Complete C Programming Summary

Summary Created By Abdelhamid Khald

June 2025

Table of Contents

1. Problem Solving with Computers

Problem-solving is the process of identifying a problem, analyzing its causes, and finding effective solutions to address it. This skill is essential in various fields, from technology and business to everyday life situations.

1.1 Key Steps in Problem-Solving

- 1. Identify the Problem: Clearly define the issue to understand its scope and impact.
- 2. Analyze the Problem: Break down the problem to identify root causes and contributing factors.
- 3. Generate Possible Solutions: Brainstorm multiple approaches to address the issue.
- 4. Evaluate and Select Solutions: Compare potential solutions based on feasibility, effectiveness, and resources.
- 5. Implement the Solution: Put the chosen solution into action with a clear plan.
- 6. Evaluate the Outcome: Assess the effectiveness of the solution and make adjustments if necessary.

1.2 Problem-Solving Techniques

- Root Cause Analysis: Finding the primary cause of a problem
- Brainstorming: Generating creative solutions in a group
- 5 Whys Technique: Asking 'why' five times to uncover the root cause
- SWOT Analysis: Evaluating Strengths, Weaknesses, Opportunities, and Threats
- Trial and Error: Testing different solutions until one works

Practice Questions:

1. What are the six key steps in problem-solving?

2. Explain the difference between Root Cause Analysis and the 5 Whys Technique.

3. Give an example of when you would use SWOT Analysis in programming.

4. How can brainstorming help in software development?

2. Flowcharts and Pseudocode

A flowchart is a diagram that represents a process or workflow using various symbols to denote operations, decisions, inputs, and outputs, connected by arrows indicating the process flow.

2.1 Common Flowchart Symbols

Symbol	Meaning
Rectangle	Process step
Diamond	Decision point
Oval	Start/End point
Parallelogram	Input/Output
Arrow	Flow direction

2.2 Algorithm Examples

Algorithm to find the largest of three numbers:

```
1. Start
2. Input three numbers: A, B, C
3. If A > B and A > C, then
   - Print A as the largest
4. Else if B > C, then
   - Print B as the largest
5. Otherwise,
   - Print C as the largest
6. End
```

Code Comments: This algorithm uses nested conditional statements to compare three numbers systematically.

Explanation: The algorithm follows a decision tree approach where we first check if A is largest, then B, and finally default to C.

Practice Questions:

1. What symbol would you use to represent a decision in a flowchart?

2. Write an algorithm to find the sum of first 10 natural numbers.

3. Draw a flowchart for checking if a number is even or odd.

4. What's the difference between an algorithm and a flowchart?

3. Introduction to C Programming

3.1 Why Learn C?

C is a high-level, general-purpose programming language developed by Dennis Ritchie in 1972 at Bell Labs. It serves as the foundation for many modern programming languages.

- Foundation of Modern Languages: C is the basis for many languages like C++, Java, and Python
- Performance: Programs written in C are fast and efficient due to low-level memory manipulation
- Portability: C code can be compiled and executed on various computer platforms with minimal modification
- System Programming: Widely used in developing operating systems, embedded systems, and hardware drivers

3.2 First C Program - Hello World

```
#include <stdio.h>    // Header file for standard input/output functions

int main() {          // Main function - program execution starts here
    /*
     * This is a multi-line comment
     * My first program in C programming language
     * Purpose: Display "Hello World!" on the screen
     */

    printf("Hello World!\n");  // Print text with newline character

    return 0;         // Return 0 to indicate successful program execution
}                     // End of main function
```

Code Comments: Each line has a specific purpose: #include adds library functions, main() is the entry point, printf() displays output, and return 0 indicates success.

Explanation: This is the standard structure of every C program. The preprocessor directive #include tells the compiler to include the stdio.h header file which contains input/output function declarations.

Program Components Explained:

- #include <stdio.h> - Preprocessor directive that includes standard I/O library
- int main() - Main function where program execution begins, returns an integer
- printf() - Function to output formatted text to the screen
- return 0 - Indicates successful program termination to the operating system
- /* */ - Multi-line comment syntax for documentation
- // - Single-line comment syntax for brief explanations

Practice Questions:

1. What is the purpose of #include <stdio.h> in a C program?

2. Why do we use return 0 at the end of the main function?
3. What's the difference between /* */ and // comment styles?
4. Modify the Hello World program to print your name instead.
5. What happens if you forget the semicolon after printf()?

4. Data Types in C

4.1 Basic Data Types

Data Type	Size	Range	Example
char	1 byte	-128 to 127	'A', 'x'
int	4 bytes	-2,147,483,648 to 2,147,483,647	42, -15
float	4 bytes	6-7 decimal places	3.14f, 2.5f
double	8 bytes	15-16 decimal places	3.14159265, 2.718281828

4.2 Variable Declaration Examples

```
#include <stdio.h>

int main() {
    // Character variable declaration and initialization
    char grade = 'A';            // Single character stored in 1 byte
    char initial = 'J';          // Another character variable

    // Integer variable declarations
    int age = 25;                // Whole number (positive or negative)
    int temperature = -5;        // Negative integer
    int count = 0;               // Initialize to zero

    // Floating-point variable declarations
    float price = 99.99f;        // Single precision decimal (6-7 digits)
    float height = 5.8f;         // 'f' suffix indicates float literal

    // Double precision declarations
    double pi = 3.14159265359;   // Higher precision decimal (15-16 digits)
    double distance = 384400.0; // Distance to moon in kilometers

    // Display all variables with their values
    printf("Grade: %c\n", grade);               // %c for character
    printf("Age: %d\n", age);                   // %d for integer
    printf("Price: %.2f\n", price);             // %.2f for 2 decimal places
    printf("Pi value: %.10lf\n", pi);           // %lf for double

    return 0;
}
```

Code Comments: Each data type serves a specific purpose: char for single characters, int for whole numbers, float for decimals with moderate precision, and double for high-precision decimals.

Explanation: Format specifiers in printf() must match the variable type: %c for char, %d for int, %f for float, %lf for double. The .2f means 2 decimal places.

Practice Questions:

1. What's the difference between float and double data types?
2. Why do we use 'f' suffix with float literals like 99.99f?
3. What format specifier would you use to print a character variable?
4. Declare variables to store a student's name initial, age, and GPA.
5. What happens if you try to store 300 in a char variable?

5. Operators in C

Operators are symbols that tell the compiler to perform specific mathematical or logical operations. C language is rich in built-in operators.

