CL101
INTRODUCTION TO COMPUTING

LAB 09

POINTERS & DYNAMIC MEMORY MANAGEMENT IN C

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

POINTER

Pointer is a variable whose value is a memory address. Normally, a variable directly contains a specific value. A pointer contains the memory address of a variable that, in turn, contains a specific value. In this sense, a variable name directly references a value, and a pointer indirectly references a value.

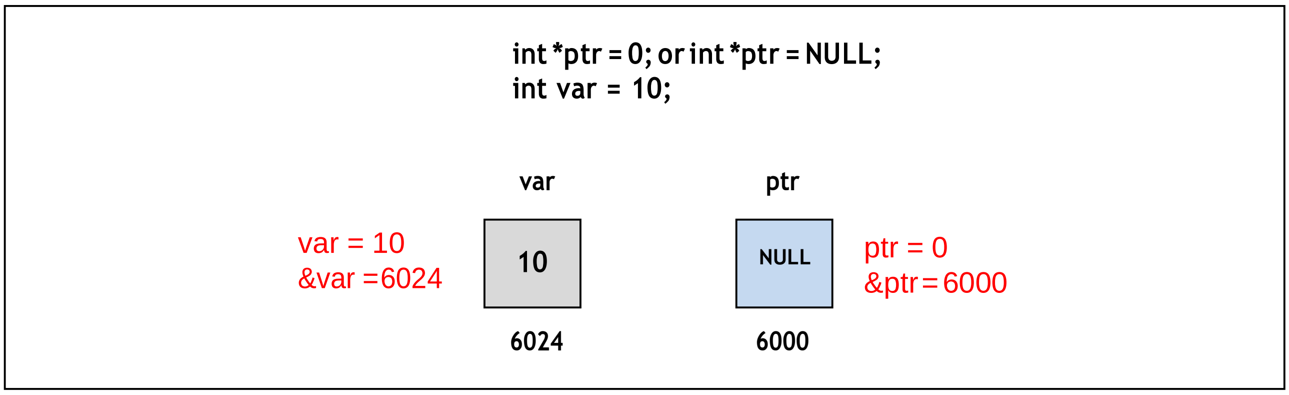
POINTER TYPE DECLARATION

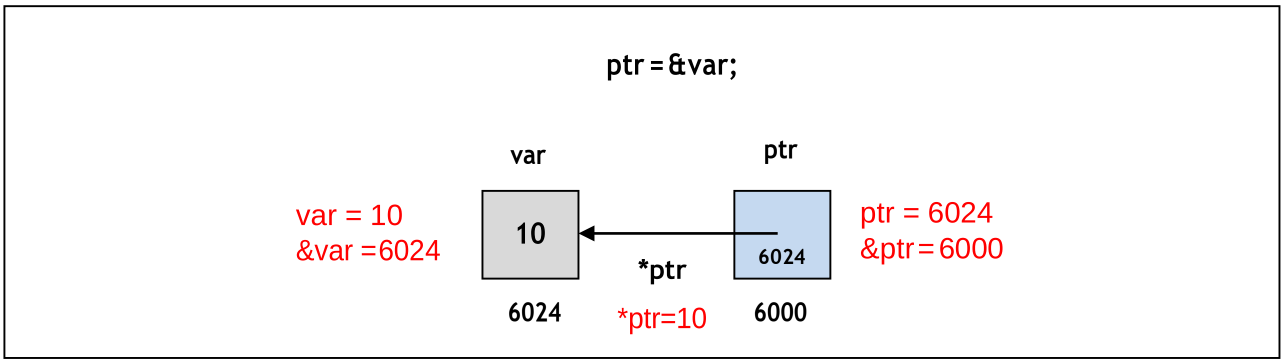
SYNTAX: type * variable ;

EXAMPLE: int *ptr;

INTERPRETATION: The value of the pointer variable ptr is a memory address. A data item whose address is stored in this variable must be of the specified type.

MEMORY REPRESENTATION





sizeof Operator

- Operator sizeof determines the size in bytes of a variable or type at compilation time.
- When applied to the name of an array, sizeof returns the total number of bytes in the array.

