than a pointer to a group of contiguous arrays.

- This array will have one less dimension than the original multidimensional array.
    - If we want to represent a two dimensional array we can use one dimensional array of pointers.

A two-dimensional array can be defined as a *one-dimensional array of pointers*

data-type *array[expression 1) ;

than using the following conventional array definition,

data- type array[expression 1] [expression 2] ;

E.g.

int x[10][2];

can be written as

int *x[10];

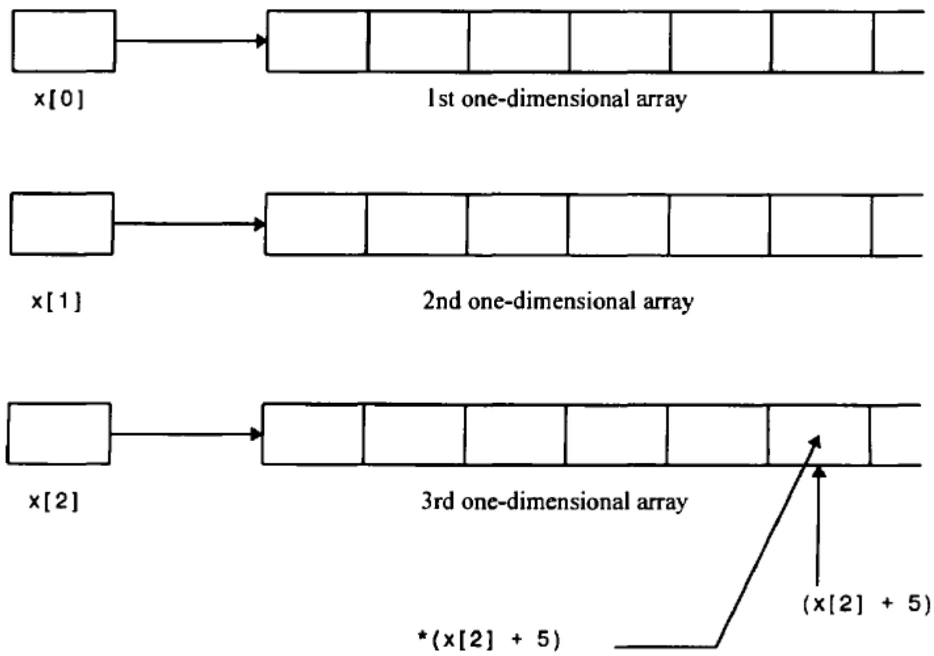
Here there are 10 pointers x[0] to x[9]

x[0] is a pointer that points to the beginning of 0th array

x[1] is a pointer that points to the beginning of 1st array

…

x[9] is a pointer that points to the beginning of 9th array



x[2][5] can be accessed using *(x[2]+5)
x[2]is the address of 0th column in 2nd row
x[2]+5 is the address of 5th column in 2nd row
*(x[2]+5) is the element in 5th column of 2nd row

Conventional array notation to store a matrix with 10 rows and 20 columns int x[10][20];	
Pointer to contiguous array of size 20	**Array of 10 pointers**
int (*x)[20]; **x is the pointerto 0th array** **(x+i) is the address of x[i] .** **(x+1) is &x[i]** ***(x+i) is same as x[i]** **(*(x+i)+j) is &x[i][j]** **So *(*(x+i)+j) is same as x[i][j]**	**int *x[10];** **x[0] is the pointer to 0th array** **x[i] is the address of ithrow .** **(x[i]+j) is &x[i][j]** **So *(x[i]+j) is same as x[i][j]**

int *x[20];
Memory can be allocated using
row=10;
col=20;
for(i=0;i<row;i++)

```
		{
		a[i]= ( int *) malloc(col * sizeof(int ) ) ;
		}
```

Each array element can be inputted as

```
	for(i = 0;i <row; ++i)
	 {
		for( j= 0; j <col; ++j)
		{
		scanf("%d", (a[i]+ j ) ) ;
		}
	}
```