5.1 Arithmetic Operators

```c
#include <stdio.h>

int main() {
    // Declare and initialize two integer variables for demonstration
    int a = 10, b = 3;
    int result;                     // Variable to store calculation results

    printf("Arithmetic Operations Demonstration\n");
    printf("a = %d, b = %d\n\n", a, b);

    // Addition operator (+)
    result = a + b;               // Add two numbers
    printf("Addition: %d + %d = %d\n", a, b, result);

    // Subtraction operator (-)
    result = a - b;               // Subtract second from first
    printf("Subtraction: %d - %d = %d\n", a, b, result);

    // Multiplication operator (*)
    result = a * b;               // Multiply two numbers
    printf("Multiplication: %d * %d = %d\n", a, b, result);

    // Division operator (/) - Integer division
    result = a / b;               // Divide first by second (integer result)
    printf("Division: %d / %d = %d\n", a, b, result);

    // Modulus operator (%) - Remainder after division
    result = a % b;               // Get remainder of a divided by b
    printf("Modulus: %d %% %d = %d\n", a, b, result);

    // Increment and Decrement operators
    printf("\nIncrement/Decrement Operations:\n");

    // Pre-increment: increment first, then use value
    printf("Pre-increment: ++a = %d\n", ++a);    // a becomes 11

    // Post-increment: use value first, then increment
    printf("Post-increment: a++ = %d\n", a++);   // prints 11, a becomes 12
    printf("Value of a after post-increment: %d\n", a);

    // Pre-decrement: decrement first, then use value
    printf("Pre-decrement: --a = %d\n", --a);    // a becomes 11
    // Post-decrement: use value first, then decrement
    printf("Post-decrement: a-- = %d\n", a--);   // prints 11, a becomes 10
    printf("Final value of a: %d\n", a);

    return 0;
}
```

Code Comments: Key points: Integer division truncates decimals (10/3 = 3), modulus gives remainder (10%3 = 1), pre-increment changes value before using it, post-increment uses current value then changes it.

Explanation: The increment/decrement operators are crucial: ++a increments a then returns new value, a++ returns current value of a then increments it. This difference matters in expressions.

5.2 Relational Operators

```
#include <stdio.h>

int main() {
    // Variables for comparison
    int x = 10, y = 20, z = 10;

    printf("Relational Operators Demonstration\n");
    printf("x = %d, y = %d, z = %d\n\n", x, y, z);

    // Equal to operator (==) - checks if values are same
    printf("x == z: %d\n", x == z);    // 1 (true) because 10 == 10
    printf("x == y: %d\n", x == y);    // 0 (false) because 10 != 20

    // Not equal to operator (!=) - checks if values are different
    printf("x != y: %d\n", x != y);    // 1 (true) because 10 != 20
    printf("x != z: %d\n", x != z);    // 0 (false) because 10 == 10

    // Greater than operator (>)
    printf("y > x: %d\n", y > x);      // 1 (true) because 20 > 10
    printf("x > y: %d\n", x > y);      // 0 (false) because 10 < 20

    // Less than operator (<)
    printf("x < y: %d\n", x < y);      // 1 (true) because 10 < 20
    printf("y < x: %d\n", y < x);      // 0 (false) because 20 > 10

    // Greater than or equal to operator (>=)
    printf("x >= z: %d\n", x >= z);    // 1 (true) because 10 >= 10
    printf("y >= x: %d\n", y >= x);    // 1 (true) because 20 >= 10

    // Less than or equal to operator (<=)
    printf("x <= z: %d\n", x <= z);    // 1 (true) because 10 <= 10
    printf("x <= y: %d\n", x <= y);    // 1 (true) because 10 <= 20

    return 0;
}
```

Code Comments: Relational operators return 1 for true and 0 for false. Common mistake: using = (assignment) instead of == (comparison) in conditions.

Explanation: These operators are essential for decision making in programs. They compare values and return boolean results represented as integers in C.

5.3 Logical Operators

```c
#include <stdio.h>

int main() {
    // Variables for logical operations
    int a = 5, b = 10, c = 15;

    printf("Logical Operators Demonstration\n");
    printf("a = %d, b = %d, c = %d\n\n", a, b, c);

    // Logical AND (&&) - both conditions must be true
    printf("Logical AND Operations:\n");
    if (a < b && b < c) {         // Both (5 < 10) AND (10 < 15) are true
        printf("Both conditions are true: a < b && b < c\n");
    }

    if (a > b && b < c) {         // (5 > 10) is false, so result is false
        printf("This won't print\n");
    } else {
        printf("First condition false: a > b && b < c is false\n");
    }

    // Logical OR (||) - at least one condition must be true
    printf("\nLogical OR Operations:\n");
    if (a > 20 || b < 15) {       // (5 > 20) is false, but (10 < 15) is true
        printf("At least one condition is true: a > 20 || b < 15\n");
    }

    if (a > 20 || b > 30) {       // Both conditions are false
        printf("This won't print\n");
    } else {
        printf("Both conditions false: a > 20 || b > 30 is false\n");
    }

    // Logical NOT (!) - reverses the truth value
    printf("\nLogical NOT Operations:\n");
    if (!(a > b)) {               // a > b is false, !(false) = true
        printf("a is NOT greater than b\n");
    }

    if (!a) {                     // !5 = !true = false (any non-zero is true)
        printf("This won't print\n");
    } else {
        printf("a is non-zero, so !a is false\n");
    }

    // Complex logical expressions
    printf("\nComplex Logical Expression:\n");
    if ((a < b && b < c) || !(a > c)) {
        printf("Complex condition is true\n");
    }

    return 0;
}
```

Code Comments: && requires BOTH conditions true, || requires AT LEAST ONE true, ! reverses the truth value. In C, 0 is false, any non-zero value is true.

Explanation: Logical operators use short-circuit evaluation: in &&, if first condition is false, second isn't checked; in ||, if first condition is true, second isn't checked.

Practice Questions:

1. What's the difference between = and == operators?

2. What will be the result of 17 % 5 and why?

3. Explain the difference between ++i and i++ with an example.

4. What is short-circuit evaluation in logical operators?

5. Write a program to check if a number is between 10 and 50 using logical operators.

6. What will happen if you use = instead of == in an if condition?

6. Control Statements

Control statements allow you to control the flow of program execution based on certain conditions or to repeat a block of code multiple times.

6.1 if Statement

```c
#include <stdio.h>

int main() {
    int age;  // Variable to store user input

    // Prompt the user for their age
    printf("Enter your age: ");
    scanf("%d", &age);  // Read integer input (&age is the memory address)

    // Simple if statement to check age
    if (age >= 18) {    // Condition: is age greater than or equal to 18?
        // This block executes only if condition is true
        printf("You are an adult.\n");
        printf("You are eligible to vote.\n");
    }

    // This line always executes regardless of the condition
    printf("Age verification completed.\n");

    return 0;
}
```

Code Comments: The if statement executes a block of code only when the specified condition evaluates to true (non-zero in C). Notice how the curly braces {} create a block of multiple statements.

Explanation: The scanf() function reads user input and stores it at the memory address of the age variable (using the & operator). The code inside if block only executes if age >= 18.

6.2 if-else Statement

```c
#include <stdio.h>

int main() {
    int number;  // Variable to store input number

    // Prompt the user for a number
    printf("Enter an integer: ");
    scanf("%d", &number);

    // if-else statement to check if number is even or odd
    if (number % 2 == 0) {  // Condition: Is remainder 0 when divided by 2?
        // This block executes if condition is true (even number)
        printf("%d is an even number.\n", number);
    } else {
        // This block executes if condition is false (odd number)
        printf("%d is an odd number.\n", number);
    }

    // Example of ternary operator (shorthand if-else)
    printf("Using ternary operator: %d is %s\n",
           number, (number % 2 == 0) ? "even" : "odd");

    return 0;
}
```

Code Comments: The % (modulo) operator gives the remainder after division. For even numbers, number % 2 equals 0. The ternary operator ?: is a compact way to write simple if-else statements.

Explanation: if-else creates a binary branch in the code: exactly one of the two blocks will execute based on the condition. The ternary operator (condition ? true_value : false_value) is useful for simple conditionals.

