Introduction to Pointers

A pointer in C is a variable that stores the memory address of another variable. It is an essential feature for dynamic memory management and manipulating data structures like arrays and linked lists.

Operators

Referencing Operator (&)

The referencing operator & appears in front of a variable and is applied to a variable of any data type. It retrieves the memory address of the respective variable.

Dereferencing Operator (*)

The dereferencing operator * appears in front of a variable and is applied to a pointer variable. It retrieves the value stored at the memory address indicated by the pointer.

Pointer Declaration

Declaring a pointer does not allocate a memory space where data can be stored. A pointer is still a data type whose value is a number representing a **memory address**.

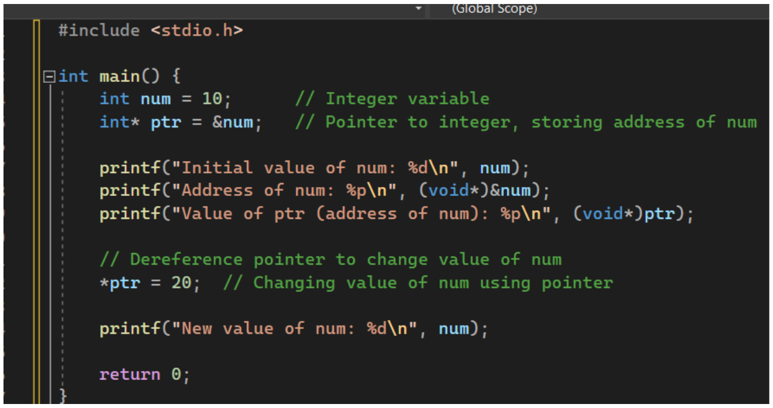
For dereferencing to succeed, the pointer must point to a **valid memory address**, which the program can access. This address could be that of a variable declared earlier or the address of a dynamically allocated memory block (as we will see later).

It is recommended to initialize pointers with the constant **`NULL`**, which is compatible with any type of pointer and indicates, by convention, an **uninitialized pointer**.

Exercise 1: Basic Pointer Declaration and Dereferencing

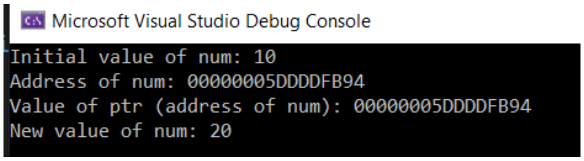
Task:

1. Declare an integer variable.
2. Declare a pointer to an integer.
3. Use the pointer to store the address of the integer.
4. Dereference the pointer to change the value of the integer.



```c
#include <stdio.h>

int main() {
    int num = 10;       // Integer variable
    int* ptr = &num;    // Pointer to integer, storing address of num

    printf("Initial value of num: %d\n", num);
    printf("Address of num: %p\n", (void*)&num);
    printf("Value of ptr (address of num): %p\n", (void*)ptr);

    // Dereference pointer to change value of num
    *ptr = 20;  // Changing value of num using pointer

    printf("New value of num: %d\n", num);

    return 0;
}
```

Expected output:

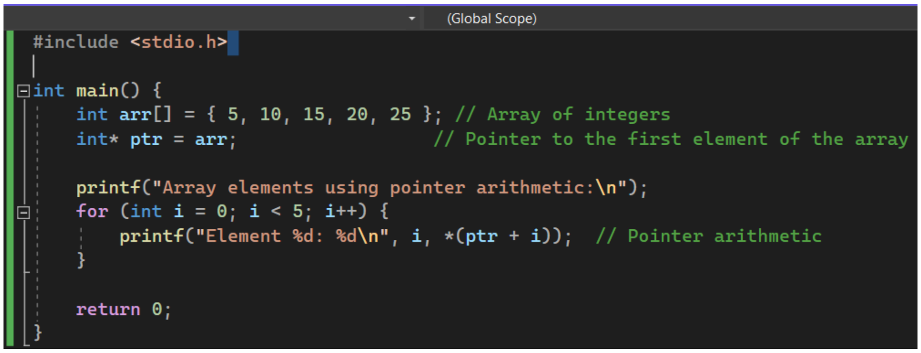


```
Initial value of num: 10
Address of num: 00000005DDDDFB94
Value of ptr (address of num): 00000005DDDDFB94
New value of num: 20
```

Exercise 2: Pointer Arithmetic

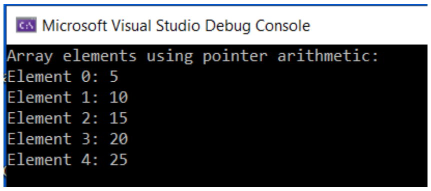
Task:

1. Declare an array of integers.
2. Declare a pointer that points to the first element of the array.
3. Use pointer arithmetic to access and print all elements of the array.



```c
#include <stdio.h>

int main() {
    int arr[] = { 5, 10, 15, 20, 25 }; // Array of integers
    int* ptr = arr;                   // Pointer to the first element of the array

    printf("Array elements using pointer arithmetic:\n");
    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i, *(ptr + i));  // Pointer arithmetic
    }

    return 0;
}
```

Expected output:

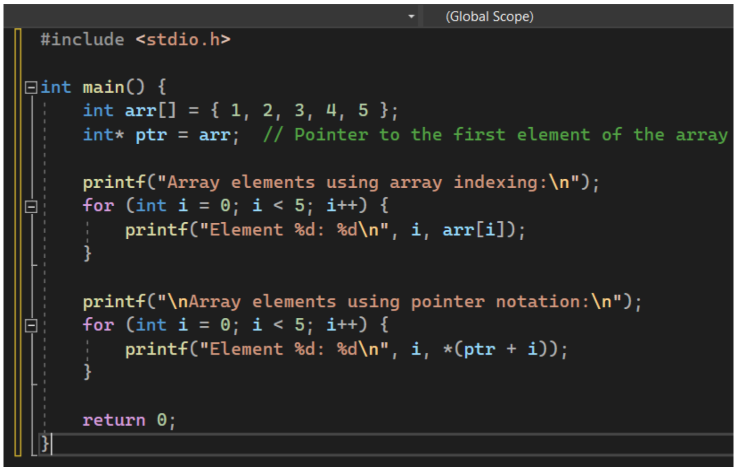


```
Array elements using pointer arithmetic:
Element 0: 5
Element 1: 10
Element 2: 15
Element 3: 20
Element 4: 25
```

Exercise 3: Pointers and Arrays

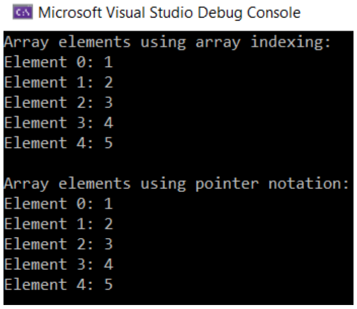
Task:

1. Create an array.
2. Use both array indexing and pointer notation to print the elements.



```
#include <stdio.h>

int main() {
    int arr[] = { 1, 2, 3, 4, 5 };
    int* ptr = arr;  // Pointer to the first element of the array

    printf("Array elements using array indexing:\n");
    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i, arr[i]);
    }

    printf("\nArray elements using pointer notation:\n");
    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i, *(ptr + i));
    }

    return 0;
}
```

Expected output:

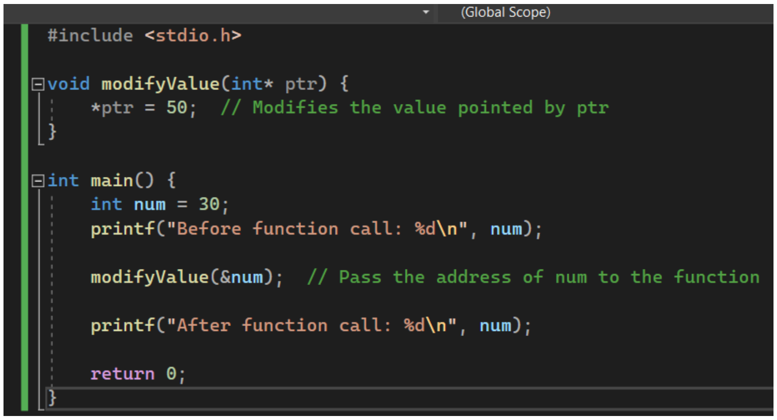


```
Array elements using array indexing:
Element 0: 1
Element 1: 2
Element 2: 3
Element 3: 4
Element 4: 5

Array elements using pointer notation:
Element 0: 1
Element 1: 2
Element 2: 3
Element 3: 4
Element 4: 5
```

Exercise 4: Pointers and Functions

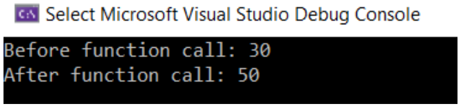
Task:

1. Write a function that takes a pointer to an integer and modifies the integer's value.
2. Call this function from the main() function.



```c
#include <stdio.h>

void modifyValue(int* ptr) {
    *ptr = 50;  // Modifies the value pointed by ptr
}

int main() {
    int num = 30;
    printf("Before function call: %d\n", num);

    modifyValue(&num);  // Pass the address of num to the function

    printf("After function call: %d\n", num);

    return 0;
}
```

Expected output:

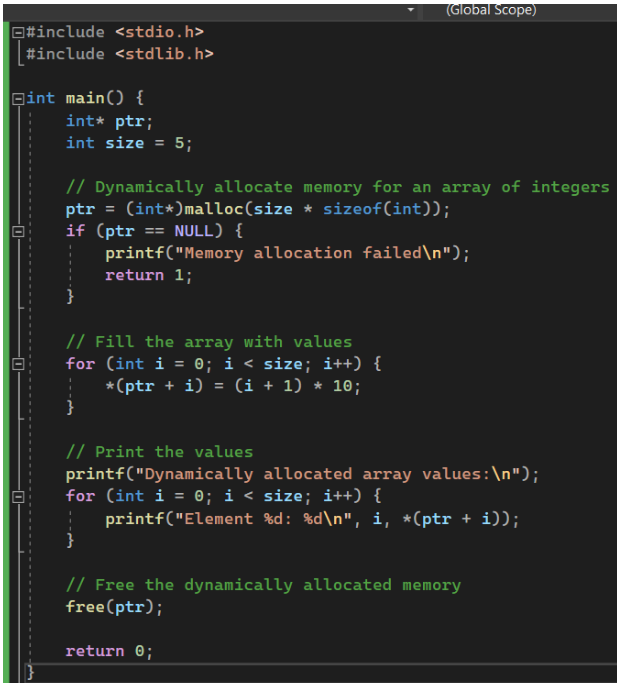


```
Before function call: 30
After function call: 50
```

Exercise 5: Dynamic Memory Allocation with Pointers

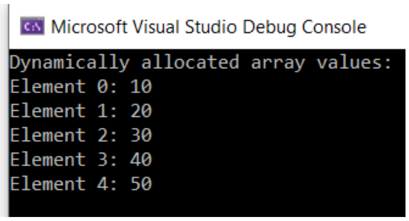
Task:

1. Use malloc() to dynamically allocate memory for an array.
2. Use a pointer to access the array and modify its values.
3. Free the dynamically allocated memory using free().



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

int main() {
    int* ptr;
    int size = 5;

    // Dynamically allocate memory for an array of integers
    ptr = (int*)malloc(size * sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Fill the array with values
    for (int i = 0; i < size; i++) {
        *(ptr + i) = (i + 1) * 10;
    }

    // Print the values
    printf("Dynamically allocated array values:\n");
    for (int i = 0; i < size; i++) {
        printf("Element %d: %d\n", i, *(ptr + i));
    }

    // Free the dynamically allocated memory
    free(ptr);

    return 0;
}
```

Expected output:

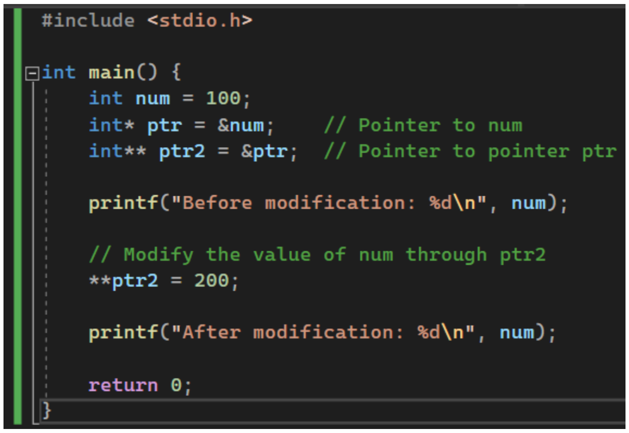


```
Dynamically allocated array values:
Element 0: 10
Element 1: 20
Element 2: 30
Element 3: 40
Element 4: 50
```

Exercise 6: Pointers to Pointers

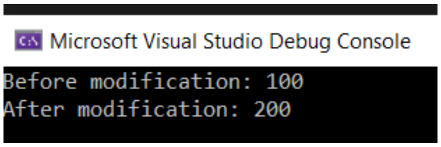
Task:

1. Declare a pointer to a pointer.
2. Use it to modify the value of an integer.



```
#include <stdio.h>

int main() {
    int num = 100;
    int* ptr = &num;     // Pointer to num
    int** ptr2 = &ptr;   // Pointer to pointer ptr

    printf("Before modification: %d\n", num);

    // Modify the value of num through ptr2
    **ptr2 = 200;

    printf("After modification: %d\n", num);

    return 0;
}
```

Expected output:



```
Before modification: 100
After modification: 200
```

Summary:

- Pointer Basics: Pointers hold the memory address of variables. You can dereference a pointer to access or modify the value at that address.
- Pointer Arithmetic: You can perform arithmetic on pointers to navigate through arrays.
- Pointers and Functions: Pointers can be passed to functions to modify values outside the function scope.
- Dynamic Memory Allocation: malloc() is used to allocate memory at runtime, and free() is used to release that memory.
- Pointers to Pointers: A pointer can point to another pointer, enabling multi-level dereferencing.