| **CL1002 Programming Fundamentals** | **LAB 10**  **Pointers & Dynamic Memory** | |
| --- | --- | --- |
|  | |  |

**Introduction to Pointers**

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 Declaration & Initialization**

**Syntax:** type \* variable;

**Code:** int \*ptr = 0; // Pointer Declaration

int var= 10;

ptr = &var; // Pointer Initialization

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.



**ptr = &var;**





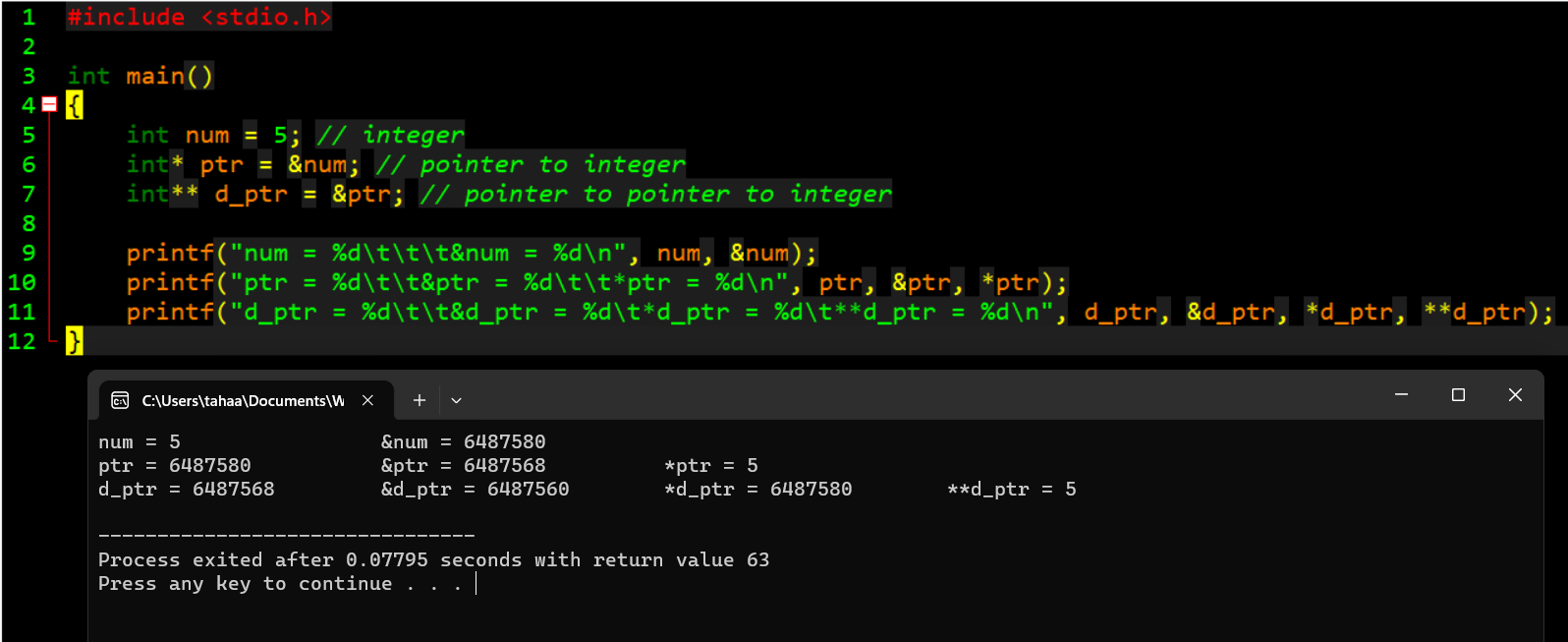






**Double Pointers:**

The same way int\* means pointer to integer, int\*\* means pointer to pointer to integer.