6.3 else-if Ladder

```
#include <stdio.h>

int main() {
    int score;  // To store exam score
    char grade;  // To store calculated grade

    // Get input from user
    printf("Enter your exam score (0-100): ");
    scanf("%d", &score);

    // Input validation
    if (score < 0 || score > 100) {  // Check for invalid score range
        printf("Error: Score must be between 0 and 100\n");
        return 1;  // Exit with error code
    }

    // Determine grade using else-if ladder
    if (score >= 90) {
        grade = 'A';  // 90-100 = A grade
    } else if (score >= 80) {
        grade = 'B';  // 80-89 = B grade
    } else if (score >= 70) {
        grade = 'C';  // 70-79 = C grade
    } else if (score >= 60) {
        grade = 'D';  // 60-69 = D grade
    } else {
        grade = 'F';  // Below 60 = F grade
    }

    // Display the result
    printf("Your grade is: %c\n", grade);

    // Additional feedback based on grade
    if (grade == 'A' || grade == 'B') {
        printf("Excellent performance!\n");
    } else if (grade == 'F') {
        printf("You need to study harder for the next exam.\n");
    }

    return 0;  // Successful execution
}
```

Code Comments: An else-if ladder evaluates multiple conditions in sequence. Once a true condition is found, its block executes and the rest are skipped. Input validation is a good practice to handle unexpected user inputs.

Explanation: The order of conditions matters! If we reversed the order (checking for score >= 60 first), all scores would match that condition, and higher grades would never be assigned.

6.4 switch Statement

```
#include <stdio.h>

int main() {
    // Variable declaration
    int day;

    // Get input from user
    printf("Enter a day number (1-7): ");
    scanf("%d", &day);

    // Use switch to select appropriate day name
    switch (day) {
        case 1:  // If day equals 1
            printf("Monday\n");
            break;  // Exit the switch block

        case 2:  // If day equals 2
            printf("Tuesday\n");
            break;

        case 3:  // If day equals 3
            printf("Wednesday\n");
            break;

        case 4:  // If day equals 4
            printf("Thursday\n");
            break;

        case 5:  // If day equals 5
            printf("Friday\n");
            break;

        case 6:  // If day equals 6
            printf("Saturday\n");
            break;

        case 7:  // If day equals 7
            printf("Sunday\n");
            break;

        default:  // If day doesn't match any case
            printf("Invalid day number! Please enter 1-7.\n");
            // No break needed for default (it's the last case)
    }

    // Example of case fall-through (no break)
    printf("\nIs it a weekend? ");
    switch (day) {
        case 6:  // Saturday
        case 7:  // Sunday
            printf("Yes, it's a weekend!\n");
            break;
        case 1:  // Monday
        case 2:  // Tuesday
        case 3:  // Wednesday
        case 4:  // Thursday
        case 5:  // Friday
            printf("No, it's a weekday.\n");
            break;
        default:
            printf("Invalid day number!\n");
    }

    return 0;
}
```

Code Comments: Each case represents a possible value of the variable. The break statement prevents fall-through to the next case. The default case handles values that don't match any case.

Explanation: In the second switch example, we deliberately omit break statements to demonstrate 'fall-through' behavior, where multiple case labels execute the same code block. This can be useful for cases that should be handled identically.

Practice Questions:

1. What's the difference between if-else and switch statements?

2. When would you use a switch statement instead of an if-else ladder?

3. What happens if you forget a break statement in a switch case?

4. Write a program to check if a year is a leap year.

5. Create a program that takes a letter grade and outputs the corresponding GPA.

6. What are the limitations of the switch statement in C?

7. Loops in C

Loops allow you to execute a block of code repeatedly based on a condition. They are essential for tasks that require repetition like processing arrays, reading user input until a specific value is entered, or performing calculations multiple times.

7.1 while Loop

```
#include <stdio.h>

int main() {
    // Example 1: Simple counter with while loop
    int count = 1;      // Initialize counter

    printf("Counting from 1 to 5 using while loop:\n");

    while (count <= 5) {  // Loop condition: keep going while count is <= 5
        printf("%d ", count);  // Print current count value
        count++;              // Increment count (VERY IMPORTANT to avoid
infinite loop)
    }
    printf("\nLoop finished! count = %d\n\n", count);

    // Example 2: Sum of numbers 1 to 10
    int i = 1;          // Loop counter
    int sum = 0;        // Initialize sum

    while (i <= 10) {   // Continue while i is less than or equal to 10
        sum += i;       // Add current value of i to sum
        i++;            // Increment i
    }

    printf("Sum of numbers from 1 to 10 is: %d\n\n", sum);

    // Example 3: User input validation
    int number;
    int attempts = 0;

    printf("Enter a positive number: ");
    scanf("%d", &number);
    attempts++;

    while (number <= 0) {  // Continue asking if input is invalid
        printf("Invalid input! Please enter a positive number: ");
        scanf("%d", &number);
        attempts++;
    }

    printf("Thank you! You entered: %d\n", number);
    printf("It took you %d attempts to enter a valid number.\n", attempts);
    return 0;
}
```

Code Comments: Always ensure the loop condition will eventually become false by updating variables inside the loop. Forgetting to increment the counter (e.g., count++) would cause an infinite loop.

Explanation: The while loop is a pre-test loop - the condition is checked BEFORE each iteration. If the condition is initially false, the loop body never executes. While loops are ideal when you don't know in advance how many iterations you need.

7.2 do-while Loop

```
#include <stdio.h>

int main() {
    // Example 1: Simple counter with do-while loop
    int count = 1;       // Initialize counter

    printf("Counting from 1 to 5 using do-while loop:\n");

    do {
        printf("%d ", count);  // Print current count value
        count++;               // Increment count
    } while (count <= 5);      // Condition checked AFTER loop body executes

    printf("\nLoop finished! count = %d\n\n", count);

    // Example 2: Difference between while and do-while
    // Case where initial condition is false
    int x = 10;

    printf("while loop (condition initially false):\n");
    while (x < 5) {
        printf("This will NOT be printed\n");
        x++;
    }

    x = 10;  // Reset x
    printf("do-while loop (condition initially false):\n");
    do {
        printf("This will be printed ONCE\n");
        x++;
    } while (x < 5);

    // Example 3: Menu-driven program
    int choice;

    do {
        printf("\nMenu:\n");
        printf("1. Option One\n");
        printf("2. Option Two\n");
        printf("3. Option Three\n");
        printf("0. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        // Process user choice
        switch (choice) {
            case 1:
                printf("You selected Option One\n");
                break;
            case 2:
                printf("You selected Option Two\n");
                break;
            case 3:
                printf("You selected Option Three\n");
                break;
            case 0:
                printf("Exiting program...\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }

    } while (choice != 0);  // Continue until user chooses to exit

    return 0;
}
```

Code Comments: The key difference: do-while loops ALWAYS execute at least once, while a while loop might execute zero times if the condition is initially false.

Explanation: The do-while loop is a post-test loop - the condition is checked AFTER each iteration. It's perfect for menu-driven programs where you want to display the menu at least once, then keep showing it until the user chooses to exit.