POINTER ARITHMETICS

- A limited set of arithmetic operations may be performed on pointers. A pointer may be incremented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may be subtracted from a pointer (- or -=) and one pointer may be subtracted from another.
- When an integer is added to or subtracted from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.
- Two pointers to elements of the same array may be subtracted from one another to determine the number of elements between them.

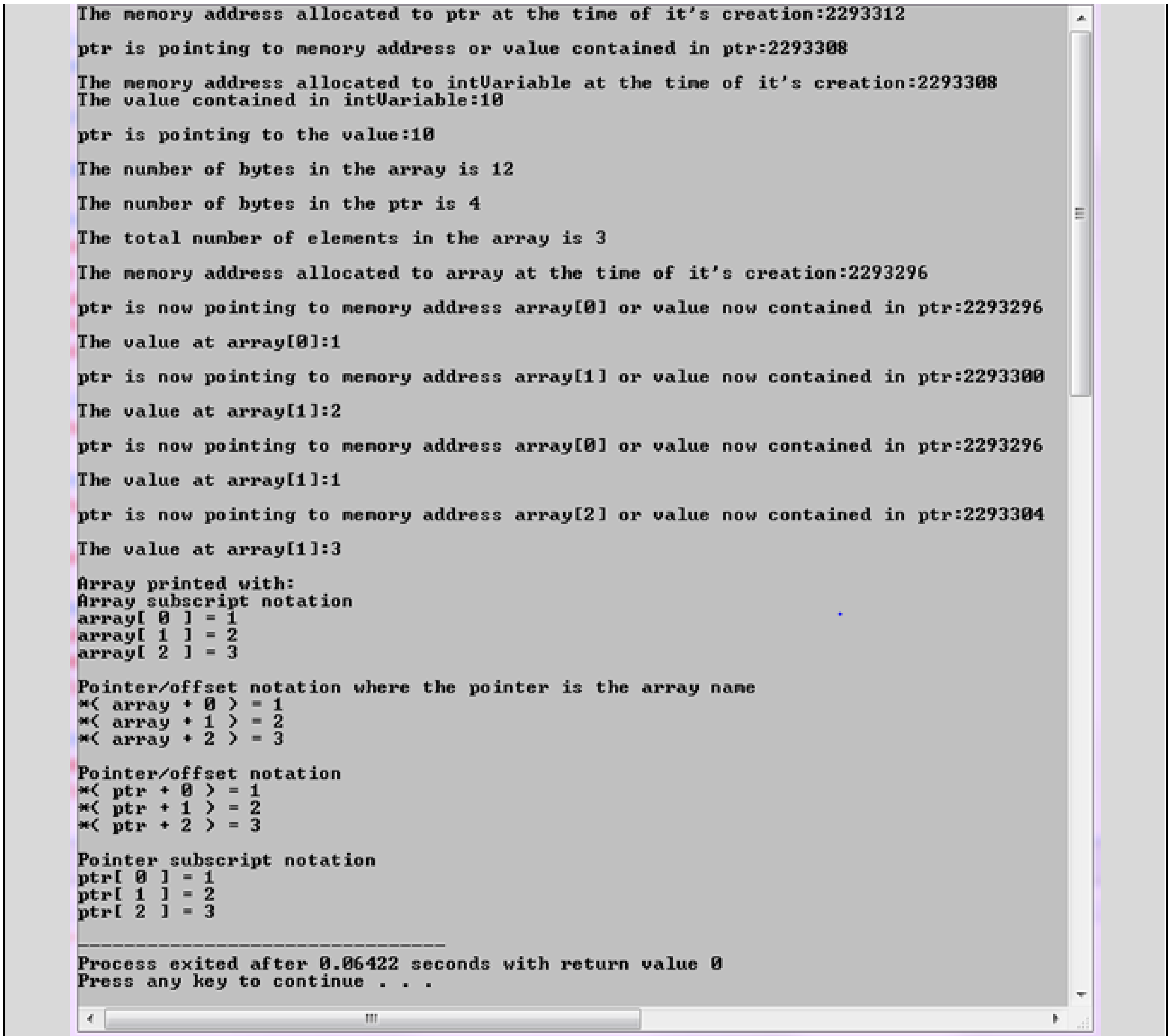
RELATIONSHIP BETWEEN POINTERS AND ARRAYS

Arrays and pointers are intimately related in C and often may be used interchangeably.