Each array element can be displayed as

```
	for(i = 0;i <row; ++i)
	 {
		for( j= 0; j <col; ++j)
		{
		printf("%d", *(a[i]+ j ) ) ;
		}
}.
```

PROGRAM TO READ AND DISPLAY MATRIX USING ARRAY OF POINTERS

```
#include<stdio.h>

void disp(int *x[10],intr,int c)
{
inti,j;
printf("\n Matrix \n");
	for(i=0;i<r;i++)
	{
		printf("\n");
		for(j=0;j<c;j++)
		{
		printf("%d\t",*(*(x+i) +j));
		}
	}
}
void read(int *x[10],intr,int c)
{
inti,j;
printf("\nEnter Matrix\n");
```

```
	for(i=0;i<r;i++)
	{
		for(j=0;j<c;j++)
		{
		scanf("%d",(*(x+i) +j));
		}
	}
}

main()
{
int *a[10];
inti,j,r1,c1;

printf("\nEnter number of rows and columns in matrix: \n");
scanf("%d%d",&r1,&c1);

for(i=0;i<r1;i++)
{
	a[i]=(int *) malloc(c1*sizeof(int));
}
read(a,r1,c1);
disp(a,r1,c1);
}
```

MATRIX ADDITION USING ARRAY OF POINTERS

```
#include<stdio.h>

void readmat(int *a[10],int r1,int c1)
{
inti,j;
printf("\nEnter Matrix\n");
for(i=0;i<r1;i++)
{
	for(j=0;j<c1;j++)
	{
	scanf("%d",(a[i]+j));
	}
}
```

```
}
void printmat(int *a[10],int r1,int c1)
{
inti,j;

printf("\nMatrix\n");
	for(i=0;i<r1;i++)
	{	printf("\n");
		for(j=0;j<c1;j++)
		{
		printf("%d\t",*(a[i]+j));
		}
	}

}
void addmat(int *a[10],int *b[10],int *c[10],int r1,int c1)
{
inti,j;
	for(i=0;i<r1;i++)
	{
	 c[i]=(int *)malloc(c1*sizeof(int));
	}
printf("\nMatrix\n");
	for(i=0;i<r1;i++)
	{	printf("\n");
		for(j=0;j<c1;j++)
		{
		*(c[i]+j)=*(a[i]+j)+*(b[i]+j);

		}
	}

}

main()
{
int *a[10],*b[10],*c[10];
inti,j,k,r1,c1,r2,c2;
```

```
printf("\nEnter rows and columns of first matrix: \n");
scanf("%d%d",&r1,&c1);
printf("\nEnter rows and columns of second matrix: \n");
scanf("%d%d",&r2,&c2);
if(r1!=r2||c1!=c2)
{
        printf("\nMatrix addition not possible\n");
}
else
{
for(i=0;i<r1;i++)
{
        a[i]=(int *)malloc(c1*sizeof(int));
         b[i]=(int *)malloc(c1*sizeof(int));

}

readmat(a,r1,c1);
printmat(a,r1,c1);
readmat(b,r1,c1);
printmat(b,r1,c1);
printf("\n Result matrix after addition\n");
addmat(a,b,c,r1,c1);
printmat(c,r1,c1);

}

}
```

PASSING POINTER TO FUNCTION

Pointer can be passed as function arguments. This is also called call by reference
Eg

```
void swap(int *a, int *b)
{
…...
}
```

This function is called as
swap(&a,&b);
STRUCTURE AND POINTER

NOTE *structpointervariable.member is wrong
variable to print rolno, use pointer variable and arrow operator to print name using pointer variable and indirection operator * to print mark

```
#include<stdio.h>

struct student
{
introllno;
char name[20];
int mark;
}stud,*s=&stud;
main()
{
printf("Enter roll number: ");
scanf("%d",&stud.rollno);
printf("Enter name: ");
scanf(" %s",&stud.name)   ;
printf("Enter mark: ");
scanf("%d",&stud.mark);
printf("\nrollno=%d  name=%s mark=%d ",stud.rollno,s->name,(*s).mark);
}
```