7.3 for Loop

```
#include <stdio.h>

int main() {
    // Example 1: Basic for loop - count from 1 to 5
    printf("Counting from 1 to 5 using for loop:\n");

    /*
     * for loop has three components:
     * 1. Initialization: int i = 1
     * 2. Condition: i <= 5
     * 3. Increment/Update: i++
     */
    for (int i = 1; i <= 5; i++) {
```

```c
        printf("%d ", i);
    }
    printf("\n\n");

    // Example 2: Sum of first 10 natural numbers
    int sum = 0;

    for (int i = 1; i <= 10; i++) {
        sum += i;  // Add each number to sum
    }

    printf("Sum of numbers from 1 to 10: %d\n\n", sum);

    // Example 3: Countdown (decrementing loop)
    printf("Countdown:\n");
    for (int i = 10; i >= 1; i--) {
        printf("%d ", i);
    }
    printf("Blast off!\n\n");

    // Example 4: Print even numbers between 1 and 20
    printf("Even numbers between 1 and 20:\n");
    for (int i = 2; i <= 20; i += 2) {  // Start at 2, increment by 2
        printf("%d ", i);
    }
    printf("\n\n");

    // Example 5: Multiple variables in for loop
    printf("Multiple variables in for loop:\n");
    for (int i = 1, j = 10; i <= 10; i++, j--) {
        printf("%d + %d = %d\n", i, j, i + j);
    }

    return 0;
}
```

Code Comments: The for loop is ideal when you know exactly how many iterations you need. It combines initialization, condition, and update in one line, making the code more compact and readable.

Explanation: All three parts of the for loop are optional. You can create an infinite loop with for(;;) or move any of the parts outside the loop definition. The scope of variables declared in the initialization part is limited to the loop body.

7.4 Nested Loops

```c
#include <stdio.h>

int main() {
    // Example 1: Simple nested loops to print a pattern
    printf("Pattern using nested loops:\n");
    for (int i = 1; i <= 5; i++) {            // Outer loop: controls rows
        for (int j = 1; j <= i; j++) {         // Inner loop: controls columns
            printf("* ");                      // Print star with space
        }
        printf("\n");                          // Move to next line after
inner loop
    }
    printf("\n");

    // Example 2: Multiplication table
    printf("Multiplication Table (1-5):\n");

    // Print header row
    printf("   |");
    for (int i = 1; i <= 5; i++) {
        printf(" %2d", i);  // %2d ensures width of 2 for alignment
    }
    printf("\n---+---------------\n");  // Separator line

    // Print table body
    for (int i = 1; i <= 5; i++) {        // Outer loop: rows
        printf("%2d |", i);               // Row label

        for (int j = 1; j <= 5; j++) {    // Inner loop: columns
            printf(" %2d", i * j);        // Print product
        }

        printf("\n");                     // End of row
    }

    // Example 3: Find all pairs of numbers that sum to 10
    printf("\nPairs of numbers (1-9) that sum to 10:\n");

    for (int a = 1; a <= 9; a++) {          // First number
        for (int b = a; b <= 9; b++) {      // Second number (start from a to
avoid duplicates)
            if (a + b == 10) {
                printf("%d + %d = 10\n", a, b);
            }
        }
    }

    return 0;
}
```

Code Comments: In nested loops, the inner loop completes all its iterations for each single iteration of the outer loop. This is perfect for working with multi-dimensional data like matrices or generating patterns.

Explanation: For each iteration of the outer loop, the inner loop runs completely. With i=1 in Example 1, j loops from 1 to 1, printing one star. With i=2, j loops from 1 to 2, printing two stars, and so on, creating a triangle pattern.

7.5 break and continue Statements

```c
#include <stdio.h>

int main() {
    // Example 1: Using break to exit a loop early
    printf("Example with break statement:\n");
    printf("Finding the first multiple of 7 between 1 and 30:\n");

    for (int i = 1; i <= 30; i++) {
        if (i % 7 == 0) {
            printf("Found it! %d is a multiple of 7\n", i);
            break;  // Exit the loop immediately when found
        }
        printf("Checking %d...\n", i);
    }
    printf("Loop finished\n\n");

    // Example 2: Using continue to skip iterations
    printf("Example with continue statement:\n");
    printf("Printing only odd numbers from 1 to 10:\n");

    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            continue;  // Skip the rest of this iteration if number is even
        }
        printf("%d ", i);  // This only executes for odd numbers
    }
    printf("\n\n");

    // Example 3: break in nested loops
    printf("break in nested loops:\n");

    for (int i = 1; i <= 3; i++) {
        printf("Outer loop iteration %d: ", i);

        for (int j = 1; j <= 5; j++) {
            if (j == 4) {
                printf("(breaking inner loop at j=4) ");
                break;  // Only breaks out of the INNER loop
            }
            printf("%d ", j);
        }

        printf("\n");
    }
    printf("\n");

    // Example 4: Labeled break equivalent (using flags)
    printf("Simulating labeled break with a flag:\n");

    int found = 0;  // Flag to indicate when to break outer loop

    for (int i = 1; i <= 3 && !found; i++) {
        printf("Outer loop iteration %d: ", i);

        for (int j = 1; j <= 5; j++) {
            printf("%d ", j);

            if (i == 2 && j == 3) {
                printf("(target found at i=2, j=3) ");
                found = 1;  // Set flag to true
                break;      // Break inner loop
            }
        }

        printf("\n");
    }

    return 0;
}
```

Code Comments: break immediately exits the current loop. continue skips the rest of the current iteration and jumps to the next iteration. These statements help control loop flow in special cases.

Explanation: In nested loops, break only exits the innermost loop containing it. To exit multiple nested loops, you can use a flag variable as shown in Example 4 (C doesn't have labeled breaks like some other languages).

Practice Questions:

1. What's the difference between while and do-while loops?

2. When would you choose a for loop over a while loop?

3. Write a program to find the factorial of a number using a loop.

4. Create a program that prints the first 10 Fibonacci numbers.

5. What happens if you use break in a nested loop?

6. Write a program that prints all prime numbers between 1 and 50.

7. How would you create an infinite loop intentionally? How would you safely exit it?

8. Arrays in C

Arrays are collections of elements of the same data type stored in contiguous memory locations. They allow you to store multiple values of the same type under a single variable name, making it easy to manage related data.

8.1 One-Dimensional Arrays

```
#include <stdio.h>

int main() {
    // Example 1: Array declaration and initialization

    // Method 1: Declare and initialize in separate steps
    int scores[5];            // Declare array of 5 integers
    scores[0] = 85;           // Initialize first element (index 0)
    scores[1] = 92;           // Initialize second element (index 1)
    scores[2] = 78;           // Initialize third element (index 2)
    scores[3] = 88;           // Initialize fourth element (index 3)
    scores[4] = 95;           // Initialize fifth element (index 4)

    // Method 2: Declare and initialize in one step
    int grades[5] = {90, 85, 78, 92, 88};  // Compact initialization

    // Method 3: Let compiler determine array size
    int values[] = {10, 20, 30, 40, 50, 60};  // Size determined
automatically (6)

    // Method 4: Partial initialization
    int data[5] = {5, 10};    // First two elements initialized, rest are 0
                              // Same as {5, 10, 0, 0, 0}

    // Example 2: Accessing array elements
    printf("First score: %d\n", scores[0]);    // Access first element
    printf("Third grade: %d\n", grades[2]);    // Access third element

    // Example 3: Using loops with arrays
    printf("\nAll scores:\n");
    for (int i = 0; i < 5; i++) {
        printf("scores[%d] = %d\n", i, scores[i]);
    }

    // Example 4: Calculate sum and average
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum += scores[i];     // Add each element to sum
    }
    float average = (float)sum / 5;  // Cast to float for decimal result

    printf("\nSum of all scores: %d\n", sum);
    printf("Average score: %.2f\n", average);

    // Example 5: Finding maximum value
    int max = scores[0];  // Start with first element
    for (int i = 1; i < 5; i++) {  // Start from second element
        if (scores[i] > max) {
            max = scores[i];  // Update max if larger value found
        }
    }
    printf("Highest score: %d\n", max);

    return 0;
}
```

Code Comments: Arrays in C are zero-indexed, meaning the first element is at index 0. Array bounds are not checked automatically in C - accessing outside the array (like scores[10] for a 5-element array) can cause undefined behavior.