- An array name can be thought of as a constant pointer.
- Pointers can be used to do any operation involving array subscripting.
- When a pointer points to the beginning of an array, adding an offset to the pointer indicates which element of the array should be referenced, and the offset value is identical to the array subscript. This is referred to as pointer/offset notation.
- An array name can be treated as a pointer and used in pointer arithmetic expressions that do not attempt to modify the address of the pointer.
- Pointers can be subscripted exactly as arrays can. This is referred to as pointer/subscript notation.

SAMPLE CODE

```
#include <stdio.h>

int main()
{
        int *ptr = NULL; // pointer variable of type “pointer to int" / null pointer
        int intVariable1 = 10; // Declare an integer variable and initialize it with 10

        // Use address-of operator & to assign memory address of intVariable1 to a pointer
        ptr = &intVariable1;
        // Pointer ptr now holds a memory address of intVariable

        // Print out associated memory addresses and their values
        printf("The memory address allocated to ptr at the time of its creation:%d\n",&ptr);
        printf("\nptr is pointing to memory address or value contained in ptr:%d\n",ptr);
        printf("\nThe memory address allocated to intVariable at the time of its creation:%d\n",&intVariable1);
        printf("The value contained in intVariable:%d\n" ,intVariable1);
        printf("\nptr is pointing to the value:%d\n" , *ptr);

        int array[3] = {1,2,3},offset,i; //Initialize an array of three elements
        printf("\nThe number of bytes in the array is %d\n", sizeof(array));

        ptr = array; // Assign memory address of arr to pointer
        printf("\nThe number of bytes in the ptr is %d\n", sizeof(*ptr));

       printf("\nThe total number of elements in the array is %d\n", sizeof(array)/sizeof(*ptr));

        // Print out associated memory addresses and their values
        printf("\nThe memory address allocated to array at the time of it's creation:%d\n", array);
        printf("\nptr is now pointing to memory address array[0] or value now contained in ptr:%d\n",ptr);
        printf("\nThe value at array[0]:%d\n",*ptr);

        ptr++; //Adds 4 to the value(address) contained in ptr i.e to address of array[0] and now contains the address of array[1]
        printf("\nptr is now pointing to memory address array[1] or value now contained in ptr:%d\n",ptr);
        printf("\nThe value at array[1]:%d\n",*ptr);

        ptr--; //Subtracts 4 from the value(address) contained in ptr i.e from address of array[1] and now contains the address of array[0]
        printf("\nptr is now pointing to memory address array[0] or value now contained in ptr:%d\n",ptr);
        printf("\nThe value at array[1]:%d\n",*ptr);

        ptr = ptr+2; //Adds 4 to the value(address) contained in ptr i.e to address of array[0] and now contains the address of array[2]
        printf("\nptr is now pointing to memory address array[2] or value now contained in ptr:%d\n",ptr);
        printf("\nThe value at array[1]:%d\n",*ptr);

        ptr = ptr - 2;

        // Displaying array using array subscript notation
        printf( "\nArray printed with:\nArray subscript notation\n" );
        for ( i = 0; i < 3; ++i )
        {
                printf( "array[ %d ] = %d\n", i,array[ i ]);
        }

        // Displaying array using array name and pointer/offset notation
        printf("\nPointer/offset notation where the pointer is the array name\n");
        for ( offset = 0; offset < 3; ++offset )
        {
                printf( "*( array + %d ) = %d\n", offset, *(array + offset) );
        }

        //Displaying array using ptr and pointer/offset notation
        printf( "\nPointer/offset notation\n" );
        for ( offset = 0; offset < 3; ++offset )
        {
                printf( "*( ptr + %d ) = %d\n", offset, *(ptr + offset) );
        }

        //Displaying array using ptr and array subscript notation
        printf( "\nPointer subscript notation\n" );
        for ( i = 0; i < 3; ++i )
        {
                printf( "ptr[ %d ] = %d\n", i,ptr[ i ]);
        }

        return 0;
}
```



```
The memory address allocated to ptr at the time of it's creation:2293312

ptr is pointing to memory address or value contained in ptr:2293308

The memory address allocated to intVariable at the time of it's creation:2293308
The value contained in intVariable:10

ptr is pointing to the value:10

The number of bytes in the array is 12

The number of bytes in the ptr is 4

The total number of elements in the array is 3

The memory address allocated to array at the time of it's creation:2293296

ptr is now pointing to memory address array[0] or value now contained in ptr:2293296

The value at array[0]:1

ptr is now pointing to memory address array[1] or value now contained in ptr:2293300

The value at array[1]:2

ptr is now pointing to memory address array[0] or value now contained in ptr:2293296

The value at array[1]:1

ptr is now pointing to memory address array[2] or value now contained in ptr:2293304

The value at array[1]:3

Array printed with:
Array subscript notation
array[ 0 ] = 1
array[ 1 ] = 2
array[ 2 ] = 3

Pointer/offset notation where the pointer is the array name
*( array + 0 ) = 1
*( array + 1 ) = 2
*( array + 2 ) = 3

Pointer/offset notation
*( ptr + 0 ) = 1
*( ptr + 1 ) = 2
*( ptr + 2 ) = 3

Pointer subscript notation
ptr[ 0 ] = 1
ptr[ 1 ] = 2
ptr[ 2 ] = 3

--------------------------------
Process exited after 0.06422 seconds with return value 0
Press any key to continue . . .
```