**POINTER ARITHMETICS**

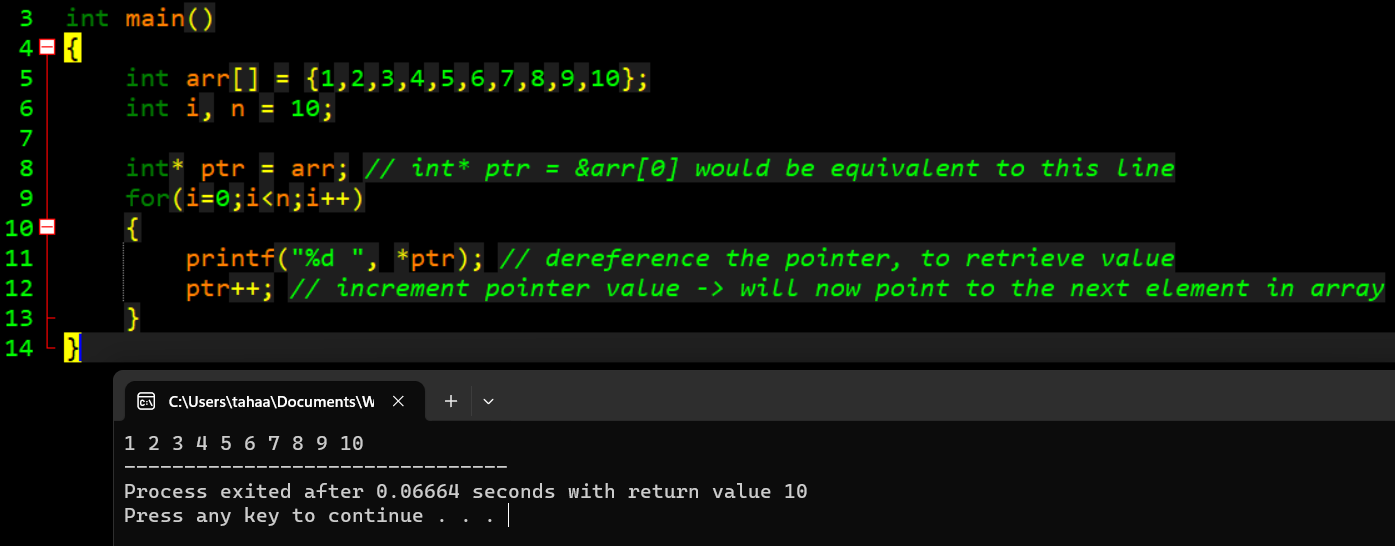
* A limited set of arithmetic operations may be performed on pointers. A pointer may be incremented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may be subtracted from a pointer (- or -=) and one pointer may be subtracted from another.
* When an integer is added 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.

**Accessing Array using Pointers**

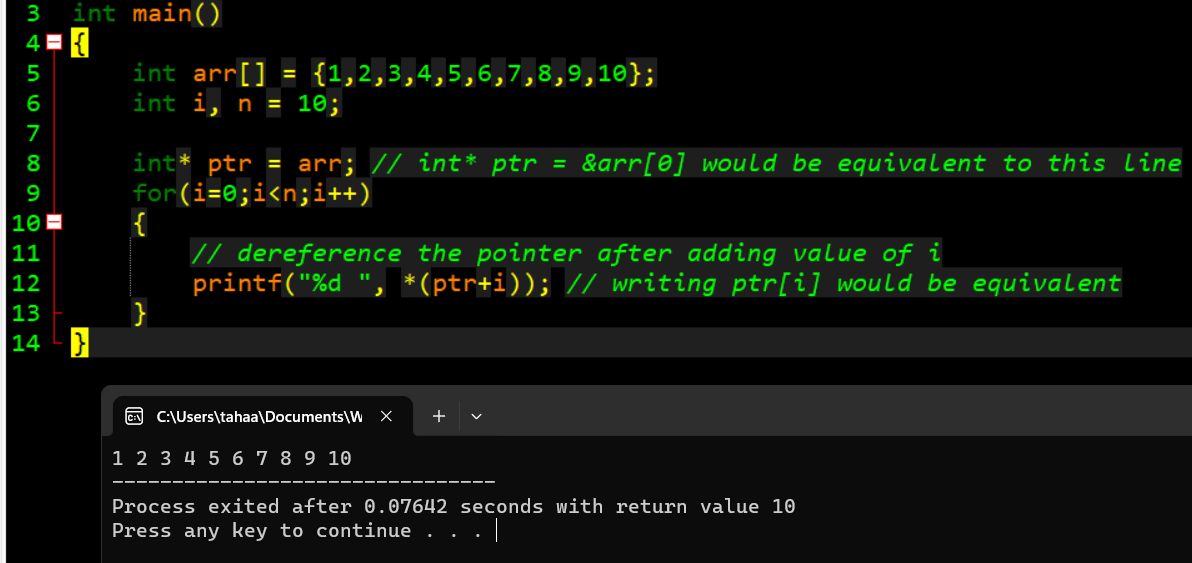
Arrays and pointers are intimately related in C and often may be used interchangeably.