Explanation: When partially initializing an array, unspecified elements are automatically set to zero. Also note the (float) cast when calculating the average - without this, integer division would truncate the decimal portion.

8.2 Two-Dimensional Arrays

```
#include <stdio.h>

int main() {
    // Example 1: Declaring and initializing 2D arrays

    // Method 1: Declare and initialize separately
    int matrix[3][4];  // 3 rows, 4 columns

    // Initialize row 0
    matrix[0][0] = 1;  matrix[0][1] = 2;  matrix[0][2] = 3;  matrix[0][3] = 4;
    // Initialize row 1
    matrix[1][0] = 5;  matrix[1][1] = 6;  matrix[1][2] = 7;  matrix[1][3] = 8;
    // Initialize row 2
    matrix[2][0] = 9;  matrix[2][1] = 10; matrix[2][2] = 11; matrix[2][3] =
12;

    // Method 2: Declare and initialize in one step (row-by-row)
    int grid[3][3] = {
        {1, 2, 3},    // Row 0
        {4, 5, 6},    // Row 1
        {7, 8, 9}     // Row 2
    };

    // Example 2: Accessing 2D array elements
    printf("Value at matrix[1][2]: %d\n", matrix[1][2]);  // Should be 7
    printf("Value at grid[2][0]: %d\n\n", grid[2][0]);    // Should be 7

    // Example 3: Printing a 2D array with nested loops
    printf("Printing the entire matrix (3x4):\n");

    for (int i = 0; i < 3; i++) {          // Loop for rows
        for (int j = 0; j < 4; j++) {      // Loop for columns
            printf("%3d ", matrix[i][j]);  // %3d ensures even spacing
        }
        printf("\n");  // New line after each row
    }
    printf("\n");

    // Example 4: Calculate row sums
    printf("Row sums of the matrix:\n");

    for (int i = 0; i < 3; i++) {
        int rowSum = 0;

        for (int j = 0; j < 4; j++) {
            rowSum += matrix[i][j];  // Add each element in row
        }

        printf("Sum of row %d: %d\n", i, rowSum);
    }
    printf("\n");

    // Example 5: Calculate column sums
    printf("Column sums of the matrix:\n");

    for (int j = 0; j < 4; j++) {    // Loop through each column
        int colSum = 0;

        for (int i = 0; i < 3; i++) {  // Loop through each row
            colSum += matrix[i][j];    // Add each element in column
        }

        printf("Sum of column %d: %d\n", j, colSum);
    }

    return 0;
}
```

Code Comments: 2D arrays can be visualized as a table with rows and columns. Memory allocation is row-major in C, meaning the entire array is stored row by row in sequential memory locations.

Explanation: In a 2D array declaration int arr[3][4], the first number [3] is the number of rows, and the second [4] is the number of columns. Nested loops are essential for processing 2D arrays - typically the outer loop iterates through rows and the inner loop through columns.

Practice Questions:

1. What happens if you access an array element outside its bounds in C?

2. How would you find the minimum value in an array of integers?

3. Write a program to reverse the elements of an array.

4. How would you calculate the sum of all elements in a 2D array?

5. What's the difference between int arr[5] = {0}; and int arr[5] = {0, 0, 0, 0, 0};

6. Write a program to perform matrix addition (adding two 2D arrays).

9. Structures in C

Structures (struct) in C allow you to group variables of different data types under a single name. They are useful for representing complex entities with multiple attributes, such as a student record, product information, or geometric shapes.

9.1 Defining and Using Structures

```c
#include <stdio.h>
#include <string.h>  // For strcpy() function

// Define a structure for a student record
struct Student {
    int id;                 // Student ID
    char name[50];          // Student name (array of characters)
    float gpa;              // Grade Point Average
    char grade;             // Letter grade
    int active;             // Boolean flag (1 = active, 0 = inactive)
};

int main() {
    // Example 1: Declaring a structure variable and initializing members
    struct Student student1;  // Declare a variable of Student type

    // Assign values to structure members using dot operator
    student1.id = 1001;
    strcpy(student1.name, "Alice Johnson");  // strcpy for string assignment
    student1.gpa = 3.8;
    student1.grade = 'A';
    student1.active = 1;

    // Example 2: Initialization at declaration time
    struct Student student2 = {1002, "Bob Smith", 3.5, 'B', 1};

    // Example 3: Designated initializers (C99 feature)
    struct Student student3 = {
        .id = 1003,
        .name = "Charlie Davis",
        .gpa = 3.9,
        .grade = 'A',
        .active = 1
    };

    // Example 4: Accessing structure members
    printf("Student 1:\n");
    printf("ID: %d\n", student1.id);
    printf("Name: %s\n", student1.name);
    printf("GPA: %.2f\n", student1.gpa);
    printf("Grade: %c\n", student1.grade);
    printf("Status: %s\n\n", student1.active ? "Active" : "Inactive");
    // Example 5: Modifying structure members
    printf("Updating student2's information...\n");
    student2.gpa = 3.7;  // Change GPA
    strcpy(student2.name, "Robert Smith");  // Change name
    student2.grade = 'A';  // Change grade

    printf("Updated Student 2:\n");
    printf("Name: %s\n", student2.name);
    printf("GPA: %.2f\n", student2.gpa);
    printf("Grade: %c\n\n", student2.grade);

    // Example 6: Copying structures
    struct Student student4 = student3;  // Copy all values from student3
    printf("Student 4 (copied from Student 3):\n");
    printf("ID: %d, Name: %s, GPA: %.2f\n\n", student4.id, student4.name, 
student4.gpa);

    return 0;
}
```

Code Comments: In C, we can't directly assign strings with = operator to char arrays; we must use strcpy(). The dot (.) operator accesses structure members. Unlike arrays, entire structures can be copied with assignment operator.

Explanation: Structures help organize related data together. The struct Student definition creates a new data type that we can use to declare variables. Each structure variable (like student1) contains all the members defined in the structure.

