Class Notes: Functions, Structures, and Unions (Sections 8 and 9)

Overview

This combined session explores two critical aspects of C programming: **functions** for modular code design and **structures/unions** for handling compound data. Functions allow breaking programs into reusable blocks, while structures and unions enable grouping related data into custom data types. Students will learn to define, call, and integrate functions with structures and unions to solve complex problems.

Learning Objectives

- Understand functions: declaration, definition, parameters, and return values.
- Learn to create and use structures and unions in C.
- Integrate functions with structures/unions for modular and efficient programming.
- Apply these concepts to real-world scenarios.

Part 1: Functions (Section 8)

Key Concepts

1. **Function Basics**:
   - **Definition**: A block of code performing a specific task, reusable across the program.
   - **Syntax**:

```
return_type function_name(parameter_list) {
    // Code
    return value; // Optional if return_type is void
}
```

   - **Declaration**: Needed if the function is defined after its call (e.g., `int add(int, int);`).
   - **Call**: Invokes the function (e.g., `sum = add(3, 4);`).
2. **Types of Functions**:
   - No parameters, no return: `void printMessage()`.
   - With parameters, no return: `void printSum(int a, int b)`.
   - With parameters and return: `int multiply(int a, int b)`.
3. **Parameter Passing**:

- **Pass by Value**: Copies of arguments are passed; original values remain unchanged.
- Arrays are passed as pointers (covered later with pointers in Section 10).

4. **Function Scope**:
   - Variables inside a function are local and inaccessible outside unless returned.

Examples for Functions

Example 1: Function to Calculate Power

```c
#include <stdio.h>
int power(int base, int exp) {
    int result = 1;
    for (int i = 0; i < exp; i++) {
        result *= base;
    }
    return result;
}
int main() {
    int result = power(2, 3);
    printf("2^3 = %d\n", result); // Output: 2^3 = 8
    return 0;
}
```

Explanation: The `power` function calculates `base` raised to `exp` using a loop.

Example 2: Function with Array Parameter

```c
#include <stdio.h>
float average(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return (float)sum / size;
}
int main() {
    int marks[5] = {70, 80, 60, 90, 85};
    printf("Average: %.2f\n", average(marks, 5)); // Output: Average: 77.00
    return 0;
}
```

Explanation: Calculates the average of array elements using a function.

Part 2: Structures and Unions (Section 9)

Key Concepts

1. **Structures**:
   - **Definition**: A user-defined data type that groups related variables of different types.
   - **Syntax**:

     ```c
     struct structure_name {
         data_type member1;
         data_type member2;
         // ...
     };
     ```

   - **Declaration**: `struct structure_name variable;`
   - **Access**: Use the dot operator (`.`) to access members (e.g., `variable.member1`).
2. **Unions**:
   - **Definition**: Similar to structures, but all members share the same memory location.
   - **Syntax**:

```
union union_name {
    data_type member1;
    data_type member2;
    // ...
};
```

- **Key Difference**: Only one member holds a value at a time; size of the union is the size of the largest member.

3. **Typedef**:
   - Simplifies structure/union declarations: `typedef struct { ... } Alias;`
4. **Arrays of Structures**:
   - Store multiple structure instances (e.g., `struct Student students[10];`).

Examples for Structures and Unions

Example 3: Structure for Student Data

```
#include <stdio.h>
struct Student {
    int rollNo;
    char name[20];
    float marks;
};
int main() {
    struct Student s1 = {101, "Alice", 85.5};
    printf("Roll No: %d, Name: %s, Marks: %.1f\n", s1.rollNo, s1.name, s1.marks);
    return 0;
}
```

Output: Roll No: 101, Name: Alice, Marks: 85.5
Explanation: Groups related data (roll number, name, marks) into a single unit.

Example 4: Union Example

```c
#include <stdio.h>
union Data {
    int i;
    float f;
    char c;
};
int main() {
    union Data d;
    d.i = 65;
    printf("Integer: %d\n", d.i); // Output: 65
    printf("Char (same memory): %c\n", d.c); // Output: A (ASCII 65)
    d.f = 3.14;
    printf("Float (overwrites): %.2f\n", d.f); // Output: 3.14
    return 0;
}
```

Explanation: Shows how a union uses the same memory for different types.

Example 5: Combining Functions and Structures

```c
#include <stdio.h>
struct Point {
    int x;
    int y;
};
int distance(struct Point p1, struct Point p2) {
    return (p2.x - p1.x) + (p2.y - p1.y); // Simplified Manhattan distance
}
int main() {
    struct Point p1 = {1, 2};
    struct Point p2 = {4, 5};
    printf("Distance: %d\n", distance(p1, p2)); // Output: 6
    return 0;
}
```

Explanation: A function calculates the Manhattan distance between two points stored as structures.

Classwork

Task 1: Function to Calculate Area

Write a function `circleArea` that takes a radius (float) and returns the area of a circle (use 3.14 for π). Call it from `main`.

Task 2: Structure for Employee

Define a structure `Employee` with members `id`, `name`, and `salary`. Create an array of 3 employees, input their data, and write a function `printEmployee` to display each employee's details.

Task 3: Union for Data Storage

Create a union `Value` with members `intVal`, `floatVal`, and `charVal`. Write a program to store and display values, demonstrating how memory is shared.

Task 4: Combined Task

Define a structure `Rectangle` with members `length` and `width`. Write a function `calculatePerimeter` that takes a `Rectangle` as a parameter and returns its perimeter. Test with user input.

Real-World Applications

Application 1: Library Management (Structures + Functions)

Context: Manage book records using structures and functions.

```c
#include <stdio.h>
struct Book {
    int id;
    char title[50];
    float price;
};
void printBook(struct Book b) {
    printf("ID: %d, Title: %s, Price: %.2f\n", b.id, b.title, b.price);
}
int main() {
    struct Book b1 = {101, "C Programming", 29.99};
    printBook(b1);
    return 0;
}
```

Output: ID: 101, Title: C Programming, Price: 29.99

Discussion: Structures store book data; functions handle operations like printing.

Application 2: Configuration Storage (Unions)

Context: Store configuration data where only one type is needed at a time.

```c
#include <stdio.h>
union Config {
    int maxUsers;
    float timeout;
};
int main() {
    union Config cfg;
    cfg.maxUsers = 100;
    printf("Max Users: %d\n", cfg.maxUsers);
    cfg.timeout = 5.5;
    printf("Timeout: %.1f\n", cfg.timeout);
    return 0;
}
```

Discussion: Unions save memory by reusing the same space.

Application 3: Geometry Calculator (Combined)

Context: Calculate area of shapes using structures and functions.

```c
#include <stdio.h>
struct Triangle {
    float base;
    float height;
};
float triangleArea(struct Triangle t) {
    return 0.5 * t.base * t.height;
}
int main() {
    struct Triangle t1 = {5.0, 3.0};
    printf("Triangle Area: %.2f\n", triangleArea(t1)); // Output: 7.50
    return 0;
}
```

Discussion: Combines structures for data and functions for computation.

Discussion Points

Question 1: Function Modularity

How do functions improve code readability and maintenance?
Answer: They break code into smaller, reusable units, making debugging and updates easier.

Question 2: Structures vs. Unions

When would you use a union instead of a structure?
Answer: Use a union when only one member needs to be active at a time (e.g., variant data types).

Question 3: Memory Efficiency

How does a union save memory compared to a structure?
Answer: A union allocates memory for the largest member only; a structure allocates for all members.

Question 4: Design Challenge

Write a program using a structure `Student` and a function to find the student with the highest marks from an array of students.
Solution: Involves array of structures and a function with comparison logic.