* 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.

**By assigning a pointer to the start of array, we can loop the entire array by incrementing the pointer:**



**The same operation can be performed by adding the value of i in each iteration, while preserving the base value of ptr (ptr will not be modified):**

****

**Dynamic Memory**

The process of allocating memory during program execution is called dynamic memory allocation. 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.

# **Difference between static memory allocation and dynamic memory allocation in C**

| **Static memory allocation** | **Dynamic memory allocation** |
| --- | --- |
| In static memory allocation, memory is allocated before program execution. | In dynamic memory allocation, memory is allocated while executing the program. That means at run time. |
| Memory size **can’t** be modified during execution.  Example: array (fixed size) | Memory size **can** be modified during execution.  Example: Linked list (variable sized) |

**Importance of Dynamic memory**

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.

C language offers 4 dynamic memory allocation functions. They are,

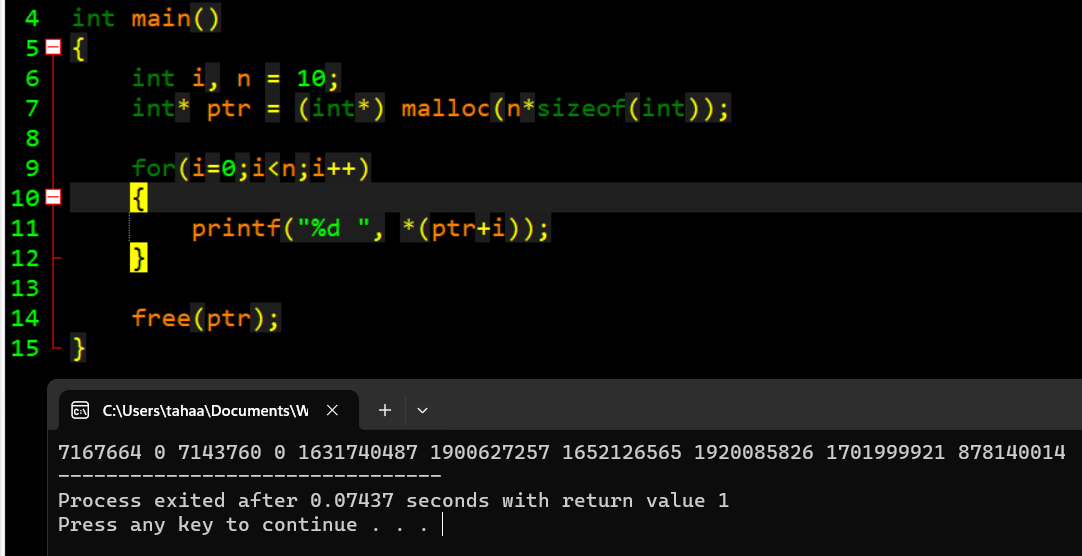
| **Function** | **Syntax** |
| --- | --- |
| malloc () | malloc (num\_elements \*sizeof(int)); |
| calloc () | calloc (num\_elements, sizeof(int)); |
| realloc () | realloc (pointer\_name, num\_elements \* sizeof(int)); |
| free () | free (pointer\_name); |

**Note that malloc, calloc & realloc return void pointers. So they must be casted according to the desired type of array.**