9.2 Array of Structures

```
#include <stdio.h>
#include <string.h>

// Define a structure for a product
struct Product {
    int id;              // Product ID
    char name[50];       // Product name
    float price;         // Price in dollars
    int stock;           // Quantity in stock
};

int main() {
    // Example 1: Declaring an array of structures
    struct Product inventory[5];  // Array of 5 Product structures

    // Example 2: Initializing array elements
    // Product 1
    inventory[0].id = 101;
    strcpy(inventory[0].name, "Laptop");
    inventory[0].price = 899.99;
    inventory[0].stock = 15;
```

```c
// Product 2
inventory[1].id = 102;
strcpy(inventory[1].name, "Smartphone");
inventory[1].price = 499.50;
inventory[1].stock = 25;

// Product 3 - Direct initialization
inventory[2] = (struct Product){103, "Tablet", 349.99, 10};

// Example 3: Initializing at declaration time
struct Product catalog[3] = {
    {201, "Headphones", 59.99, 30},
    {202, "Mouse", 25.50, 45},
    {203, "Keyboard", 45.99, 20}
};

// Example 4: Processing array of structures with loops
printf("Inventory Report:\n");
printf("---------------------------\n");
printf("ID  |  Name        |  Price   | Stock\n");
printf("---------------------------\n");

// Print first 3 products from inventory array
for (int i = 0; i < 3; i++) {
    printf("%3d | %-12s | $%7.2f | %3d\n",
           inventory[i].id,
           inventory[i].name,
           inventory[i].price,
           inventory[i].stock);
}

// Example 5: Calculate total inventory value
float totalValue = 0.0;
for (int i = 0; i < 3; i++) {
    totalValue += inventory[i].price * inventory[i].stock;
}

printf("\nTotal inventory value: $%.2f\n\n", totalValue);

// Example 6: Finding product with lowest stock
int lowestStockIndex = 0;

for (int i = 1; i < 3; i++) {
    if (inventory[i].stock < inventory[lowestStockIndex].stock) {
        lowestStockIndex = i;
    }
}
```

```
    printf("Product with lowest stock: %s (Only %d in stock)\n",
           inventory[lowestStockIndex].name,
           inventory[lowestStockIndex].stock);

    return 0;
}
```

Code Comments: Arrays of structures are perfect for managing collections of related objects. Each element in the array is a complete structure with all its members. Notice the formatting in printf to create a well-aligned table output.

Explanation: When accessing elements in an array of structures, we first use [] to access the array element, then use . to access the specific member: inventory[i].price. This pattern is extremely common when processing collections of data.

Practice Questions:

1. What's the difference between an array and a structure in C?

2. How would you create a structure to store information about a car?

3. Why can't we assign strings directly in C (e.g., student.name = "John")?

4. Create a program that uses a structure to store and calculate rectangle properties (length, width, area, perimeter).

5. Write a function that takes a structure as an argument and prints its contents.

6. Design an array of structures to maintain a small library catalog.

10. Functions in C

Functions are reusable blocks of code that perform specific tasks. They help break down complex programs into smaller, manageable pieces, making code more organized, readable, and maintainable.

10.1 Function Declaration and Definition

```c
#include <stdio.h>

// Function declarations (prototypes) - tells compiler about function
int add(int a, int b);                // Function prototype
void greetUser(char name[]);          // Void function prototype
float calculateArea(float radius);    // Float return type

// Function definitions (implementations)

/*
 * Function: add
 * Purpose: Adds two integers and returns the result
 * Parameters: a - first integer, b - second integer
 * Returns: Sum of a and b
 */
int add(int a, int b) {
    int sum = a + b;     // Calculate sum
    return sum;          // Return the result to caller
}

/*
 * Function: greetUser
 * Purpose: Displays a greeting message
 * Parameters: name - user's name as string
 * Returns: Nothing (void)
 */
void greetUser(char name[]) {
    printf("Hello, %s! Welcome to our program.\n", name);
    // void functions don't need return statement
}

/*
 * Function: calculateArea
 * Purpose: Calculates area of a circle
 * Parameters: radius - radius of the circle
 * Returns: Area as floating-point number
 */
float calculateArea(float radius) {
    const float PI = 3.14159;  // Local constant
    float area = PI * radius * radius;
    return area;
}
```

```c
// Main function - program entry point
int main() {
    // Example 1: Using functions with return values
    int x = 10, y = 20;
    int result = add(x, y);  // Call function and store result

    printf("Sum of %d and %d is: %d\n", x, y, result);

    // Can also use function call directly in expressions
    printf("Sum of 5 and 8 is: %d\n", add(5, 8));

    // Example 2: Using void functions
    char userName[50] = "Alice";
    greetUser(userName);  // Call void function

    // Example 3: Using functions in calculations
    float radius = 5.0;
    float circleArea = calculateArea(radius);

    printf("Area of circle with radius %.2f is: %.2f\n", radius, circleArea);

    // Example 4: Function calls can be nested
    float largerArea = calculateArea(add(3, 2));  // add(3,2) returns 5
    printf("Area of circle with radius 5 is: %.2f\n", largerArea);

    return 0;
}
```

Code Comments: Function prototypes tell the compiler about functions before they're used. Parameters are local copies of the values passed from main. Functions create their own scope - variables inside functions are separate from main.

Explanation: The return statement immediately exits the function and sends a value back to the caller. Functions can call other functions (like using add() inside calculateArea() call). Each function has its own memory space for local variables.

10.2 Recursion

```c
#include <stdio.h>

/*
 * Recursive function to calculate factorial
 * Factorial of n (n!) = n × (n-1) × (n-2) × ... × 1
 * Base case: 0! = 1, 1! = 1
 * Recursive case: n! = n × (n-1)!
 */
int factorial(int n) {
    // Base case: stop recursion when n reaches 0 or 1
    if (n <= 1) {
        printf("Base case reached: factorial(%d) = 1\n", n);
        return 1;
    }

    // Recursive case: n! = n × (n-1)!
    printf("Calculating factorial(%d) = %d * factorial(%d)\n", n, n, n-1);
    int result = n * factorial(n - 1);  // Function calls itself
    printf("Returning factorial(%d) = %d\n", n, result);

    return result;
}

/*
 * Recursive function to calculate Fibonacci numbers
 * Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
 * Formula: F(n) = F(n-1) + F(n-2) where F(0) = 0, F(1) = 1
 */
int fibonacci(int n) {
    // Base cases
    if (n == 0) return 0;  // F(0) = 0
    if (n == 1) return 1;  // F(1) = 1

    // Recursive case: F(n) = F(n-1) + F(n-2)
    return fibonacci(n - 1) + fibonacci(n - 2);
}

/*
 * Function to calculate sum of digits using recursion
 * Example: sumDigits(123) = 1 + 2 + 3 = 6
 */
int sumDigits(int num) {
    // Base case: single digit number
    if (num < 10) {
        return num;
    }

    // Recursive case: last digit + sum of remaining digits
    int lastDigit = num % 10;          // Get last digit
    int remainingNumber = num / 10;    // Remove last digit

    return lastDigit + sumDigits(remainingNumber);
}

int main() {
    // Example 1: Factorial calculation
    printf("Factorial Example:\n");
    printf("------------------\n");
    int number = 5;
    printf("Calculating factorial of %d:\n", number);
    int fact = factorial(number);
    printf("\nFinal result: %d! = %d\n\n", number, fact);
    // Example 2: Fibonacci sequence
    printf("Fibonacci Example:\n");
    printf("-------------------\n");
    printf("First 10 Fibonacci numbers:\n");
    for (int i = 0; i < 10; i++) {
        printf("F(%d) = %d\n", i, fibonacci(i));
    }
    printf("\n");

    // Example 3: Sum of digits
    printf("Sum of Digits Example:\n");
    printf("----------------------\n");
    int testNum = 12345;
    int digitSum = sumDigits(testNum);
    printf("Sum of digits in %d = %d\n", testNum, digitSum);

    return 0;
}
```

Code Comments: Recursion requires two essential parts: a base case that stops the recursion, and a recursive case where the function calls itself with a modified parameter. Without a proper base case, you get infinite recursion and stack overflow.

Explanation: Each recursive call creates a new function context on the call stack. The factorial example shows how recursion works: factorial(5) calls factorial(4), which calls factorial(3), etc., until reaching the base case, then returns values back up the chain.

Practice Questions:

1. What's the difference between a function declaration and definition?

2. What happens if a function doesn't have a return statement but is supposed to return a value?

3. Write a function that finds the maximum of three numbers.

4. Explain what would happen in recursive factorial if we forgot the base case.

5. Create a recursive function to calculate the power of a number (e.g., 2^5).

6. What are the advantages and disadvantages of using recursion versus loops?

11. Pointers in C

Pointers are variables that store memory addresses of other variables. They are one of the most powerful features of C, enabling dynamic memory allocation, efficient array manipulation, and direct memory access.

11.1 Basic Pointer Operations

```c
#include <stdio.h>

int main() {
    // Example 1: Basic pointer declaration and usage
    int number = 42;          // Regular variable
    int *ptr = &number;       // Pointer to integer, initialized with address
of number

    printf("Basic Pointer Operations:\n");
    printf("========================\n");

    // Displaying values and addresses
    printf("Value of number: %d\n", number);          // Direct access
    printf("Address of number: %p\n", &number);       // Address using &
operator
    printf("Value of ptr (address): %p\n", ptr);      // Pointer value
(address)
    printf("Value pointed by ptr: %d\n", *ptr);       // Dereferencing pointer
    printf("Address of ptr itself: %p\n\n", &ptr);    // Address of pointer
variable

    // Example 2: Modifying values through pointers
    printf("Modifying through pointers:\n");
    printf("==========================\n");

    printf("Before: number = %d\n", number);
    *ptr = 100;  // Change value at the address pointed by ptr
    printf("After *ptr = 100: number = %d\n\n", number);

    // Example 3: Pointer arithmetic
    int arr[5] = {10, 20, 30, 40, 50};
    int *arrPtr = arr;  // Points to first element (arr and &arr[0] are same)

    printf("Pointer Arithmetic with Arrays:\n");
    printf("===============================\n");

    for (int i = 0; i < 5; i++) {
        printf("arr[%d] = %d, ", i, arr[i]);
        printf("*(arrPtr + %d) = %d, ", i, *(arrPtr + i));
        printf("Address: %p\n", arrPtr + i);
    }
    printf("\n");
    // Example 4: Incrementing pointer
    printf("Incrementing pointer through array:\n");
    printf("=====================================\n");

    arrPtr = arr;  // Reset to first element
    for (int i = 0; i < 5; i++) {
        printf("Value: %d, Address: %p\n", *arrPtr, arrPtr);
        arrPtr++;  // Move to next element
    }

    return 0;
}
```

*Code Comments: The & operator gets the address of a variable, and the * operator dereferences a pointer (gets the value at that address). When incrementing a pointer, it moves by the size of the data type it points to.*

Explanation: Pointer arithmetic is scaled by the size of the data type. For an int pointer, ptr++ moves 4 bytes forward (assuming int is 4 bytes). This is why array[i] and *(ptr + i) are equivalent - both access the element at offset i from the starting address.

11.2 Pointers and Arrays

```
#include <stdio.h>

int main() {
    // Example 1: Array name as pointer
    int numbers[5] = {100, 200, 300, 400, 500};

    printf("Array name as pointer:\n");
    printf("======================\n");
    printf("numbers (array name): %p\n", numbers);
    printf("&numbers[0]: %p\n", &numbers[0]);
    printf("Both are the same address!\n\n");

    // Example 2: Different ways to access array elements
    printf("Different ways to access array elements:\n");
    printf("=========================================\n");

    for (int i = 0; i < 5; i++) {
        // All four methods below are equivalent
        printf("Element %d: ", i);
        printf("arr[%d]=%d  ", i, numbers[i]);            // Array notation
        printf("*(numbers+%d)=%d  ", i, *(numbers + i)); // Pointer
arithmetic
        printf("*(arr+%d)=%d\n", i, *(numbers + i));     // Same as above
    }
    printf("\n");

    // Example 3: Using pointer to traverse array backwards
    int *ptr = numbers + 4;  // Point to last element
    printf("Traversing array backwards with pointer:\n");
    printf("========================================\n");

    for (int i = 4; i >= 0; i--) {
        printf("Element %d: %d (Address: %p)\n", i, *ptr, ptr);
        ptr--;  // Move to previous element
    }
    printf("\n");

    // Example 4: Modifying array through pointer
    ptr = numbers;  // Reset to first element

    printf("Doubling all array elements using pointer:\n");
    printf("==========================================\n");

    printf("Before: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");

    // Double each element
    for (int i = 0; i < 5; i++) {
        *ptr = *ptr * 2;  // Double the value
        ptr++;            // Move to next element
    }

    printf("After:  ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n\n");

    return 0;
}
```

Code Comments: In C, an array name is a constant pointer to the first element. You can't change what an array name points to (like numbers = something_else), but you can use it in pointer arithmetic to access any element.

Explanation: The expressions arr[i], *(arr + i), *(i + arr), and even i[arr] are all equivalent in C! This flexibility allows for various programming styles and optimizations.

11.3 Pointer to Pointer

```
#include <stdio.h>

int main() {
    // Example 1: Basic pointer to pointer
    int value = 100;        // Regular variable
    int *ptr = &value;      // Pointer to int
```

```c
    int **ptrToPtr = &ptr;  // Pointer to pointer to int

    printf("Pointer to Pointer Example:\n");
    printf("===========================\n");

    printf("value = %d\n", value);
    printf("Address of value = %p\n", &value);
    printf("\n");

    printf("ptr = %p (points to value)\n", ptr);
    printf("*ptr = %d (value through ptr)\n", *ptr);
    printf("Address of ptr = %p\n", &ptr);
    printf("\n");

    printf("ptrToPtr = %p (points to ptr)\n", ptrToPtr);
    printf("*ptrToPtr = %p (address stored in ptr)\n", *ptrToPtr);
    printf("**ptrToPtr = %d (value through double dereference)\n",
**ptrToPtr);
    printf("Address of ptrToPtr = %p\n\n", &ptrToPtr);

    // Example 2: Modifying value through double pointer
    printf("Modifying value through double pointer:\n");
    printf("========================================\n");

    printf("Before: value = %d\n", value);
    **ptrToPtr = 250;  // Change value through double pointer
    printf("After **ptrToPtr = 250: value = %d\n\n", value);

    // Example 3: Array of pointers (useful for strings)
    char *fruits[4] = {"Apple", "Banana", "Cherry", "Date"};
    char **fruitPtr = fruits;  // Pointer to first element of pointer array

    printf("Array of pointers example:\n");
    printf("=========================\n");

    for (int i = 0; i < 4; i++) {
        printf("fruits[%d] = %s\n", i, fruits[i]);
        printf("*(fruitPtr + %d) = %s\n", i, *(fruitPtr + i));
        printf("fruitPtr[%d] = %s\n\n", i, fruitPtr[i]);
    }

    // Example 4: Memory layout visualization
    printf("Memory Layout Visualization:\n");
    printf("===========================\n");
    printf("Variable    | Address    | Value/Points to\n");
    printf("------------|------------|----------------\n");
    printf("value       | %p | %d\n", &value, value);
    printf("ptr         | %p | %p\n", &ptr, ptr);
    printf("ptrToPtr    | %p | %p\n", &ptrToPtr, ptrToPtr);
    return 0;
}
```

Code Comments: Double pointers are pointers that store the address of another pointer. They're commonly used for dynamic memory allocation, string arrays, and when you need to modify a pointer in a function.

