Class Notes: Arrays (Section 7)

Overview

This section introduces arrays in C, a data structure that allows storing multiple elements of the same data type in contiguous memory locations. Students will learn how to declare, initialize, access, and manipulate arrays, integrating them with loops and decision-making constructs.

Learning Objectives

- Understand the concept and purpose of arrays in C.
- Learn to declare, initialize, and access array elements.
- Apply loops and conditions to process array data.
- Use arrays to solve real-world problems.

Key Concepts

1. Array Basics

- **Definition**: A collection of elements of the same data type stored in contiguous memory.
- **Syntax**:
  - Declaration: `data_type array_name[array_size];`
  - Example: `int numbers[5];`
- **Indexing**: Elements are accessed using indices starting from 0 (e.g., `numbers[0]`).
- **Size**: Must be a constant expression (e.g., defined with `#define` or `const`).

2. Initialization

- **At Declaration**: `int numbers[5] = {1, 2, 3, 4, 5};`
- **Partial Initialization**: Unspecified elements are set to 0 (e.g., `int numbers[5] = {1, 2};` sets the rest to 0).
- **Omitted Size**: Size can be inferred (e.g., `int numbers[] = {1, 2, 3};` creates an array of size 3).

3. Accessing and Modifying

- **Access**: Use index (e.g., `printf("%d", numbers[2]);` prints 3).
- **Modification**: Assign new values (e.g., `numbers[1] = 10;`).
- **Bounds**: Accessing beyond the array size (e.g., `numbers[5]`) leads to undefined behavior.

4. Processing Arrays

- **Loops**: Use `for` loops to iterate over elements.
- **Decision-Making**: Use `if` to filter or modify elements.

Examples

Example 1: Array Declaration and Initialization

```c
#include <stdio.h>
int main() {
    int marks[4] = {85, 90, 78, 95};
    printf("Marks: %d, %d, %d, %d\n", marks[0], marks[1], marks[2], marks[3]);
    return 0;
}
```

Output: Marks: 85, 90, 78, 95
Explanation: Declares and prints an initialized array.

Example 2: Array with Loop

```c
#include <stdio.h>
int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum += numbers[i];
    }
    printf("Sum: %d\n", sum); // Output: 15
    return 0;
}
```

Explanation: Uses a `for` loop to calculate the sum of array elements.

Example 3: Array with Decision

```c
#include <stdio.h>
int main() {
    int scores[5] = {50, 75, 30, 85, 60};
    int pass_count = 0;
    for (int i = 0; i < 5; i++) {
        if (scores[i] >= 50) {
            pass_count++;
        }
    }
    printf("Number of passes: %d\n", pass_count); // Output: 3
    return 0;
}
```

Explanation: Counts elements above a threshold using an `if` condition.

Example 4: Multi-Dimensional Array

```c
#include <stdio.h>
int main() {
    int matrix[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
1 2 3
4 5 6
```

Explanation: Demonstrates a 2D array with nested loops.

Classwork

Task 1: Array Input and Output

Write a program to accept 5 integers from the user into an array and print them in reverse order.

Task 2: Array with Condition

Write a program to find the highest score in an array of 6 student scores input by the user, using a `for` loop and `if` statement.

Task 3: 2D Array Sum

Write a program to create a 3x3 matrix, accept values from the user, and calculate the sum of all elements using nested `for` loops.

Task 4: Search in Array

Write a program to search for a specific number in an array of 10 elements (input by the user) and print its index if found, or "Not found" if not.

Real-World Applications

Application 1: Student Grade Tracker

Context: Store and analyze student marks.

```c
#include <stdio.h>
int main() {
    int marks[5] = {85, 60, 45, 90, 70};
    int above_avg = 0;
    for (int i = 0; i < 5; i++) {
        if (marks[i] >= 70) {
            above_avg++;
        }
    }
    printf("Students above 70: %d\n", above_avg); // Output: 2
    return 0;
}
```

Discussion: Arrays manage multiple data points efficiently.

Application 2: Matrix Rotation

Context: Rotate a 2x2 matrix (e.g., for image processing).

```c
#include <stdio.h>
int main() {
    int matrix[2][2] = {{1, 2}, {3, 4}};
    printf("Original:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
    // Rotate logic (simplified)
    int temp = matrix[0][0];
    matrix[0][0] = matrix[1][0];
    matrix[1][0] = matrix[1][1];
    matrix[1][1] = matrix[0][1];
    matrix[0][1] = temp;
    printf("Rotated:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
Original:
1 2
3 4
Rotated:
3 1
4 2
```

Discussion: 2D arrays model matrix operations.

Discussion Points

Question 1: Array Bounds

What happens if you access `array[5]` in an array of size 5? How can you prevent it?
Answer: Undefined behavior (crash or garbage value). Use size checks or loop limits.

Question 2: 1D vs. 2D Arrays

When would you use a 2D array instead of a 1D array?
Answer: For grid-based data (e.g., matrices, game boards).

Question 3: Optimization

How can you improve the efficiency of searching an array?
Answer: Use sorted arrays with binary search or early `break` in linear search.

Question 4: Design Challenge

Write a program to transpose a 2x3 matrix (swap rows and columns) using a 2D array.
Solution: Involves creating a new 3x2 array and swapping indices.

Additional Notes

- **Practice**: Test with edge cases (e.g., empty arrays, maximum indices).
- **Tools**: Use NetBeans to debug array access errors.
- **Extension**: Introduce dynamic arrays (via pointers) in the next section.

These notes provide a solid foundation for teaching arrays in C, blending theory with practical examples. Let me know if you'd like exercises or further details for Section 7!