**Free() is used to return the allocated memory back to the system.**

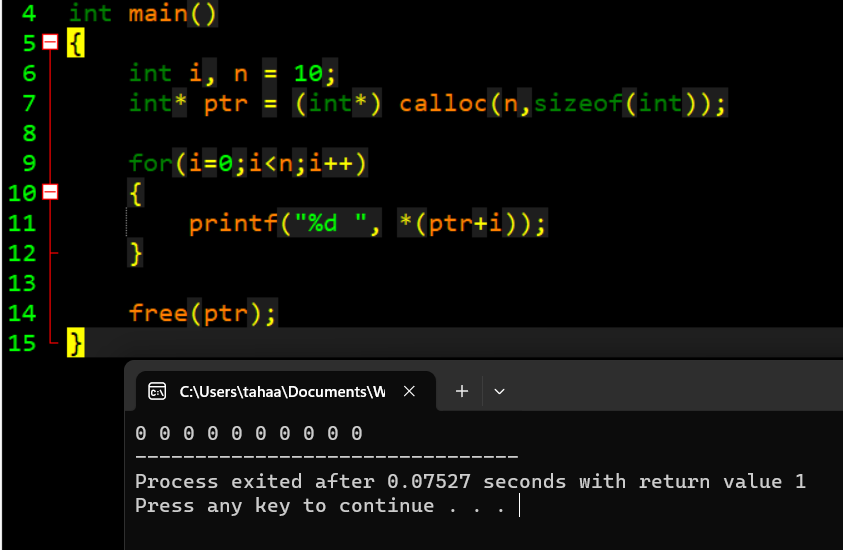
**Malloc()**

Note that malloc does not initialise the allocated memory. Therefore garbage values are printed.



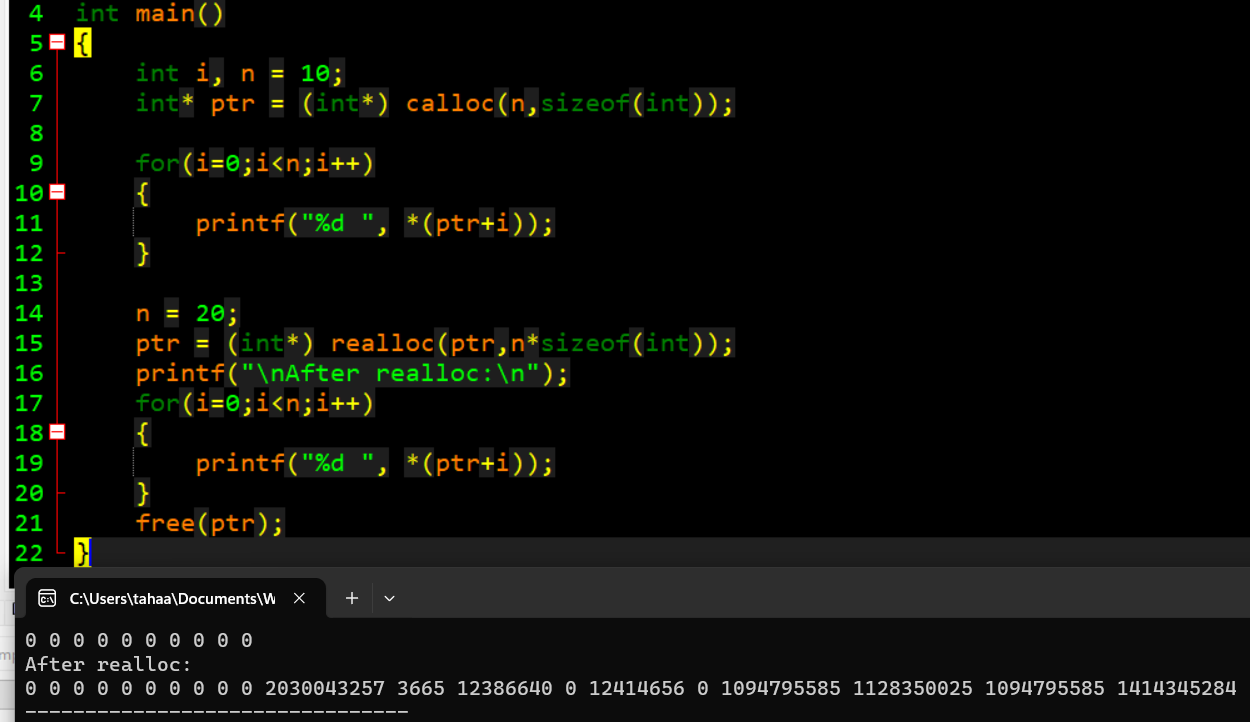
**Calloc()**

Note that calloc initialises the allocated memory to 0.



**Realloc()**

In case the previously allocated memory is not enough, realloc will allocate a new memory to the same pointer, according to the desired size. Realloc also does not initialise the allocated memory to 0. It works similarly to malloc. Therefore garbage values are assigned.



Note that when using realloc(), it is not necessary that the old memory will be extended/reduced. It is possible that completely new memory will be allocated for the entire array.

**Dynamic 2D Arrays**

2D arrays can be understood as “many 1D arrays”. Using this concept, dynamic memory allocation is possible for 2D arrays. Think about it. If 1 pointer is used to point to a 1D array, this means we will need multiple pointers pointing to “many 1D arrays”. Therefore this makes a 2D array.

