```c
#include <stdio.h>
#include <stdlib.h> //contains prototypes of malloc() and free()

int main(void)
{
    //variable declarations
    int *ptr_iArray = NULL; //IT IS GOOD DISCIPLINE TO INITIALIZE ANY POINTER WITH NULL ADDRESS TO PREVENT ANY GARBAGE VALUE GETTING INTO IT, THAT WAY, IT MAKES IT EASY TO CHECK FOR SUCCESS OR FAILURE OF MEMORY ALLOCATION LATER ON AFTER malloc()...
    unsigned int intArrayLength = 0;
    int i;

    //code
    printf("\n\n");
    printf("Enter The Number Of Elements You Want In Your Integer Array : ");
    scanf("%d", &intArrayLength);

    // ****** ALLOCATING AS MUCH MEMORY REQUIRED TO THE INTEGER ARRAY ******
    // ****** MEMORY REQUIRED FOR INTEGER ARRAY = SIZE IN BYTES OF ONE INTEGER * NUMBER OF INTEGERS TO BE STORED IN ARRAY ******
    // ****** TO ALLOCATE SAID AMOUNT OF MEMORY, FUNCTION malloc() WILL BE USED ******
    // ****** malloc() WILL ALLOCATE SAID AMOUNT OF MEMORY AND WILL RETURN THE INITIAL / STARTING / BASE ADDRESS OF THE ALLOCATED MEMORY, WHICH MUST BE CAPTURED IN A POINTER VARIABLE ******
    // ****** USING THIS BASE ADDRESS, THE INTEGER ARRAY CAN BE ACCESSED AND USED ******

    ptr_iArray = (int *)malloc(sizeof(int) * intArrayLength);
    if (ptr_iArray == NULL) //IF ptr_iArray IS STILL NULL, EVEN AFTER CALL TO malloc(), IT MEANS malloc() HAS FAILED TO ALLOCATE MEMORY AND NO ADDRESS HAS BEEN RETURNED BY malloc() in ptr_iArray...
    {
        printf("\n\n");
        printf("MEMORY ALLOCATION FOR INTEGER ARRAY HAS FAILED !!! EXITTING NOW...\n\n");
        exit(0);
    }
    else //IF ptr_iArray IS NOT NULL, IT MEANS IT MUST CONTAIN A VALID ADDRESS WHICH IS RETURNED BY malloc(), HENCE, malloc() HAS SUCCEEDED IN MEMORY ALLOCATION...
    {
        printf("\n\n");
        printf("MEMORY ALLOCATION FOR INTEGER ARRAY HAS SUCCEEDED !!!\n\n");
        printf("MEMORY ADDRESSES FROM %p TO %p HAVE BEEN ALLOCATED TO INTEGER ARRAY !!!\n\n", ptr_iArray, (ptr_iArray + (intArrayLength - 1)));
    }

    printf("\n\n");
    printf("Enter %d Elements For The Integer Array : \n\n", intArrayLength);
    for (i = 0; i < intArrayLength; i++)
        scanf("%d", (ptr_iArray + i));

    printf("\n\n");
    printf("The Integer Array Entered By You, Consisting Of %d Elements : \n
\n", intArrayLength);
    for (i = 0; i < intArrayLength; i++)
    {
        printf("ptr_iArray[%d] = %d \t \t At Address &ptr_iArray[%d] : %p\n", i, ptr_iArray[i], i, &ptr_iArray[i]);
    }

    printf("\n\n");
    for (i = 0; i < intArrayLength; i++)
    {
        printf("*(ptr_iArray + %d) = %d \t \t At Address (ptr_iArray + %d) : %p\n", i, *(ptr_iArray + i), i, (ptr_iArray + i));
    }

    // ***** CHECKING IF MEMORY IS STILL ALLOCATED BY CHECKING VALIDITY OF BASE ADDRESS 'ptr_iArray' ******
    // ***** IF ADDRESS IS VALID, THAT IS IF 'ptr_iArray' EXISTS, THAT IS, IF IT IS NOT NULL, MEMORY IS STILL ALLOCATED ******
    // ***** IN THAT CASE, THE ALLOCATED MEMORY MUST BE FREED ******
    // ***** MEMORY IS ALLOCATED USING malloc() AND FREED USING free() ******
    // ***** ONCE MEMORY IS FREED USING free(), THE BASE ADDRESS MUST BE CLEANED, THAT IS, IT MUST BE RE-INITILAIZED TO 'NULL' TO KEEP AWAY GARBAGE VALUES. THIS IS NOT COMPULSORY, BUT IT IS GOOD CODING DISCIPLINE ******

    if (ptr_iArray)
    {
        free(ptr_iArray);
        ptr_iArray = NULL;

        printf("\n\n");
        printf("MEMORY ALLOCATED FOR INTEGER ARRAY HAS BEEN SUCCESSFULLY FREED !!!\n\n");
    }

    return(0);
}

```