OUTPUT: SAMPLE CODE

DIFFERENCE BETWEEN POINTERS AND ARRAYS

- Assigning any address to an array variable is not allowed while address can be assigned to a pointer variable using address operator.
- A pointer is a place in memory that keeps address of another place inside while an array is a single, pre allocated chunk of contiguous elements (all of the same type), fixed in size and location.
- Pointer is dynamic in nature. The memory allocation can be resized or freed later while the arrays are static in nature. Once memory is allocated, it cannot be resized or freed dynamically.

SIGNIFICANCE OF POINTERS

- Pointers allow sharing without copying. This provides advantage when passing around big arrays as arguments to functions.
- Pointers allow modifications by a function that is not the creator of the memory i.e. function A can allocate the memory and function C can modify it, without using global variables, which is a not good for safe programming.
- Pointers allow us to use dynamic memory allocation.
- Pointers allow us to resize the data structure whenever needed. For example, if you have an array of size 10, it cannot be resized. But, an array created out of malloc and assigned to a pointer can be resized easily by creating a new memory area through malloc and copying the old contents over.

DYNAMIC MEMORY MANAGEMENT

C enables programmers to control the allocation and deallocation of memory in a program for any built in or user defined type.
The ability for a program to obtain more memory space at execution time to hold new nodes, and to release space no longer needed is known as dynamic memory management.

WHY IS IT REQUIRED?

Many times, it is not known in advance how much memory will be needed to store particular information in a defined variable and the size of required memory can be determined at run time. For example, we may want to hold someone's name, but we do not know how long their name is until they enter it. Or we may want to read in a number of records from disk, but we don't know in advance how many records there are. Or we may be creating a game, with a variable number of monsters (that changes over time as some monsters die and new ones are spawned) trying to kill the player.

FUNCTIONS USED FOR DYNAMIC MEMORY ALLOCATION (MALLOC & FREE)

MALLOC

In order to allocate a variable dynamically, the C memory allocation function malloc is used, which resides in the stdlib library. This function requires a single argument, that is, a number indicating the amount of memory space needed.

```
malloc(sizeof (int))
```

allocates exactly enough space to hold one type int value and returns a pointer to (the address of) the block allocated. When we work with pointers in C, we always deal with a "pointer to some specific type," rather than simply a "pointer." Therefore, the data type (void *) of the value returned by malloc should always be cast to the specific type we need.

```
int *num;
num = (int *)malloc(sizeof (int));
```

Dynamically allocating memory in this fashion causes an array (or any other built in or user defined type) to be created in the free store (sometimes called the heap) – a region of memory assigned to each program for storing objects created at execution time.

FREE

Function free deallocates memory—i.e., the memory is returned to the system so that it can be reallocated in the future. To free memory dynamically allocated by the preceding malloc call, use the statement

```
free( num );
```

DYNAMIC ARRAY ALLOCATION WITH CALLOC

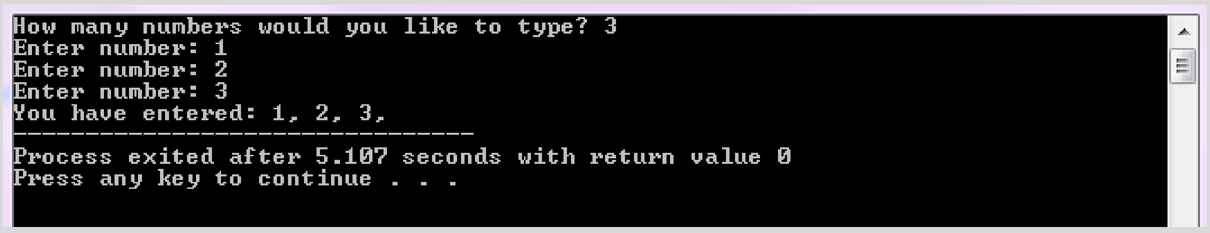
The name calloc stands for "contiguous allocation".

Function malloc can be used to allocate a single memory block of any built-in or user-defined type. To dynamically create an array of elements of any built-in or user-defined type, we use the contiguous allocation function from stdlib, calloc .Function calloc takes two arguments: The number of array elements needed and the size of one element.

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

EXAMPLE(DYNAMIC 1D-ARRAY ALLOCATION USING CALLOC)

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

int main ()
{
        int numCount,n;
        printf("How many numbers would you like to type? ");
        scanf("%d",&numCount);

        int *ptr = (int *)calloc(numCount, sizeof (int)); /* returns a pointer to a section of memory just large
enough to hold the integers, whose quantity is being obtained through user input stored in the variable
numCount. */

        for (n=0; n< numCount; n++)
        {
                printf("Enter number: ");
                scanf("%d",&ptr[n]);
        }
        printf("You have entered: ");
        for (n=0; n<numCount; n++)
        printf("%d, ",ptr[n] );
        free(ptr); //returns to the system whatever memory was pointed to by ptr.

        return 0;
}
```



```
How many numbers would you like to type? 3
Enter number: 1
Enter number: 2
Enter number: 3
You have entered: 1, 2, 3,
--------------------------------
Process exited after 5.107 seconds with return value 0
Press any key to continue . . .
```

OUTPUT: EXAMPLE(DYNAMIC 1D-ARRAY ALLOCATION USING CALLOC)

POINTER TO POINTER (DOUBLE POINTER)

A pointer to pointer variable can hold the address of another pointer variable. It can be declared as:

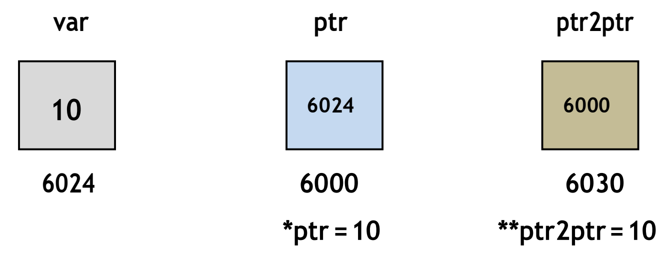
POINTER TO POINTER DECLARATION

SYNTAX: type ** variable ;

EXAMPLE: int **ptr2ptr;

INTERPRETATION: The value of the pointer variable ptr2ptr is a memory address of another pointer.

