Class Note: Pointers (Section 10)

Overview

This section introduces **pointers** in C, a powerful feature that allows direct memory manipulation. Pointers store memory addresses of variables, enabling efficient data handling, dynamic memory allocation, and passing data by reference. Students will learn to declare, use, and manipulate pointers, integrating them with arrays, functions, and structures.

Learning Objectives

- Understand the concept of pointers and their role in memory management.
- Learn to declare, initialize, and dereference pointers.
- Explore pointer arithmetic and its applications with arrays.
- Use pointers with functions and structures for efficient programming.
- Apply pointers to solve real-world problems.

Key Concepts

1. Pointer Basics

- **Definition**: A pointer is a variable that stores the memory address of another variable.
- **Declaration**: Use the `*` operator (e.g., `int *ptr;` declares a pointer to an integer).
- **Initialization**: Assign the address of a variable using `&` (e.g., `ptr = &num;`).
- **Dereferencing**: Access the value at the pointer's address using `*` (e.g., `*ptr`).

2. Pointer Arithmetic

- Pointers can be incremented or decremented to point to the next/previous memory location of the same type.
- Example: For an `int` pointer, `ptr + 1` moves to the next integer (typically 4 bytes ahead, depending on the system).

3. Pointers and Arrays

- Arrays and pointers are closely related; the array name is a pointer to its first element (e.g., `arr == &arr[0]`).
- Access elements using pointer arithmetic (e.g., `*(arr + i)` is equivalent to `arr[i]`).

4. Pointers with Functions

- **Pass by Reference**: Pass pointers to functions to modify original variables.
- Example: `void swap(int *a, int *b)`.

5. Pointers with Structures

- Use pointers to structures for efficient data handling.
- Access members using the arrow operator (`->`) (e.g., `ptr->member`).

Examples

Example 1: Basic Pointer Usage

```c
#include <stdio.h>
int main() {
    int num = 10;
    int *ptr = &num; // Pointer stores address of num
    printf("Address of num: %p\n", ptr);
    printf("Value at ptr: %d\n", *ptr); // Dereferencing
    *ptr = 20; // Modify value via pointer
    printf("New value of num: %d\n", num);
    return 0;
}
```

Output:

```
Address of num: (some address)
Value at ptr: 10
New value of num: 20
```

Explanation: Demonstrates pointer declaration, initialization, and dereferencing.

Example 2: Pointer Arithmetic with Arrays

```c
#include <stdio.h>
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int *ptr = arr; // Points to the first element
    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i, *(ptr + i));
    }
    return 0;
}
```

Output:

```
Element 0: 10
Element 1: 20
Element 2: 30
Element 3: 40
Element 4: 50
```

Explanation: Uses pointer arithmetic to access array elements.

Example 3: Pointers in Functions (Swap)

```c
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main() {
    int x = 5, y = 10;
    printf("Before swap: x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("After swap: x = %d, y = %d\n", x, y);
    return 0;
}
```

Output:

```
Before swap: x = 5, y = 10
After swap: x = 10, y = 5
```

Explanation: Passes addresses to modify original variables.

Example 4: Pointers with Structures

```c
#include <stdio.h>
struct Student {
    int rollNo;
    char name[20];
};
int main() {
    struct Student s1 = {101, "Alice"};
    struct Student *ptr = &s1;
    printf("Roll No: %d, Name: %s\n", ptr->rollNo, ptr->name);
    return 0;
}
```

Output:

```
Roll No: 101, Name: Alice
```

Explanation: Uses the arrow operator to access structure members via a pointer.

Classwork

Task 1: Pointer to Modify Value

Write a program to declare an integer variable and a pointer to it. Use the pointer to change the variable's value from 50 to 100, then print the new value.

Task 2: Array Sum Using Pointers

Write a program to calculate the sum of an array of 5 integers using a pointer instead of array indexing. Accept the values from the user.

Task 3: Function with Pointer Parameter

Write a function `doubleValue` that takes a pointer to an integer and doubles its value. Call the function from `main` and display the result.

Task 4: Structure with Pointer

Define a structure `Rectangle` with members `length` and `width`. Write a function `increaseDimensions` that takes a pointer to a `Rectangle` and increases its dimensions by 2 units each. Test in `main`.

Real-World Applications

Application 1: Memory Management

Context: Pointers are used to dynamically allocate memory (e.g., for a variable-size array).

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *arr;
    int size = 3;
    arr = (int *)malloc(size * sizeof(int)); // Dynamic allocation
    for (int i = 0; i < size; i++) {
        arr[i] = (i + 1) * 10;
    }
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    free(arr); // Release memory
    return 0;
}
```

Output: 10 20 30

Discussion: Pointers enable dynamic memory allocation.

Application 2: Data Sorting

Context: Use pointers to sort an array efficiently.

```c
#include <stdio.h>
void sort(int *arr, int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (*(arr + j) > *(arr + j + 1)) {
                int temp = *(arr + j);
                *(arr + j) = *(arr + j + 1);
                *(arr + j + 1) = temp;
            }
        }
    }
}
int main() {
    int arr[4] = {40, 10, 30, 20};
    sort(arr, 4);
    for (int i = 0; i < 4; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output: 10 20 30 40

Discussion: Pointers allow efficient array manipulation.

Objective Questions

1. **What does the following code print?**

```c
int x = 10;
int *ptr = &x;
printf("%d", *ptr);
```

a) Address of x

b) 10

c) Garbage value

d) Error

Answer: b) 10

2. **Which operator is used to access structure members via a pointer?**

a) `.`

b) `&`

c) `->`

d) `*`

Answer: c) `->`

3. **What is the output of the following code?**

```c
int arr[3] = {1, 2, 3};
int *ptr = arr;
ptr++;
printf("%d", *ptr);
```

a) 1

b) 2

c) 3

d) Address

Answer: b) 2

4. **What happens if you dereference an uninitialized pointer?**

a) It prints 0

b) It causes undefined behavior

c) It prints the address

d) It compiles but doesn't run

Answer: b) It causes undefined behavior

5. **In the function declaration `void func(int *p)`, what does `*p` indicate?**

a) p is an integer

b) p is a pointer to an integer

c) p is an array

d) p is a structure

Answer: b) p is a pointer to an integer

Additional Notes

- **Practice**: Test with edge cases (e.g., null pointers, invalid memory access).
- **Tools**: Use NetBeans to debug pointer-related errors.
- **Extension**: Explore dynamic memory allocation (`malloc` , `free`) in more depth.