MORE ABOUT POINTER DECLARATIONS

```
int  *p; /* p i s a pointer t o an integer quantity */
int  *p[l0]; /* p i s a 10-element array of pointers t o integer q u a n t i t i e s */
int  (*p) [ 10]; /* p i s a pointer t o a 10-element int  e g e r a r r a y */
int  *p (void) ; / * p i s a function that
returns a pointer t o an integer quantity */
int  p(char *a); /* p i s a function t h a t
accepts an argument which i s a pointer t o a character and
returns an integer quantity */
int  *p(char a*); /* p i s a function t h a t
accepts an argument which i s a pointer t o a character
returns a pointer t o an integer quantity */
```

int (*p)(char *a); /* p i s a pointer t o a function t h a t
accepts an argument which is a pointer t o a character
returns an integer quantity */
int (*p(char * a)) [l O] ; /* p i s a function t h a t
accepts an argument which i s a pointer t o a character
returns a pointer t o a 10-element int e g e r a r r a y */
int p(char (* a) [ ]) ; /* p i s a function t h a t
accepts an argument which i s a pointer t o a character array
returns an integer quantity */
int p(char * a [ ]) ; /* p i s a function t h a t
accepts an argument which i s an array of pointers t o
characters
returns an integer quantity */
int *p(char a [ ]) ; /* p i s a function t h a t
accepts an argument which i s a character array
returns a pointer t o an integer quantity */
int *p(char (* a) [ ]) ; /* p i s a function t h a t
accepts an argument which i s a pointer t o a character array
returns a pointer t o an integer quantity */
int *p(char * a [ ]) ; /* p i s a function t h a t

accepts an argument which i s an array of pointers t o
characters
returns a pointer t o an integer quantity */
int (*p)(char (* a) [ ]) ; /* p i s a pointer t o a function t h a t
accepts an argument which i s a pointer t o a character array
returns an integer quantity */
int *(*p)(char (* a) [ ]) ; / * p i s pointer t o a function t h a t
accepts an argument which i s a pointer t o a character array
returns a pointer t o an integer q u a n t i t y */
int *(*p)(char * a [ ]) ; / * p i s a pointer t o a function t h a t
accepts an argument which i s an array of pointers t o
characters
returns a pointer t o an integer quantity */
int (* p [l 0]) (v o i d) ; / * p i s a 10-element array of pointers t o functions;
each function returns an integer quantity */
int (*p[10](char a); /* p i s a 10-element array of pointers t o functions;
each function accepts an argument which i s a character, and
returns an integer quantity */
int * (* p [l 0]) (c h a r a); /* p i s a 10-element array of pointers t o functions;

each function accepts an argument which i s a character, and
returns a pointer t o an integer quantity */
int * (* p [l 0]) (c h a r *a); /* p is a 10-element array of pointers t o functions;
each function accepts an argument which i s a pointer t o a
character, and
returns a pointer t o an integer quantity */

EXAMPLES

***p++ is same as *(p++)**
It will first compute *p (value stored in content of p(address of an item)). Then p will be incremented to point to next location.

Eg. x=*p++;
Or
x=*(p++);
Is equivalent to
x=*p;
p=p+1;
But (*p)++ will first compute *p. Then the value of *p is incremented by 1.

x=(*p)++;
is equivalent to
x=*p;
***p=*p+1;**

```
#include <stdio.h>
main()
{
intarr[] = {10, 20};
int *p = arr;
printf("arr[0] = %d, arr[1] = %d, *p = %d",  arr[0], arr[1], *p);
printf("\narr[0] = %d, arr[1] = %d, (*p)++ = %d",  arr[0], arr[1], (*p)++);
printf("\narr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
   }
```

Output

close

```
arr[0] = 10, arr[1] = 20, *p = 10
arr[0] = 11, arr[1] = 20, (*p)++ = 10
arr[0] = 11, arr[1] = 20, *p = 11
```

```
#include <stdio.h>
main(void)
{
intarr[] = {10, 20};
int *p = arr;
```

```
printf("arr[0] = %d, arr[1] = %d, *p = %d",
                                        arr[0], arr[1], *p);
printf("\narr[0] = %d, arr[1] = %d, *p++ = %d",arr[0], arr[1], *p++);

printf("\narr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
}
```

Output

```
arr[0] = 10, arr[1] = 20, *p = 10
arr[0] = 10, arr[1] = 20, *p++ = 10
arr[0] = 10, arr[1] = 20, *p = 20
```

```
#include <stdio.h>
main(void)
{
intarr[] = {10, 20};
int *p = arr;
printf("arr[0] = %d, arr[1] = %d, *p = %d",arr[0], arr[1], *p);
printf("\narr[0] = %d, arr[1] = %d, *(p++) = %d", arr[0], arr[1], *(p++));
printf("\narr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);    }
```

Output

```
arr[0] = 10, arr[1] = 20, *p = 10
arr[0] = 10, arr[1] = 20, *p++ = 10
arr[0] = 10, arr[1] = 20, *p = 20
```

```
#include <stdio.h>
int main(void)
{
intarr[] = {10, 20,30};
int *p = arr;
   *++p;
printf("arr[0] = %d, arr[1] = %d, arr[2]=%d, *p = %d", arr[0], arr[1],arr[2],  *p);

printf("\narr[0] = %d, arr[1] = %d,arr[2] = %d, *(p++) = %d", arr[0], arr[1], arr[2], *(p++));

printf("\narr[0] = %d, arr[1] = %d,arr[2] = %d, (*p) = %d", arr[0], arr[1], arr[2], *p);
printf("\narr[0] = %d, arr[1] = %d,arr[2] = %d, (*p)++ = %d", arr[0], arr[1],arr[2],  (*p)++);
printf("\narr[0] = %d, arr[1] = %d,arr[2] = %d, (*p) = %d", arr[0], arr[1], arr[2], *p);
  return 0;
}
arr[0] = 10, arr[1] = 20,arr[2] = 30, *p = 20
arr[0] = 10, arr[1] = 20,arr[2] = 30, *(p++) = 20
arr[0] = 10, arr[1] = 20,arr[2] = 30, (*p) = 30
arr[0] = 10, arr[1] = 20,arr[2] = 31, (*p)++ = 30
```

arr[0] = 10, arr[1] = 20,arr[2] = 31, (*p) = 31

Use of pointers

- pointers can be used to pass information back and forth between a function and its reference point
- pointers provide a way to return multiple data items from a function via function arguments.
- permit references to other functions to be specified as arguments to a given function.

EXAMPLE

Read and display matrix using pointer

```
#include<stdio.h>

void disp(int *x[10],intr,int c)
{
inti,j;
printf("\n Matrix r=%d c=%d\n",r,c);
for(i=0;i<r;i++)
{
	printf("\n");
	for(j=0;j<c;j++)
	{
	printf("%d\t",*(*(x+i) +j));
	}
}
}
void read(int *x[10],intr,int c)
{
inti,j;
printf("\n Matrix\n r=%d c=%d\n",r,c);
for(i=0;i<r;i++)
{
	for(j=0;j<c;j++)
	{
```

```
            scanf("%d",(*(x+i) +j));
            }
}
}

main()
{
int *a[10];

inti,j,r1,c1;

printf("\nEnter rows and columns of first matrix: \n");
scanf("%d%d",&r1,&c1);

printf("\nEnterfirstMatrix\n");
for(i=0;i<r1;i++)
{
            a[i]=(int *) malloc(c1*sizeof(int));
}
 read(a,r1,c1);
disp(a,r1,c1);
}
```

Pointer Examples

```
#include<stdio.h>
main()
{
int a=10, b=20, *p, *j;
p=&a;
j=&b;
printf("\nAddition *p + b = %d", *p + b);
printf("\nAddition *p + *j = %d", *p + *j);
printf("\nAddition *(p) + *(j) = %d", *(p) + *(j));
printf("\nAddition *(&a) + *(&b) = %d", *(&a) + *(&b));
}
```

```
Addition *p + b=30
Addition *p + *j =30
Addition *(p) + *(j) =30
Addition *(&a) + *(&b) = 30
```

===

Questions

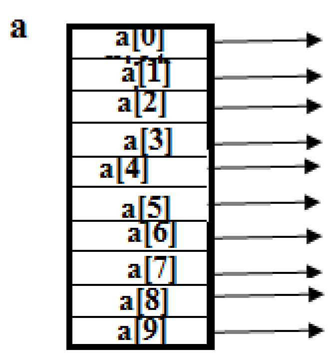
1. A C program contains the following declaration.
   static intx[8]={11,22,33,44,55,66,77,88};
   Int *p=x;

   Suppose address of starting location of array x is 2000.What is the value of the following
   (a) (x+2)
   (b) *(x+2)
   (c) *x+2
   Answer:
   (a) 2004 (because int need 2 bytes)
   (b) 33
   (c) 13
2. What is the meaning of variable a int he following declarations
   (a) float *a;
   (b) int *a[10];
   (c) Int (*a)[10];

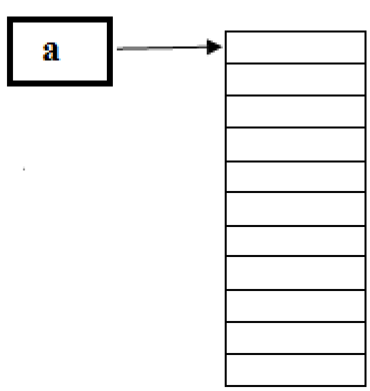
Answer:

(a) a is a floating point type pointer variable

(b) a is an array of 10 integer pointers



(c) a is an integer pointer to a array of 10 integers



3.

```c
#include<stdio.h>
main()
{
inta[4]={10,20,30};
int *p=a;
printf("\na[0]=%d  a[1]=%d  a[2]=%d  a[3]=%d *p++=%d", a[0],a[1],a[2],a[3],*p++);
printf("\na[0]=%d  a[1]=%d  a[2]=%d  a[3]=%d *p=%d", a[0],a[1],a[2],a[3],*p);
}
```

Predict the output of the above code.

What will be the output if the first printf statement is replaced by the following. Write output in each of the following .

a) printf("\na[0]=%d a[1]=%d a[2]=%d a[3]=%d *(p++)=%d", a[0],a[1],a[2],a[3],*(p++));

b) printf("\na[0]=%d a[1]=%d a[2]=%d a[3]=%d (*p)++=%d", a[0],a[1],a[2],a[3],(*p)++);

Answers

*a[0]=10 a[1]=20 a[2]=30 a[3]=0 *p++=10*
*a[0]=10 a[1]=20 a[2]=30 a[3]=0 *p=20*

*a) a[0]=10 a[1]=20 a[2]=30 a[3]=0 *(p++)=10*
*a[0]=10 a[1]=20 a[2]=30 a[3]=0 *p=20*

*b) a[0]=11 a[1]=20 a[2]=30 a[3]=0 (*p)++=10*
*a[0]=11 a[1]=20 a[2]=30 a[3]=0 *p=11*

*Note:*p++ is same as *(p++) it Is equivalent to*
*Taking *p then p=p+1*
*(*p)++ means take *p then increment that value by 1*