```
int var = 10, *ptr, **ptr2ptr;
ptr = &var;
ptr2ptr = &ptr;
```

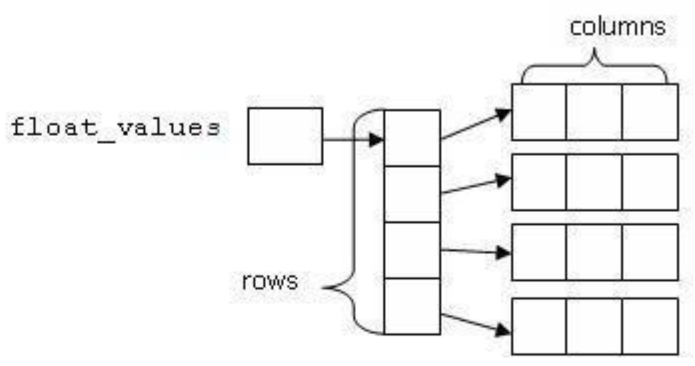


MEMORY REPRESENTATION

Think of the memory to be allocated to a pointer to pointer variable as two dimensional. It has 'rows' and 'columns'; i.e., if the size is m x n, there will be 'm' rows, and for each row, there will be 'n' columns.

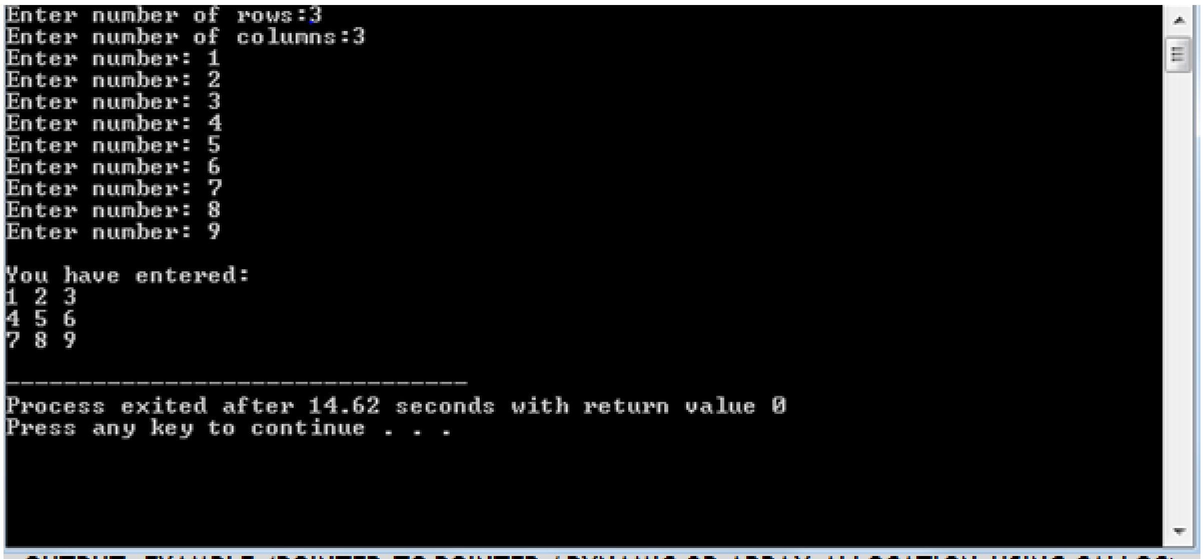
- First, allocate memory for 'm' rows.
- Secondly, allocate 'n' columns for each of the 'm' rows.

Allocating memory to a pointer to pointer to float values. Let the number of rows be '4' and the number of columns '3'.



EXAMPLE(POINTER TO POINTER / DYNAMIC 2D-ARRAY ALLOCATION USING CALLOC)

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

int main()
{
	int rowCount,colCount,row,col;

	printf("Enter number of rows:");
	scanf("%d",&rowCount);

	printf("Enter number of columns:");
	scanf("%d",&colCount);

	//allocate memory for rows
	int **ptr = (int **)calloc(rowCount, sizeof (int*));

	//for each row allocate memory for columns
	for(row=0;row<rowCount;row++)
		*(ptr+row) = calloc(colCount,sizeof(int));

	//Storing elements
	for(row=0;row< rowCount;row++)
		for(col=0;col<colCount;col++)
		{
			printf("Enter number: ");
			scanf("%d",&ptr[row][col]);
		}

	//Displaying pointer to pointer to ptr
	printf("\nYou have entered:\n");
	for (row=0; row< rowCount; row++)
	{
		for(col=0;col<colCount;col++)
		{
			printf("%d ",ptr[row][col]);
		}
		printf("\n");
	}

	free(ptr); //returns to the system whatever memory was pointed to by ptr.

	return 0;
}
```



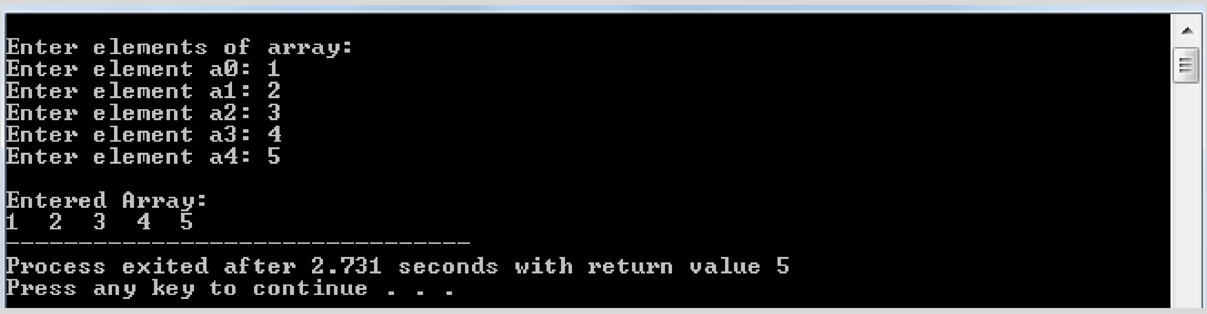
```
Enter number of rows:3
Enter number of columns:3
Enter number: 1
Enter number: 2
Enter number: 3
Enter number: 4
Enter number: 5
Enter number: 6
Enter number: 7
Enter number: 8
Enter number: 9

You have entered:
1 2 3
4 5 6
7 8 9

--------------------------------
Process exited after 14.62 seconds with return value 0
Press any key to continue . . .
```

OUTPUT: EXAMPLE (POINTER TO POINTER / DYNAMIC 2D-ARRAY ALLOCATION USING CALLOC)

RETURNING ONE DIMENSIONAL ARRAY FROM FUNCTION

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

int* Input_Matrix(int size)
{
        int i=0;
        int* ptr = (int *)calloc(size,sizeof(int));

        // Storing elements of the array
        printf("\nEnter elements of array:\n");
        for(i=0; i<size;++i)
        {
                printf("Enter element a%d: ",i);
                scanf("%d", &ptr[i]);
        }
        return ptr;
}

void Display_Matrix(int *array, int size)
{
        int i;
        printf("\nEntered Array: \n");
        for(i=0; i<size; i++)
        {
                  printf("%d ", array[i]);
        }
}
```

```
int main()
{
        int size = 5;
        int *input = Input_Matrix(size);
        Display_Matrix(input,size);
        return 0;
}
```



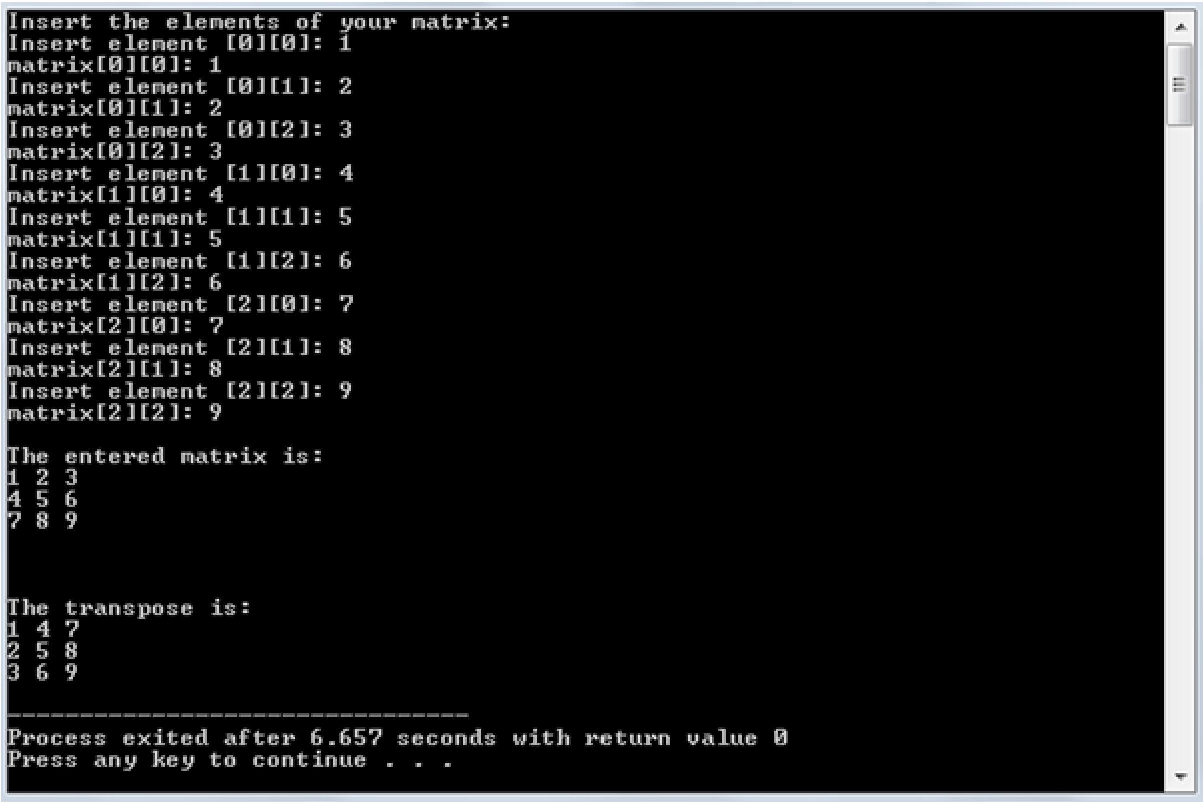
```
Enter elements of array:
Enter element a0: 1
Enter element a1: 2
Enter element a2: 3
Enter element a3: 4
Enter element a4: 5

