Unit - 8
Pointer
(marks: 5)

CTEVT Diploma in Computer Engineering
Subject : C Programming
Shree Sarwajanik Secondary Technical School
Prepared By: Er. Abinash Adhikari

Introduction to Pointer

A pointer is a variable that stores the memory address of another variable. Instead of holding a direct value, it holds the address where the value is stored in memory. Pointers are powerful in C because they allow for dynamic memory management, efficient array handling, and function passing by reference.

There are 2 important operators that we will use in pointers concepts i.e.

1. Dereferencing operator(*) used to declare pointer variable and access the value stored in the address.
2. Address operator(&) used to returns the address of a variable or to access the address of a variable to a pointer.

Key Features of Pointers:

- They store memory addresses.
- They allow direct memory access and manipulation.
- They support dynamic memory allocation.
- They improve program efficiency and flexibility.

Syntax:

```
int x = 10;   // An integer variable
int *p;       // A pointer to an integer
p = &x;       // p now holds the address of x
```

Here, ptr holds the memory address of a. The * operator is used to declare a pointer, and the & operator is used to get the address of a variable.

Address (&) and Indirection (*) Operators

1. **Address-of Operator (&):**
   a. The & operator is used to obtain the memory address of a variable. Every variable in C has a unique memory address where its value is stored.
   b. **Syntax:**

```
int a = 10;
printf("Value of a: %d\n", a);          // Output: 10
printf("Address of a: %p\n", &a);       // Output: Address of a (e.g.,
0x7ffee4b5c9ac)
```

2. **Indirection Operator (*):**
   a. The * operator is used to access the value stored at the memory address held by a pointer. It is also known as the dereference operator.
   b. **Syntax:**

```
int a = 10;
int *p = &a;  // p holds the address of a
printf("Value of a: %d\n", *p);  // Output: 10 (value at the address stored
in p)
```

Pointer Arithmetic Operations

Pointer arithmetic involves operations like addition, subtraction, increment, and decrement on pointers. The result depends on the type of data the pointer points to.

Types of Pointer Arithmetic:

1. Incrementing/Decrementing a Pointer:
   a. When you increment a pointer, it moves to the next memory location based on the size of the data type it points to.

   Example:

```
int arr[3] = {10, 20, 30};
int *p = arr;  // p points to the first element of arr
p++;           // p now points to arr[1]
```

2. Adding/Subtracting an Integer to/from a Pointer:
   a. Adding an integer n to a pointer moves it n elements forward.

   Example:

```
int arr[3] = {10, 20, 30};
int *p = arr;
p = p + 2;     // p now points to arr[2]
```

Subtracting Two Pointers:

1. Subtracting two pointers gives the number of elements between them.
2. Syntax:

```
int arr[5] = {10, 20, 30, 40, 50};
int *p1 = &arr[1];
int *p2 = &arr[4];
int diff = p2 - p1;  // diff = 3
```

Important Notes:

1. Pointer arithmetic is only valid within arrays.
2. You cannot perform multiplication or division on pointers.
3. The size of the data type affects how much the pointer moves:
    a. For int, the pointer moves by sizeof(int) bytes.
    b. For char, the pointer moves by sizeof(char) bytes (which is typically 1 byte).

Pointer to Pointer in C

A pointer to a pointer is a form of multiple indirection, where a pointer holds the address of another pointer. This concept is useful in scenarios where you need to pass a pointer to a function and modify the pointer itself.

Syntax:

```
int x = 10;
int *p = &x;      // p is a pointer to x
int **pp = &p;    // pp is a pointer to p
```

Accessing Values:

```
printf("%d\n", **pp);  // Outputs 10, the value of x
```

Use Cases:

1. Useful in scenarios where you need to pass a pointer to a function and modify the pointer itself.
2. Commonly used in multi-dimensional arrays and complex data structures.

Dynamic Memory Allocation (malloc(), calloc(), realloc(), free())

Dynamic memory allocation allows you to allocate memory during runtime rather than at compile time. This is particularly useful when the amount of memory needed isn't known beforehand.

Best Practices:

- Always check if malloc() or calloc() returns NULL to ensure memory was successfully allocated.
- Always free dynamically allocated memory to avoid memory leaks.
- Avoid using memory after it has been freed.

1. **malloc() (Memory Allocation):**

Allocates a block of memory of a specified size and returns a pointer to the beginning of the block.

```
int *p = (int *) malloc(5 * sizeof(int));  // Allocates memory for 5 integers
if (p == NULL) {
    printf("Memory allocation failed\n");
}
```

2. **calloc() (Contiguous Allocation):**

Similar to malloc(), but initializes all bytes in the allocated storage to zero.

```
int *p = (int *) calloc(5, sizeof(int));  // Allocates memory for 5 integers and initializes to 0
```

3. **realloc() (Reallocate Memory):**

Changes the size of the previously allocated memory block.

```
int *p = (int *) malloc(5 * sizeof(int));
p = (int *) realloc(p, 10 * sizeof(int));  // Resizes the memory to hold 10 integers
```

4. **free() (Free Memory):**

Deallocates the memory previously allocated by malloc(), calloc(), or realloc().

```
free(p);  // Frees the memory pointed to by p
```

Summary

- Pointers store memory addresses and allow efficient manipulation of data.
- The & operator retrieves the address of a variable, while the * operator accesses the value at a given address.
- Pointer arithmetic allows you to navigate through arrays and other contiguous memory blocks.
- Pointer to pointer adds another layer of indirection, useful in certain advanced scenarios.
- Dynamic memory allocation (malloc(), calloc(), realloc(), free()) provides flexibility in managing memory at runtime, crucial for handling unknown or varying amounts of data.