Explanation: To access the final value through a double pointer, you need double dereferencing (**ptrToPtr). Each * operator removes one level of indirection. This concept can extend to triple pointers (***) and beyond, though they're rarely used in practice.

Practice Questions:

1. What's the difference between &variable and *pointer?

2. Why do we say 'array name is a pointer' in C?

3. What happens if you try to dereference a NULL pointer?

4. Write a program to swap two numbers using pointers.

5. How would you access the third element of an array using pointer arithmetic?

6. Explain why pointer arithmetic is scaled by the data type size.

12. File Handling in C

File handling in C allows programs to store and retrieve data from external files on disk. This enables data persistence - information can be saved and accessed even after the program terminates, making it essential for practical applications.

12.1 Basic File Operations

```c
#include <stdio.h>
#include <stdlib.h>  // For exit() function

int main() {
    // Example 1: Writing to a file
    printf("Writing to a file...\n");

    FILE *writeFile = fopen("output.txt", "w");  // Open file for writing

    // Always check if file opened successfully
    if (writeFile == NULL) {
        printf("Error: Could not create/open file for writing!\n");
        return 1;  // Exit with error code
    }

    // Write data to file using fprintf (similar to printf)
    fprintf(writeFile, "Hello, World!\n");
    fprintf(writeFile, "This is line 2\n");
    fprintf(writeFile, "Number: %d\n", 42);
    fprintf(writeFile, "Float: %.2f\n", 3.14);

    // Always close files when done
    fclose(writeFile);
    printf("Data written to output.txt successfully!\n\n");

    // Example 2: Reading from the file we just created
    printf("Reading from the file...\n");

    FILE *readFile = fopen("output.txt", "r");  // Open file for reading

    if (readFile == NULL) {
        printf("Error: Could not open file for reading!\n");
        return 1;
    }

    char line[100];  // Buffer to store each line

    // Read file line by line using fgets
    printf("File contents:\n");
    printf("=============\n");

    while (fgets(line, sizeof(line), readFile) != NULL) {
        printf("%s", line);  // Print each line (fgets includes \n)
    }

    fclose(readFile);
    printf("\nFile read successfully!\n\n");

    // Example 3: Appending to a file
    printf("Appending to the file...\n");

    FILE *appendFile = fopen("output.txt", "a");  // Open in append mode

    if (appendFile == NULL) {
        printf("Error: Could not open file for appending!\n");
        return 1;
    }

    fprintf(appendFile, "This line was appended\n");
    fprintf(appendFile, "Current time: (would be timestamp)\n");

    fclose(appendFile);
    printf("Data appended successfully!\n");

    return 0;
}
```

Code Comments: Always check if fopen() returns NULL (file couldn't be opened). Use fclose() to save changes and free system resources. Different modes: 'w' (write/overwrite), 'r' (read), 'a' (append).

Explanation: The FILE* pointer acts as a handle to the file. fprintf() works like printf() but outputs to a file instead of the screen. fgets() reads one line at a time until it reaches the end of file (EOF), at which point it returns NULL.

12.2 File Modes and Functions

Mode	Description
r	Read only. File must exist. Fails if file doesn't exist.
w	Write only. Creates new file or overwrites existing file.
a	Append only. Creates new file or adds to end of existing file.
r+	Read and write. File must exist.
w+	Read and write. Creates new file or overwrites existing.
a+	Read and append. Creates new file or opens existing for reading and appending.

12.3 Student Record Management System

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

```c
#include <string.h>

// Structure to represent a student record
struct Student {
    int id;
    char name[50];
    float gpa;
    char major[30];
};

// Function to add a new student record to file
void addStudent() {
    FILE *file = fopen("students.txt", "a");  // Open in append mode

    if (file == NULL) {
        printf("Error: Cannot open file!\n");
        return;
    }

    struct Student student;

    // Get student information from user
    printf("Enter student ID: ");
    scanf("%d", &student.id);

    printf("Enter student name: ");
    scanf(" %49s", student.name);  // Space before % to skip whitespace

    printf("Enter GPA: ");
    scanf("%f", &student.gpa);

    printf("Enter major: ");
    scanf(" %29s", student.major);

    // Write to file in formatted way
    fprintf(file, "%d %s %.2f %s\n",
            student.id, student.name, student.gpa, student.major);

    fclose(file);
    printf("Student record added successfully!\n\n");
}

// Function to display all student records
void displayStudents() {
    FILE *file = fopen("students.txt", "r");

    if (file == NULL) {
        printf("No student records found or cannot open file!\n");
        return;
    }

    struct Student student;

    printf("\nStudent Records:\n");
    printf("================\n");
    printf("ID    | Name          | GPA  | Major\n");
    printf("------|---------------|------|------------------\n");

    // Read records using fscanf
    while (fscanf(file, "%d %s %f %s",
                  &student.id, student.name, &student.gpa, student.major) ==
4) {
        printf("%-6d| %-14s| %-5.2f| %s\n",
               student.id, student.name, student.gpa, student.major);
    }

    fclose(file);
    printf("\n");
}

// Function to search for a student by ID
void searchStudent() {
    FILE *file = fopen("students.txt", "r");

    if (file == NULL) {
        printf("No student records found!\n");
        return;
    }

    int searchId;
    printf("Enter student ID to search: ");
    scanf("%d", &searchId);

    struct Student student;
    int found = 0;  // Flag to check if student was found

    // Search through file
    while (fscanf(file, "%d %s %f %s",
                  &student.id, student.name, &student.gpa, student.major) ==
4) {
        if (student.id == searchId) {
            printf("\nStudent Found:\n");
            printf("ID: %d\n", student.id);
            printf("Name: %s\n", student.name);
            printf("GPA: %.2f\n", student.gpa);
            printf("Major: %s\n\n", student.major);
            found = 1;
            break;
```

```c
        }
    }

    if (!found) {
        printf("Student with ID %d not found!\n\n", searchId);
    }

    fclose(file);
}

// Main function with menu system
int main() {
    int choice;

    printf("Student Record Management System\n");
    printf("================================\n");

    do {
        // Display menu
        printf("\nMenu Options:\n");
        printf("1. Add Student\n");
        printf("2. Display All Students\n");
        printf("3. Search Student by ID\n");
        printf("4. Exit\n");
        printf("Enter your choice (1-4): ");

        scanf("%d", &choice);

        // Process user choice
        switch (choice) {
            case 1:
                addStudent();
                break;
            case 2:
                displayStudents();
                break;
            case 3:
                searchStudent();
                break;
            case 4:
                printf("Exiting program. Goodbye!\n");
                break;
            default:
                printf("Invalid choice! Please enter 1-4.\n");
        }

    } while (choice != 4);  // Continue until user chooses to exit
    return 0;
}
```

Code Comments: This complete system demonstrates practical file handling: appending new records, reading all records, and searching. Notice how fscanf() returns the number of successfully read items, which we use to detect end of file.

Explanation: The system uses text format for simplicity. In real applications, you might use binary files for efficiency. The search function demonstrates sequential file access - reading from beginning until the target is found or file ends.

Practice Questions:

1. What's the difference between 'w' and 'a' file modes?

2. Why is it important to check if fopen() returns NULL?

3. How would you count the number of lines in a text file?

4. What happens if you forget to call fclose() on a file?

5. Write a program that copies the contents of one file to another.

6. How would you modify the student system to allow updating existing records?

Document prepared by: Abdelhamid Khald

June 2025