Entered Array:
1  2  3  4  5
--------------------------------
Process exited after 2.731 seconds with return value 5
Press any key to continue . . .
```

OUTPUT: RETURNING ONE DIMENSIONAL ARRAY FROM FUNCTION

RETURNING TWO DIMENSIONAL ARRAY FROM FUNCTION

```
#include <stdio.h>
#include  <stdlib.h>
int i,j;

int** callocMatrix(int rmax, int colmax)
{
  //allocate memory for rows
        int **mat = (int **)calloc(rmax, sizeof(int));

  //for each row allocate memory for columns
  for(i = 0; i < rmax; i++)
        *(mat+i) = (int *)calloc(colmax, sizeof(int));

  return mat;
}

// fill matrix
void setMatrix(int **mat, int r, int c)
{
  printf("Insert the elements of your matrix:\n");
  for (i = 0; i < r; i++)
   {
     for (j = 0; j < c; j++)
     {
       printf("Insert element [%d][%d]: ", i, j);
       scanf("%d", &mat[i][j]);
       printf("matrix[%d][%d]: %d\n", i, j, mat[i][j]);
     }
   }
```

```c
}

// print matrix
void printMatrix(int **mat, int r, int c)
{

   for (i=0; i<r;i++)
   {
      for (j=0; j<c;j++)
      {
         printf("%d ", mat[i][j]);
      }
      printf("\n");
   }
}

void Transpose(int **transpose, int **mat, int r, int c)
{
            for (i=0; i<r;i++)
            {
                        for (j=0; j<c;j++)
                        {
                                    transpose[j][i] = mat[i][j];
                        }
                        printf("\n");
            }
}

int main()
{
   int r = 3, c = 3;
   int**mat=callocMatrix(r,c);
   setMatrix(mat, r,c);
   printf("\nThe entered matrix is:\n");
   printMatrix(mat, r,c);
   int **transpose = callocMatrix(r,c);
   Transpose(transpose, mat, r, c);
   printf("The transpose is:\n");
   printMatrix(transpose, r, c);
   return 0;
}
```



```
Insert the elements of your matrix:
Insert element [0][0]: 1
matrix[0][0]: 1
Insert element [0][1]: 2
matrix[0][1]: 2
Insert element [0][2]: 3
matrix[0][2]: 3
Insert element [1][0]: 4
matrix[1][0]: 4
Insert element [1][1]: 5
matrix[1][1]: 5
Insert element [1][2]: 6
matrix[1][2]: 6
Insert element [2][0]: 7
matrix[2][0]: 7
Insert element [2][1]: 8
matrix[2][1]: 8
Insert element [2][2]: 9
matrix[2][2]: 9

The entered matrix is:
1 2 3
4 5 6
7 8 9



The transpose is:
1 4 7
2 5 8
3 6 9

--------------------------------
Process exited after 6.657 seconds with return value 0
Press any key to continue . . .
```

OUTPUT: RETURNING TWO DIMENSIONAL ARRAY FROM FUNCTION