For a 1D array, we used to write:

**int\* ptr = (int\*) calloc(rows,sizeof(int));**

Now we need multiple 1D arrays, therefore we need multiple pointers. So first, we must dynamically allocate an “array of pointers”. For example, for a 4x3 2D array we need 4 pointers (1 pointer for each row). You can consider each row as a 1D array. See below:

**int\*\* ptr = (int\*\*) calloc(4,sizeof(int\*));**

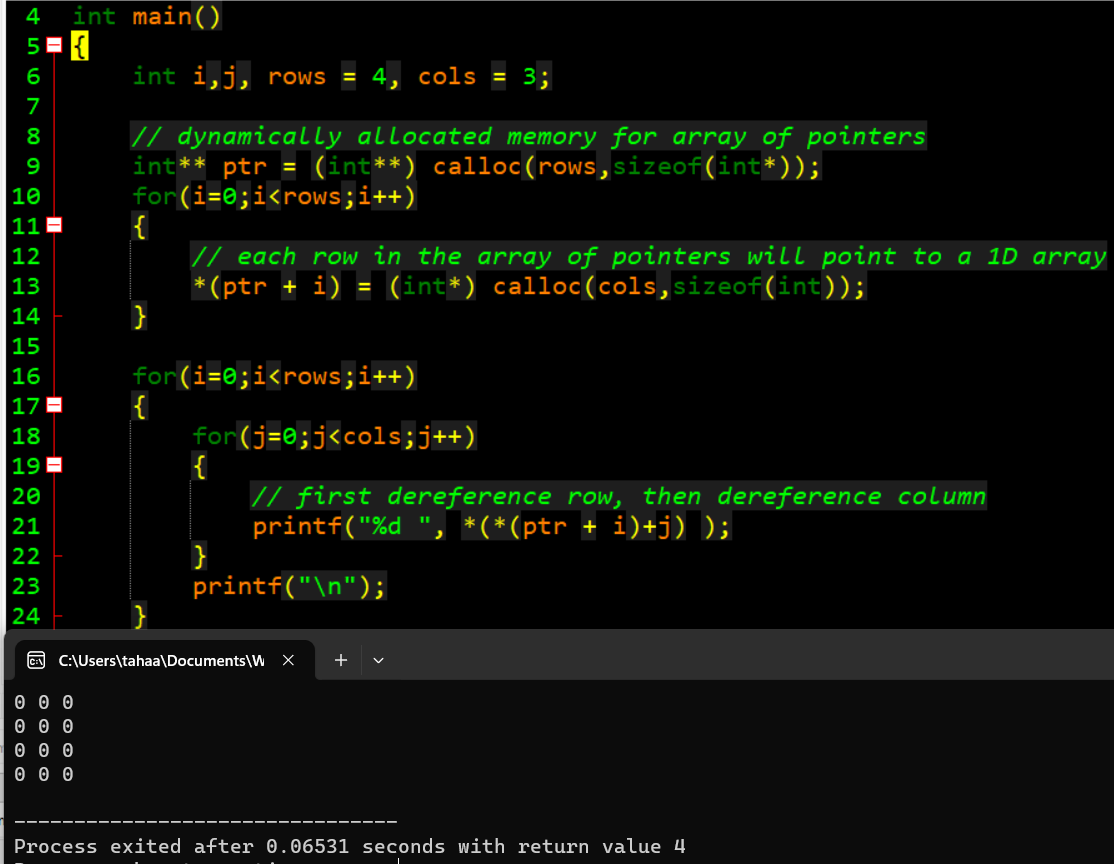
Notice carefully the difference between this and the previous initialization. We must declare a pointer to pointer since we are dealing with 2D arrays. And the allocated array has 4 integer pointers, initialized to 0 due to calloc().

This part only allocates the 4 pointers. Now for each of these 4 pointers, we must dynamically allocate the “many 1D arrays”. For a 4x3 2D array, we can think of it as 4 different 1D arrays, each of length 3. Since we have 4 pointers, we would loop 4 times and allocate 1D arrays. See below:

**for(i=0;i<rows;i++)**

**\*(ptr + i) = (int\*) calloc(3,sizeof(int));**

**This entire process is shown in the code snippet below. Including how to dereference elements of a 2D array using \* operator.**



When releasing the space for this 2D array using free(), the rows (1D arrays) will have to be freed first, and then the array of pointers. See below:

