Manuale di Programmazione in C

Alberto Abate

13 August 2025

Contents

Chapter 1

C Basics

```c
/*
    ================================================================================
    Tutorial: C Basics - A Simple Function for Summation
    ================================================================================

    Welcome to your C tutorial! This first exercise covers some of the most
    fundamental concepts in C programming.

    Concepts Covered:
    - The #include directive for using standard libraries.
    - The main() function: the entry point of every C program.
    - Declaring integer variables.
    - Using printf() for output to the console.
    - Using scanf() to read user input.
    - Defining and calling a simple function.
    - The return statement.
*/

// The #include directive tells the compiler to include the contents of a file.
// <stdio.h> is the standard input/output library in C. It provides functions
// like printf() and scanf().
#include <stdio.h>

/*
    This is a user-defined function named 'sum'.
    - It takes two integer arguments, 'x' and 'y'.
    - It returns an integer value, which is the result of their addition.
    - Defining functions like this helps in organizing code and reusing it.
*/
int sum(int x, int y) {
    // The 'return' statement provides the result of the function call.
    return x + y;
}

/*
    The main() function is special. It's where the execution of any C program begins.
    'int' before main indicates that the function returns an integer value.
    'void' in the parentheses means it takes no arguments.
*/
int main(void) {
    // Variable Declaration:
    // We declare two integer variables, 'a' and 'b', to store the numbers
    // provided by the user.
    int a, b;

    // We use printf() to display a message to the user, prompting them for input.
    // The '\n' at the end of a string is an "escape sequence" that represents a newline
    ↪  character.
    printf("Please enter two numbers separated by a space: ");

    // Using scanf() to read input from the user:
    // - The first argument "%d%d" is a format string. It tells scanf() to expect
    //   two integer values.
    // - The following arguments, &a and &b, are the memory addresses of the variables
    //   where the input values should be stored. The '&' is the "address-of" operator.
    scanf("%d%d", &a, &b);

    // Calling our 'sum' function:
    // We pass the user's numbers (a and b) to our sum() function.
    // The returned value is then used as an argument to printf().
    printf("The result of the sum is %d.\n", sum(a, b));

    // The 'return 0;' statement at the end of main() indicates that the program
    // has executed successfully. A non-zero value typically indicates an error.
    return 0;
}

/*
    ===============================================================================
    Further Exploration:
    ===============================================================================
    1.  Can you modify this program to calculate the product of the two numbers
        instead of the sum? You'll need to create a new function called 'product'.
    2.  What happens if the user enters a non-integer value (like "hello")?
        Experiment and see. We'll cover how to handle such errors later.
    3.  Try creating functions for subtraction and division. Remember that division
        with integers might not give you the result you expect if there's a
        remainder!
    ===============================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: C Basics - Integer Division and the Modulo Operator
    ================================================================================

    This exercise demonstrates how integer arithmetic works in C, specifically
    division and finding the remainder.

    Concepts Covered:
    - Integer division: how C handles division between two integers.
    - The Modulo Operator (%): how to get the remainder of a division.
    - Variable initialization.
*/

#include <stdio.h>

int main(void) {
    // Variable Declaration and Initialization:
    // We declare two integer variables to store the total number of matches
    // and the capacity (size) of each box.
    // It's good practice to initialize variables to 0 to avoid using them
    // with garbage values from memory.
    int total_matches = 0;
    int box_size = 0;

    printf("Enter the total number of matches: ");
    scanf("%d", &total_matches);

    printf("Enter the number of matches each box can hold: ");
    scanf("%d", &box_size);

    // Integer Division:
    // When you divide two integers in C, the result is also an integer.
    // Any fractional part is truncated (simply cut off, not rounded).
    // For example, 10 / 3 results in 3.
    int full_boxes = total_matches / box_size;
    printf("Number of full boxes: %d\n", full_boxes);

    // The Modulo Operator (%):
    // This operator gives you the remainder of an integer division.
    // For example, 10 % 3 results in 1, because 10 divided by 3 is 3 with
    // a remainder of 1.
    int remaining_matches = total_matches % box_size;
    printf("Number of remaining matches: %d\n", remaining_matches);

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  What happens if the user enters 0 for the box size? Your program will
        likely crash due to a "division by zero" error. This is a very common
        bug to watch out for!
    2.  Can you add an 'if' statement to check if 'box_size' is 0 before you
        perform the division? If it is, print an error message to the user.
    3.  Think about a real-world scenario where the modulo operator would be
        useful. For example, determining if a number is even or odd. (Hint:
```

```
        An even number has a remainder of 0 when divided by 2).
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: C Basics - Reading Multiple Inputs with a Loop
    ================================================================================

    This program demonstrates how to use a loop to read a specific number of
    inputs from the user and calculate their sum.

    Concepts Covered:
    - The 'for' loop for repeating actions a set number of times.
    - Using a counter variable in a loop.
    - Combining scanf() with a loop for multiple inputs.
    - Accumulating a sum.
*/

#include <stdio.h>

int main() {
    // how_many_to_sum: stores the total count of numbers the user wants to add.
    int how_many_to_sum = 0;
    // current_number: stores the number entered by the user in each iteration of the loop.
    int current_number = 0;
    // sum: an "accumulator" variable, initialized to 0, to store the running total.
    int sum = 0;

    printf("How many numbers do you want to sum? ");
    // Read the total count from the user.
    scanf("%d", &how_many_to_sum);

    printf("Great! Please enter %d numbers, one at a time.\n", how_many_to_sum);

    // The 'for' loop:
    // This is a powerful construct for repeating a block of code.
    // It consists of three parts, separated by semicolons:
    // 1. Initialization (int i = 0): Runs once at the beginning. Creates a counter 'i'.
    // 2. Condition (i < how_many_to_sum): Checked before each iteration. The loop
    //    continues as long as this condition is true.
    // 3. Post-iteration (i++): Runs at the end of each iteration. Increments the counter.
    for (int i = 0; i < how_many_to_sum; i++) {
        printf("Enter number %d: ", i + 1); // We use i+1 to show a 1-based count to the user.
        scanf("%d", &current_number);

        // Accumulation:
        // We add the number just entered to our running total.
        // 'sum = sum + current_number;' is often written with the shorthand 'sum +=
        ↪  current_number;'.
        sum = sum + current_number;

        printf("You entered %d. The current sum is %d.\n", current_number, sum);
    }

    printf("\nAll numbers have been entered.\n");
    printf("The final sum is: %d\n", sum);

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  Can you modify this program to calculate the average of the numbers instead
        of the sum? You would calculate the sum first, then divide by 'how_many_to_sum'.
        Be careful about integer division vs. floating-point division! You might
        need to use 'double' or 'float' variables.
    2.  What if you wanted to find the largest number entered instead of the sum?
        You would need a new variable, maybe called 'largest_so_far', and an 'if'
        statement inside the loop to update it.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: C Basics - Type Casting for Accurate Division
    ================================================================================

    This program calculates the average of a set of grades. It highlights a
    critical concept in C: type casting, which is necessary to get an accurate
    decimal result from a division of two integers.

    Concepts Covered:
    - The problem with integer division when a decimal result is needed.
    - Type Casting: Explicitly converting a value from one data type to another.
    - Using 'double' for floating-point numbers.
    - The '%lf' format specifier for printing doubles.
*/

#include <stdio.h>

int main(void) {
    int num_grades = 0;
    int current_grade = 0;
    int sum = 0;
    double average = 0.0; // Use a 'double' for the average to store decimal values.

    printf("How many grades do you want to enter? ");
    scanf("%d", &num_grades);

    printf("Enter %d grades, one at a time:\n", num_grades);
    for (int i = 0; i < num_grades; i++) {
        printf("Enter grade #%d: ", i + 1);
        scanf("%d", &current_grade);
        sum += current_grade; // Shorthand for sum = sum + current_grade
    }

    // The Problem with Integer Division:
    // If we were to write 'average = sum / num_grades;', C would perform
    // integer division, because both 'sum' and 'num_grades' are integers.
    // The result would be an integer, and any decimal part would be lost
    // *before* it gets assigned to the 'average' variable.
    // For example, if sum=10 and num_grades=4, `10 / 4` would result in 2, not 2.5.

    // The Solution: Type Casting
    // To fix this, we must convert one of the operands to a floating-point type
    // *before* the division happens. We do this by prefixing the variable with
    // the desired type in parentheses, like `(double)sum`.
    // When one operand is a double, C performs floating-point division,
    // preserving the decimal part of the result.
    average = (double)sum / num_grades;

    // The '%.2lf' format specifier tells printf to display a double ('lf')
    // rounded to two decimal places ('.2').
    printf("The average of the grades is: %.2lf\n", average);

    return 0;
}

/*
    ================================================================================
    Further Exploration:
```

```
    ================================================================================
    1.  Remove the `(double)` cast from the calculation and see what happens.
        Enter numbers that should result in a decimal average (e.g., 3 grades: 5, 5, 6).
        You'll see the incorrect result firsthand.
    2.  What if you cast the entire result, like `(double)(sum / num_grades)`?
        Try it out. You'll find that this doesn't work, because the integer
        division happens first inside the parentheses *before* the cast can occur.
        This demonstrates that the cast must happen on an operand *before* the division.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: C Basics - The 'char' Data Type
    ================================================================================

    This exercise introduces the 'char' data type, which is used to store
    single characters.

    Concepts Covered:
    - The 'char' data type for single characters.
    - Character literals, enclosed in single quotes (e.g., 'A').
    - The '%c' format specifier for printing characters.
*/

#include <stdio.h>

int main(void) {
    // Variable Declaration and Initialization:
    // We declare three variables of type 'char'.
    // A 'char' holds a single character value.
    // Character literals in C are enclosed in single quotes (' ').
    // Note the difference: 'A' is a char, while "A" is a string.
    char letter1 = 'i';
    char letter2 = 'n';
    char letter3 = 'C';

    // The '%c' format specifier in printf is used to print a single character.
    // We provide the char variables as arguments, and they are substituted
    // for the %c placeholders in the output.
    printf("Programming %c%c %c\n", letter1, letter2, letter3);

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  In C, characters are actually stored as small integer values according to
        a standard like ASCII. You can see this by printing a char variable
        using the '%d' (integer) format specifier instead of '%c'.
        Try it: `printf("The ASCII value of %c is %d\n", letter3, letter3);`
    2.  You can also perform arithmetic on char variables. What do you think
        `letter3 + 1` would produce if you print it as a character? Try it out!
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Control Flow - Nested Loops
    ================================================================================

    This program simulates a game with multiple players (or "throws") where each
    player rolls multiple dice. It's a perfect example of a "nested loop",
    which is one loop inside another.

    The filename `06_random_numbers.c` is a bit of a misnomer, as this program
    doesn't generate *random* numbers. Instead, it reads a series of dice values
    that you provide as input. Later in the tutorial, we'll see how to generate
    truly random numbers.

    Concepts Covered:
    - Nested 'for' loops.
    - Resetting an accumulator variable for an inner loop.
    - Simulating a 2D problem (throws and dice) with loops.
*/

#include <stdio.h>

int main(void) {
    int num_throws = 0; // The number of players or turns in the game (the "outer" loop).
    int num_dice = 0;   // The number of dice each player rolls (the "inner" loop).
    int dice_value = 0; // The value of a single die, read from input.
    int throw_sum = 0;  // The sum of dice values for the current player's turn.

    printf("Enter the number of throws and the number of dice per throw: ");
    scanf("%d %d", &num_throws, &num_dice);

    // Outer Loop: Iterates once for each player/throw.
    // 'throw_idx' is the index for the current throw, from 0 to num_throws-1.
    for (int throw_idx = 0; throw_idx < num_throws; throw_idx++) {

        // Inner Loop: Iterates once for each die within a single throw.
        // 'dice_idx' is the index for the current die, from 0 to num_dice-1.
        printf("\nEnter the %d dice values for throw #%d: ", num_dice, throw_idx + 1);
        for (int dice_idx = 0; dice_idx < num_dice; dice_idx++) {
            scanf("%d", &dice_value);
            throw_sum += dice_value; // Add the die's value to the sum for this throw.
        }

        // This printf is inside the outer loop but *after* the inner loop has finished.
        // This means it runs once per throw, after all dice for that throw are summed up.
        printf("Throw %d sum equals %d\n", throw_idx + 1, throw_sum);

        // Resetting the accumulator:
        // It's crucial to reset the sum to 0 after each throw. If we didn't,
        // the sum for the second throw would start with the total from the first,
        // which is not what we want.
        throw_sum = 0;
    }

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  What would happen if you moved the `throw_sum = 0;` line to the very
        beginning of the program, before the outer loop? The program would
        calculate a grand total of all dice rolls instead of a per-throw sum.
    2.  Can you modify this program to find the highest single die roll within
        each throw? You'd need another variable, `highest_roll_this_throw`, and
        an 'if' statement inside the inner loop. Remember to reset it in the
        outer loop!
    ================================================================================

    Example Input to Copy/Paste:
    (First line is for the first scanf, the rest are for the loops)

    5 4
    1 4 2 3
    3 2 6 4
    2 4 1 4
    2 2 1 4
    3 6 1 2
*/
```

```c
/*
    ================================================================================
    Tutorial: C Basics - A Simple Unit Converter
    ================================================================================

    This program is a simple unit converter that brings together many of the
    concepts we've learned so far. It converts meters to feet, grams to pounds,
    and Celsius to Fahrenheit.

    This is a great example of a complete, small program.

    Concepts Covered:
    - Function prototyping (declaring functions before they are defined).
    - Breaking a problem down into smaller functions.
    - Using a loop to process multiple conversions.
    - Conditional logic with 'if-else if-else'.
    - The 'switch' statement for multi-way branching.
    - Using 'const' for defining constants.
*/

#include <stdio.h>

// Function Prototypes:
// These lines tell the main function about the functions we plan to use later.
// This is necessary because we define them *after* main(), but we call them
// *inside* main(). A prototype specifies the function's return type, name,
// and the types of its arguments.
double perform_conversion(double value, char input_unit);

int main(void) {
    int num_conversions = 0;
    double input_value = 0.0;
    double output_value = 0.0;
    char input_unit_char; // e.g., 'm', 'g', 'c'

    printf("How many conversions would you like to perform? ");
    scanf("%d", &num_conversions);

    for (int i = 0; i < num_conversions; i++) {
        printf("\nEnter the value and unit (m, g, or c) for conversion #%d: ", i + 1);
        scanf("%lf %c", &input_value, &input_unit_char);

        // We call our conversion function to get the result.
        output_value = perform_conversion(input_value, input_unit_char);

        // Check if the conversion was successful. Our function returns a special
        // value (-1.0 in this case, a simple error handling method) to indicate
        // an invalid unit was entered.
        if (output_value == -1.0) {
            printf("Error: Invalid unit '%c'. Please use 'm', 'g', or 'c'.\n", input_unit_char);
        } else {
            // Here we use a 'switch' statement to print the correct unit label.
            // A switch is often cleaner than a long if-else-if chain when you are
            // checking a single variable against multiple constant values.
            printf("Result: %.6lf ", output_value);
            switch (input_unit_char) {
                case 'm':
                    printf("ft\n");
                    break; // The 'break' is important! It exits the switch.
                case 'g':
                    printf("lbs\n");
                    break;
                case 'c':
                    printf("F\n");
                    break;
            }
        }
    }

    return 0;
}

/*
    This function contains the logic for the conversions.
    It takes the value and the input unit character.
    It returns the converted value, or -1.0 if the unit is invalid.
*/
double perform_conversion(double value, char input_unit) {
    // Using 'const' declares a variable whose value cannot be changed.
    // This is good practice for conversion factors and other magic numbers.
    const double METERS_TO_FEET = 3.2808;
    const double GRAMS_TO_POUNDS = 0.002205;

    double result = 0.0;

    switch (input_unit) {
        case 'm': // meters to feet
            result = value * METERS_TO_FEET;
            break;
        case 'g': // grams to pounds
            result = value * GRAMS_TO_POUNDS;
            break;
        case 'c': // Celsius to Fahrenheit
            // The formula is: F = (C * 9/5) + 32.  1.8 is the same as 9/5.
            result = 32.0 + (1.8 * value);
            break;
        default: // Handle invalid units
            result = -1.0; // Use a special value to signal an error
            break;
    }

    return result;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  Add more conversions! For example, add a case for converting from
        kilograms ('k') to pounds. You'll need to add it to the switch
        statement and provide the correct conversion factor.
    2.  This program only converts from metric to imperial. Can you add the
        reverse conversions? You might need to add new character codes (e.g., 'f'
        for feet) and expand the logic to handle them.
    ================================================================================
*/
```

Chapter 2

Control Flow

```c
/*
    ================================================================================
    Tutorial: Control Flow - The 'if' Statement
    ================================================================================

    This program demonstrates the most fundamental control flow statement: the 'if'
    statement. It allows the program to make decisions and execute different
    code blocks based on a condition.

    This example was recreated to replace a file that was lost due to technical
    issues. It captures the concept of checking a temperature value.

    Concepts Covered:
    - The 'if' statement for conditional execution.
    - Relational operators (e.g., > for "greater than").
    - The 'else' statement for handling the case when the condition is false.
*/

#include <stdio.h>

int main(void) {
    int temperature;

    printf("Enter the current temperature in Celsius: ");
    scanf("%d", &temperature);

    // The 'if' statement checks a condition in the parentheses.
    // If the condition is true, the block of code immediately following it is executed.
    if (temperature > 30) {
        printf("It's hot outside! Stay hydrated.\n");
    }

    // The 'else' statement is optional.
    // Its code block is executed if the 'if' condition is false.
    else {
        printf("It's not too hot. Enjoy the weather!\n");
    }

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  Can you add an 'else if' statement to handle more conditions? For example,
        check for temperatures below 10 degrees Celsius and print a message
        saying "It's cold, wear a jacket!".
    2.  What happens if you only have an 'if' without an 'else'? Try removing the
        'else' block and see how the program behaves for different temperatures.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Control Flow - Logical Operators and a Bug Fix
    ================================================================================

    This program simulates a simple dice game where a tax is calculated based
    on the sum of two dice. It's a good example of nested 'if' statements.

    This file also contains a subtle bug that provides a great learning opportunity
    about logical operators!

    Concepts Covered:
    - Nested 'if' statements.
    - Boolean logic stored in an integer (a common C idiom).
    - The logical AND (&&) vs. logical OR (||) operators.
*/

#include <stdio.h>

int main(void) {
    int dice1 = 0;
    int dice2 = 0;
    int sum = 0;

    printf("Enter the values for two dice: ");
    scanf("%d %d", &dice1, &dice2);

    // A common C idiom: Storing a boolean result in an int.
    // The expression `sum >= 10` evaluates to either 1 (true) or 0 (false).
    // This result is then stored in the integer variable 'is_special_total'.
    sum = dice1 + dice2;
    int is_special_total = (sum >= 10);

    // --- A BUG TO FIX ---
    // The original code had a bug here: `(dice1 >= 1 || dice1 <= 6)`.
    // The logical OR (||) operator is true if *either* condition is true. Since
    // any number is either "greater than or equal to 1" OR "less than or equal to 6",
    // this condition was always true, even for invalid dice rolls like 99.
    //
    // The FIX is to use the logical AND (&&) operator, which is only true if
    // *both* conditions are true. This correctly validates the dice roll.
    if ((dice1 >= 1 && dice1 <= 6) && (dice2 >= 1 && dice2 <= 6)) {
        // This is a valid dice roll, so we can proceed.
        printf("The roll is valid.\n");

        // Nested 'if' statement:
        // This 'if' is checked only if the outer 'if' was true.
        if (is_special_total) { // This is the same as writing `if (is_special_total == 1)`
            printf("This is a special total!\n");
            printf("Special tax: 36\n");
        } else {
            printf("This is a regular total.\n");
            printf("Regular tax: %d\n", 2 * sum);
        }

    } else {
        // This 'else' belongs to the outer 'if'. It catches invalid dice rolls.
        printf("Error: Invalid dice values entered. Please enter numbers between 1 and 6.\n");
    }
    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  Change the `&&` back to `||` in the validation 'if' statement and try
        entering invalid dice numbers like `7` or `-2`. See how the program
        incorrectly processes them. This is a great way to understand the bug.
    2.  The variable 'is_special_total' is convenient but not strictly necessary.
        Can you rewrite the program to not use it, putting the `sum >= 10`
        check directly into the nested 'if' statement?
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Control Flow - Tug of War
    ================================================================================

    This program simulates a tug of war game by summing the weights of players
    on two teams and determining which team has the higher total weight.

    Concepts Covered:
    - Using a 'for' loop to process a set number of players.
    - Accumulating totals for multiple groups within a single loop.
    - Simple 'if' statement for final comparison.
    - Handling multiple conditions with 'if-else'.
*/

#include <stdio.h>

int main(void) {
    int num_players_per_team = 0;
    int current_weight = 0;
    int team1_total_weight = 0;
    int team2_total_weight = 0;
    int winning_team = 0;

    printf("Enter the number of players on each team: ");
    scanf("%d", &num_players_per_team);

    printf("Enter the weights for each player, alternating between Team 1 and Team 2.\n");

    // This loop iterates once for each *pair* of players (one from each team).
    for (int i = 0; i < num_players_per_team; i++) {
        // Read weight for the player from Team 1
        printf("Enter weight for Team 1, player %d: ", i + 1);
        scanf("%d", &current_weight);
        team1_total_weight += current_weight;

        // Read weight for the player from Team 2
        printf("Enter weight for Team 2, player %d: ", i + 1);
        scanf("%d", &current_weight);
        team2_total_weight += current_weight;
    }

    // Determine the winner.
    // The original code defaulted the winner to Team 2. Let's make the logic
    // more explicit by checking for all three conditions: Team 1 wins, Team 2 wins, or a tie.
    if (team1_total_weight > team2_total_weight) {
        printf("Team 1 has the advantage.\n");
    } else if (team2_total_weight > team1_total_weight) {
        printf("Team 2 has the advantage.\n");
    } else {
        printf("It's a tie! The teams are perfectly matched.\n");
    }

    printf("Total weight for Team 1: %d\n", team1_total_weight);
    printf("Total weight for Team 2: %d\n", team2_total_weight);

    return 0;
}
```

```
/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  The current input method is a bit awkward. Can you restructure the
        program to use two separate 'for' loops? The first loop would read all
        the weights for Team 1, and the second loop would read all the weights
        for Team 2. This is a more user-friendly design.
    2.  Instead of just declaring a winner, can you calculate and print by how
        much weight the winning team is ahead? (e.g., "Team 1 wins by 50 lbs").
        You might need the `abs()` function from the `<stdlib.h>` library to get
        the absolute difference.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Control Flow - Simplifying with 'else if'
    ================================================================================

    This program calculates a cost based on a person's age and weight, using
    a set of rules. The original code uses nested 'if-else' statements.

    We will refactor this code to use the 'else if' structure, which is often
    cleaner and easier to read when you have a chain of mutually exclusive conditions.

    Concepts Covered:
    - Nested 'if-else' statements.
    - The 'else if' ladder for cleaner multi-condition logic.
    - Logical AND (&&) to combine conditions.
*/

#include <stdio.h>

int main(void) {
    int age = 0;
    int weight = 0; // in pounds
    int cost = 0;

    printf("Enter age: ");
    scanf("%d", &age);

    printf("Enter weight in pounds: ");
    scanf("%d", &weight);

    /*
        Original Logic (with nested if-else):
        This works, but it can be hard to read as more conditions are added,
        leading to deeper and deeper nesting (the "arrowhead" anti-pattern).

        if (age == 60) {
            cost = 0;
        } else {
            if (age < 10) {
                cost = 5;
            } else {
                cost = 30;
                if (weight > 20) {
                    cost = cost + 10;
                }
            }
        }
    */

    // Refactored Logic (with 'else if' ladder):
    // This structure is much flatter and easier to follow. The conditions are
    // checked from top to bottom. As soon as one is true, its block is executed,
    // and the rest of the chain is skipped.
    if (age == 60) {
        cost = 0;
    } else if (age < 10) {
        cost = 5;
    } else { // This block handles everyone else (age >= 10 and not 60)
        cost = 30;
        // This nested 'if' is now much simpler. It only applies to the group
        // that qualifies for the base $30 cost.
        if (weight > 20) {
            cost = cost + 10;
        }
    }

    printf("The calculated cost is: %d\n", cost);

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  The rules in this program are a bit ambiguous. For example, what is the
        cost for a person who is exactly 10 years old? The current logic charges
        them $30. Is that correct? Modify the conditions to handle the age of 10
        explicitly if you want a different cost.
    2.  Add a new rule: if a person is over 100 years old, the cost is also 0.
        Where would you add this 'else if' statement in the chain? Does the
        order matter? (Hint: Yes, it can! You'd want to check for > 100 before
        the final 'else' block).
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Control Flow - Decisions Based on String Length
    ================================================================================

    This program reads a word from the user and then prints either 1 or 2
    based on whether the length of the word is even or odd.

    The original code calculates the string length manually with a 'while' loop.
    We will show that method and then introduce the standard C library function
    `strlen()` from `<string.h>`, which is the correct and more efficient way
    to do this.

    Concepts Covered:
    - Character arrays (strings) in C.
    - The null terminator ('\0') at the end of a string.
    - Manually calculating string length with a loop.
    - Using the standard `strlen()` function from the `<string.h>` library.
    - The ternary operator as a shortcut for 'if-else'.
*/

// We include <string.h> to get access to standard string functions like strlen().
#include <stdio.h>
#include <string.h>

int main(void) {
    // Declare a character array to hold the user's input.
    // We make it size 51 to hold a 50-character word plus the null terminator.
    char word[51];
    int length = 0;
    int output_value = 0;

    printf("Enter a single word (up to 50 characters): ");
    // When using scanf with %s, it reads a sequence of non-whitespace characters.
    // It automatically adds the null terminator '\0' at the end.
    scanf("%s", word);

    /*
        Method 1: Manual Calculation (as in the original code)
        This is a great way to understand how strings work in C. The loop
        continues until it finds the special null terminator character ('\0')
        that marks the end of every string.

        int i = 0;
        while (word[i] != '\0') {
            i++;
        }
        length = i;
    */

    // Method 2: The Standard Library Function `strlen()` (Preferred)
    // This function does the same loop for us, but it's optimized and less
    // error-prone. It returns the number of characters in the string, not
    // including the null terminator.
    length = strlen(word);
    printf("The length of the word \"%s\" is %d.\n", word, length);

    // Now, we make a decision based on the length.
    if (length % 2 == 0) {
        // The length is even.
        output_value = 1;
    } else {
        // The length is odd.
        output_value = 2;
    }
    printf("The output value is: %d\n", output_value);

    // Bonus: The Ternary Operator
    // For a simple if-else that assigns a value to a single variable, C provides
    // a shortcut called the ternary operator `(condition ? value_if_true : value_if_false)`.
    int ternary_output = (length % 2 == 0) ? 1 : 2;
    printf("Result using the ternary operator: %d\n", ternary_output);


    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  What happens if the user enters a sentence with spaces, like "hello world"?
        `scanf("%s", ...)` stops reading at the first space. To read a whole line
        of text, you would need a different function, like `fgets()`.
    2.  Try to write a program that prints "VOWEL" if a user-entered word starts
        with a vowel (a, e, i, o, u) and "CONSONANT" otherwise. You would need to
        check the first character of the string, which is `word[0]`.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Control Flow - The 'while' Loop
    ================================================================================

    This program calculates the total power of a fleet of robots based on their
    attributes. It's a great example of reading and processing multiple,
    more complex lines of input.

    The original code used a 'for' loop. We have refactored it to use a 'while'
    loop to demonstrate an alternative looping structure.

    Concepts Covered:
    - The 'while' loop.
    - The difference between 'for' and 'while' loops.
    - Reading multiple values from a single line of input.
    - Performing a calculation within a loop to accumulate a total.
*/

#include <stdio.h>

int main(void) {
    int num_robots = 0;
    int height, weight, engine_power, resistance;
    int total_fleet_power = 0;
    int i = 0; // It's common to initialize the counter *before* a while loop.

    printf("How many robots are in the fleet? ");
    scanf("%d", &num_robots);

    printf("Enter the stats for each robot (height weight power resistance):\n");

    // The 'while' loop:
    // A 'while' loop is simpler than a 'for' loop. It only has one part: the
    // condition. It will continue to execute its block as long as the condition
    // is true. It's up to us to manage the counter variable inside the loop.
    while (i < num_robots) {
        printf("Robot #%d: ", i + 1);
        scanf("%d %d %d %d", &height, &weight, &engine_power, &resistance);

        // Calculate the power for the current robot
        int single_robot_power = (engine_power + resistance) * (weight - height);

        // Add it to the fleet's total power
        total_fleet_power += single_robot_power;

        // Manual Increment:
        // We must remember to increment our counter variable 'i' at the end
        // of the loop. If we forget this, `i < num_robots` will always be true,
        // and we will have an infinite loop!
        i++;
    }

    printf("The total power of the robot fleet is: %d\n", total_fleet_power);

    return 0;
}

/*
```

```
    ============================================================================
    When to use 'for' vs. 'while'?

    - 'for' loop: Best when you know exactly how many times you want to loop
      (e.g., iterating from 0 to 'n', looping through an array). The initialization,
      condition, and increment are all neatly packaged in one line. This is the
      most common type of loop.

    - 'while' loop: Best when you want to loop as long as a certain condition
      is true, but you don't necessarily know how many iterations it will take.
      (e.g., "keep looping until the user enters -1", or "keep looping while
      the file has more data").

    In this specific example, a 'for' loop is arguably a better fit because we
    know the exact number of iterations from the start (`num_robots`). However,
    it's important to understand how to accomplish the same task with a 'while' loop.
    ============================================================================
*/
```

Chapter 3

Functions

```c
/*
    ================================================================================
    Tutorial: Functions - Decomposition and Reusability
    ================================================================================

    This program demonstrates one of the most powerful concepts in programming:
    decomposition. We break down a complex problem (drawing shapes) into smaller,
    manageable, and reusable pieces called functions.

    Notice how the `printTriangle` and `printRectangle` functions both reuse
    the `printLine` function. This avoids code duplication and makes the program
    easier to understand and maintain.

    Concepts Covered:
    - Defining 'void' functions (functions that don't return a value).
    - Passing arguments to functions.
    - Function prototyping.
    - Reusing functions (one function calling another).
*/

#include <stdio.h>

// Function Prototypes:
// We declare the functions here so that `main` knows about them before they are
// called. This allows us to define the functions in any order we want after `main`.
void printLine(int nCols, char pattern);
void printTriangle(int nLines, char pattern);
void printRectangle(int nLines, int nCols, char pattern);

int main(void) {
    int nCols;
    int nLines;

    printf("--- Printing a Line ---\n");
    printf("How many columns would you like? ");
    scanf("%d", &nCols);
    printLine(nCols, 'X');
    printf("\n");

    printf("--- Printing a Triangle ---\n");
    printf("How many lines would you like? ");
    scanf("%d", &nLines);
    printTriangle(nLines, '*');
    printf("\n");

    printf("--- Printing a Rectangle ---\n");
    // We can reuse the nLines and nCols variables from before.
    printf("Using %d lines and %d columns.\n", nLines, nCols);
    printRectangle(nLines, nCols, '#');

    return 0;
}

/*
    Function: printLine
    Purpose: Prints a single line of characters of a specified length and pattern.
    Parameters:
        - nCols: The number of characters to print.
        - pattern: The character to use for printing.
    Returns: void (nothing)
*/
void printLine(int nCols, char pattern) {
    for (int i = 0; i < nCols; i++) {
        printf("%c", pattern);
    }
    printf("\n"); // Move to the next line after printing the characters.
}

/*
    Function: printTriangle
    Purpose: Prints a right-angled triangle of a specified height.
    This function *calls* `printLine` repeatedly to do its work.
*/
void printTriangle(int nLines, char pattern) {
    // The outer loop controls which line we are on (from 1 to nLines).
    for (int line = 1; line <= nLines; line++) {
        // For each line, we call printLine, telling it to print 'line' number of characters.
        // Line 1 prints 1 char, Line 2 prints 2 chars, and so on.
        printLine(line, pattern);
    }
}

/*
    Function: printRectangle
    Purpose: Prints a solid rectangle of a specified height and width.
    This function also *calls* `printLine` repeatedly.
*/
void printRectangle(int nLines, int nCols, char pattern) {
    for (int i = 0; i < nLines; i++) {
        // For each line of the rectangle, we simply print a full line of characters.
        printLine(nCols, pattern);
    }
}

/*
    ==============================================================================
    Further Exploration:
    ==============================================================================
    1.  Can you create a new function `printSquare(int size, char pattern)` that
        calls `printRectangle`? This shows another layer of decomposition.
    2.  Create a function `printHollowRectangle(int nLines, int nCols, char pattern)`.
        For this, you'll need to modify the logic inside. The top and bottom lines
        will be solid, but the middle lines will only have the pattern character
        at the start and end, with spaces in between. This is a good challenge!
    ==============================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Functions - Pass-by-Value vs. Pass-by-Reference
    ================================================================================

    This program demonstrates a critical concept: how to make a function modify
    a variable that belongs to another function (in this case, `main`).

    In C, arguments are "passed by value". This means when you pass a variable
    like `a` to a function, the function gets a *copy* of its value. Any changes
    to that copy inside the function do not affect the original variable in `main`.

    To get around this, we use "pass-by-reference". We don't pass the variable's
    value; instead, we pass its *memory address* using a pointer. The function
    can then use this address to find and modify the original variable.

    Concepts Covered:
    - Pointers as function arguments.
    - The "address-of" operator (&) to get a variable's address.
    - The "dereference" operator (*) to access the value at an address.
    - How pointers enable a function to "return" a value through its arguments.
*/

#include <stdio.h>

// Function Prototype:
// Notice the third argument: `int *`. This declares the argument as a
// pointer to an integer. It's a variable that will hold the memory address
// of another integer variable.
void add(int a, int b, int *sum_pointer);

int main(void) {
    int num1, num2, sum;

    printf("Please enter two integers: ");
    scanf("%d%d", &num1, &num2);

    // Calling the function:
    // We pass num1 and num2 by value (the function gets copies of them).
    // For the third argument, we use the address-of operator (&) to pass the
    // *memory address* of our 'sum' variable. We are not passing the value of
    // sum (which is garbage at this point), but its location in memory.
    add(num1, num2, &sum);

    // Because the 'add' function had the address of 'sum', it was able to
    // modify it directly. When we print 'sum' here, it will have the new
    // value calculated by the function.
    printf("Back in main: %d + %d = %d\n", num1, num2, sum);

    return 0;
}

/*
    Function: add
    Purpose: Calculates the sum of two integers and stores the result in a
             variable provided by the caller via a pointer.
    Parameters:
        - a: The first integer (passed by value).
        - b: The second integer (passed by value).
        - sum_pointer: A pointer holding the memory address of the integer
                       variable where the result should be stored.
*/
void add(int a, int b, int *sum_pointer) {
    int result = a + b;

    printf("Inside add(): The calculated sum is %d.\n", result);

    // Modifying the variable in main():
    // We use the dereference operator (*) on the pointer.
    // `*sum_pointer` means "the value at the address that sum_pointer holds".
    // By assigning to `*sum_pointer`, we are not changing the pointer itself,
    // but rather the value in the memory location it points to. This directly
    // changes the 'sum' variable back in the main() function.
    *sum_pointer = result;

    printf("Inside add(): Updated the value in main() via the pointer.\n");
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  What would happen if you changed the `add` function to accept `int sum`
        instead of `int *sum_pointer` and tried to do `sum = result;`? Try it!
        You'll find that the 'sum' variable in main() is never updated, because
        the function only received a copy of its initial (garbage) value.
    2.  Write a function `void getMinMax(int x, int y, int *min, int *max)` that
        takes two integers and "returns" both the minimum and the maximum value
        through pointers. This is a classic example of why pass-by-reference is
        so useful, as a C function can only have one direct return value.
    ================================================================================
*/
```

Chapter 4

Arrays

```c
/*
    ================================================================================
    Tutorial: Arrays - Storing and Accessing Data
    ================================================================================

    This program introduces the concept of an array. An array is a collection
    of elements of the same data type, stored in contiguous memory locations.
    This allows you to manage a list of items using a single variable name.

    This example creates an array to store recipe ingredient amounts, reads
    values into it, and then retrieves a specific ingredient amount based on
    a user-provided ID (index).

    Concepts Covered:
    - Declaring an array with a fixed size.
    - Accessing array elements using an index (e.g., `my_array[0]`).
    - The concept of zero-based indexing.
    - Using a 'for' loop to iterate over an array.
*/

#include <stdio.h>

int main(void) {
    // Array Declaration:
    // This declares an array named 'ingredients' that can hold 10 integers.
    // The elements are accessed with an index from 0 to 9.
    int ingredients[10];
    int read_id = 0; // The index of the ingredient to read back.

    printf("Please enter 10 integer ingredient amounts:\n");

    // Populating the array using a loop:
    // We loop from i = 0 to 9 to read a value for each of the 10 array slots.
    for (int i = 0; i < 10; i++) {
        printf("Enter amount for ingredient #%d: ", i);
        // We read the value into the element at the current index 'i'.
        scanf("%d", &ingredients[i]);
    }

    printf("\nAll ingredients entered.\n");
    printf("Which ingredient ID (0-9) would you like to retrieve? ");
    scanf("%d", &read_id);

    // Accessing an array element:
    // We use the user-provided 'read_id' as the index to retrieve and print
    // a specific value from the array.
    // IMPORTANT: C does not check if the index is valid. If the user enters 15,
    // the program will try to access memory outside the array's bounds, which
    // leads to "undefined behavior" (it might crash, or it might print garbage).
    if (read_id >= 0 && read_id < 10) {
        printf("The amount for ingredient ID %d is %d.\n", read_id, ingredients[read_id]);
    } else {
        printf("Error: Invalid ID. Please enter a value between 0 and 9.\n");
    }

    return 0;
}

/*
```

```
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  Modify the program to calculate and print the sum of all the ingredient
        amounts after they have been entered. You'll need another 'for' loop.
    2.  Instead of hard-coding the size of the array to 10, use a constant
        (e.g., `#define RECIPE_SIZE 10`). This makes it much easier to change
        the number of ingredients later, as you only have to modify one line.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Arrays - A Practical Example with Doubles
    ================================================================================

    This program demonstrates a practical use of arrays. It performs two "passes"
    over the data. First, it reads a set of weights into an array of doubles and
    calculates their sum. Second, after calculating the average, it iterates over
    the array again to show how much each individual weight needs to be adjusted
    to match the average.

    This two-pass approach is very common in data processing and highlights why
    we need arrays: we need to store all the values first before we can calculate
    the average and compare each value against it.

    Concepts Covered:
    - Declaring an array of `double`s.
    - Using a variable to define the number of elements to use in an array.
    - Performing multiple passes over array data.
*/

#include <stdio.h>

int main(void) {
    // We declare an array with a maximum capacity of 50.
    // This means the program can handle *up to* 50 cars, but not more.
    double car_weights[50];
    int num_cars = 0;
    double total_weight = 0.0;
    double average_weight = 0.0;

    printf("How many cars are there (up to 50)? ");
    scanf("%d", &num_cars);

    // It's good practice to check if the user's input is within our array's capacity.
    if (num_cars > 50 || num_cars <= 0) {
        printf("Error: Invalid number of cars. Please enter a number between 1 and 50.\n");
        return 1; // Exit the program with an error code.
    }

    // First Pass: Read weights and calculate the sum.
    printf("Enter the weight for each of the %d cars:\n", num_cars);
    for (int i = 0; i < num_cars; i++) {
        printf("Car #%d weight: ", i + 1);
        scanf("%lf", &car_weights[i]);
        total_weight += car_weights[i];
    }

    // Calculate the average.
    average_weight = total_weight / num_cars;
    printf("\nThe average weight of the cars is: %.2lf\n", average_weight);

    // Second Pass: Calculate and print the difference from the average for each car.
    printf("Weight adjustment needed for each car to reach the average:\n");
    for (int i = 0; i < num_cars; i++) {
        double difference = average_weight - car_weights[i];
        // The '%.1lf' format specifier prints the double with 1 decimal place.
        // A positive number means the car is lighter than average and needs weight added.
        // A negative number means the car is heavier and needs weight removed.
        printf("Car #%d: %.1lf\n", i + 1, difference);
    }

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  Can you find the car with the heaviest weight and the car with the
        lightest weight? You would do this by iterating through the array after
        it's populated, keeping track of the `max_weight_so_far` and
        `min_weight_so_far`.
    2.  What if you wanted to store the name (a string) of each car along with its
        weight? This would require a more advanced data structure, like two
        separate arrays (one for weights, one for names) or an array of `structs`,
        which we will cover later in the tutorial.
    ================================================================================
*/
```

```c
/*
    ===============================================================================
    Tutorial: Arrays - Finding Maximum and Minimum Values
    ===============================================================================

    A very common task when working with data is to find the largest or smallest
    value in a set. This program shows the standard algorithm for finding the
    maximum and minimum values in an array.

    This file merges the logic from two previous examples (`03_array_max_min.c`
    and `04_array_max_age.c`).

    Concepts Covered:
    - The algorithm for finding the maximum value in an array.
    - The algorithm for finding the minimum value in an array.
    - The importance of correct initialization when searching for min/max.
*/

#include <stdio.h>

int main(void) {
    int ages[10];

    printf("Please enter the ages of 10 people:\n");
    for (int i = 0; i < 10; i++) {
        scanf("%d", &ages[i]);
    }

    // --- Finding the Maximum and Minimum ---

    // The key to finding the min or max is to have a variable that holds the
    // "biggest value seen so far" or "smallest value seen so far".

    // A ROBUST way to initialize is to set both max and min to the *first*
    // element of the array. This guarantees they hold a real value from the
    // data set to start with.
    int age_max = ages[0];
    int age_min = ages[0];

    // The original code for finding the minimum was `int ageMin = 200;`. This is
    // a "magic number" and is a bug! It would fail if all the ages entered
    // were greater than 200. Initializing with the first element is safer.

    // Now, we loop through the *rest* of the array (starting from the second element, index 1).
    for (int i = 1; i < 10; i++) {
        // Check for a new maximum
        if (ages[i] > age_max) {
            // We found a new biggest age, so we update our variable.
            age_max = ages[i];
        }

        // Check for a new minimum
        if (ages[i] < age_min) {
            // We found a new smallest age, so we update our variable.
            age_min = ages[i];
        }
    }

    printf("\nThe maximum age is %d.\n", age_max);
    printf("The minimum age is %d.\n", age_min);

    // --- Using the Results ---
    printf("\nAge differences with the eldest person:\n");
    for (int i = 0; i < 10; i++) {
        printf("%d:%d ", ages[i], age_max - ages[i]);
    }

    printf("\n\nAge differences with the youngest person:\n");
    for (int i = 0; i < 10; i++) {
        printf("%d:%d ", ages[i], ages[i] - age_min);
    }
    printf("\n");

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  What if you wanted to find the *index* of the oldest person, not just
        their age? You would need another variable, `max_age_index`, that you
        update inside the `if (ages[i] > age_max)` block.
    2.  This program uses a fixed-size array of 10. Can you modify it to first
        ask the user how many ages they want to enter, and then process that
        many? (You'll still need to declare an array with a fixed *maximum* size).
    ================================================================================
*/
```

```c
/*
    ==============================================================================
    Tutorial: Arrays - Passing Arrays to Functions
    ==============================================================================

    This is a very important topic. When you pass an array to a function in C,
    you are not passing a copy of the entire array. Instead, you are passing a
    *pointer* to the first element of the array.

    This has a huge implication: any modifications the function makes to the
    array elements will affect the original array in the calling function!
    This is different from passing a simple variable (like an int), where the
    function only gets a copy.

    This file merges concepts from three different examples to provide a
    comprehensive overview.

    Concepts Covered:
    - How arrays are passed to functions as pointers.
    - How functions can modify the original array's data.
    - The equivalence of pointer notation (`*(ptr + i)`) and array notation (`ptr[i]`).
    - The need to pass the array's size as a separate argument.
*/

#include <stdio.h>

// Function Prototypes
// When a function expects an array, you can declare the parameter as a pointer
// (e.g., `int *ptr`) or with empty array brackets (e.g., `int arr[]`). Both are
// treated identically by the compiler: they are both pointers.
void printArray(int arr[], int size);
void squareArray(int arr[], int size);
void resetArray(int *ptr, int size); // Showing the pointer syntax is the same

int main(void) {
    // We can initialize an array with values when we declare it.
    // The compiler will automatically determine the size.
    int array[] = {6, 2, -4, 8, 5, 1};
    int size = 6;

    printf("Original array: ");
    printArray(array, size);

    // This function will modify the original array.
    squareArray(array, size);
    printf("After squaring: ");
    printArray(array, size);

    // This function also modifies the original array.
    resetArray(array, size);
    printf("After resetting:  ");
    printArray(array, size);

    return 0;
}

/*
    Function: squareArray
    Purpose: Squares each element of an integer array.
```

```c
*/
void squareArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        // This modification directly changes the 'array' in main().
        arr[i] = arr[i] * arr[i];
    }
}

/*
    Function: resetArray
    Purpose: Resets the first 3 elements of an array to 0.
    This demonstrates using pointer notation to achieve the same result.
*/
void resetArray(int *ptr, int size) {
    // In C, `ptr[i]` is just "syntactic sugar" for `*(ptr + i)`.
    // The following lines show the equivalence. Most programmers use the
    // array notation `ptr[i]` because it's easier to read.
    if (size >= 1) {
        *(ptr + 0) = 0; // Same as ptr[0] = 0
    }
    if (size >= 2) {
        *(ptr + 1) = 0; // Same as ptr[1] = 0
    }
    if (size >= 3) {
        ptr[2] = 0;     // Same as *(ptr + 2) = 0
    }
}


/*
    Function: printArray
    Purpose: Prints all elements of an integer array.
    Note: A function that receives an array has no way of knowing its size on
    its own. This is why we MUST pass the size as a separate argument.
*/
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  One of the original examples had a function that found the maximum value
        in an array and another function that used that result to modify the array.
        Can you write a function `int findMax(int arr[], int size)` that returns
        the largest value, and then call it from `main`?
    2.  What happens if you declare a function parameter with `const`, like
        `void printArray(const int arr[], int size)`? This is a promise to the
        compiler that the function will *not* modify the array. If you then try
        to write `arr[i] = 0;` inside that function, the compiler will give you
        an error. This is a very good practice for functions that should only
        read from an array, not write to it.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Arrays - Reversing an Array
    ================================================================================

    This program demonstrates a classic and efficient algorithm for reversing
    an array "in-place" (meaning, without creating a second, temporary array).

    The core idea is to swap the first element with the last, the second element
    with the second-to-last, and so on, stopping at the middle of the array.

    Concepts Covered:
    - In-place modification of an array.
    - The "two-pointer" or "symmetric swap" algorithm for reversal.
    - Using a temporary variable for swapping values.
*/

#include <stdio.h>

// Function Prototypes
void readArray(int arr[], int size);
void reverseArray(int arr[], int size);
void printArray(int arr[], int size);

int main(void) {
    const int SIZE = 6; // Using a constant for the size is good practice.
    int array[SIZE];

    printf("Please enter %d integers:\n", SIZE);
    readArray(array, SIZE);

    printf("\nOriginal array: ");
    printArray(array, SIZE);

    reverseArray(array, SIZE);

    printf("Reversed array: ");
    printArray(array, SIZE);

    return 0;
}

/*
    Function: reverseArray
    Purpose: Reverses the elements of an array in-place.
*/
void reverseArray(int arr[], int size) {
    int temp; // A temporary variable is essential for swapping.

    // The loop only needs to go to the middle of the array.
    // If size is 6, we loop from i=0 to 2.
    // i=0 swaps with 5.
    // i=1 swaps with 4.
    // i=2 swaps with 3.
    // If we went all the way to size-1, we would just swap them back!
    for (int i = 0; i < size / 2; i++) {
        // The index of the element symmetric to `i` is `size - 1 - i`.
        int symmetric_index = size - 1 - i;
        // The three steps for a classic swap:
        // 1. Store the value of the first element in a temporary variable.
        temp = arr[i];
        // 2. Overwrite the first element with the value of the second.
        arr[i] = arr[symmetric_index];
        // 3. Overwrite the second element with the value from the temp variable.
        arr[symmetric_index] = temp;
    }
}

/*
    Function: printArray
    Purpose: Prints all elements of an integer array.
*/
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

/*
    Function: readArray
    Purpose: Reads integers from the user to populate an array.
*/
void readArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  What happens if the array has an odd number of elements (e.g., size 7)?
        Trace the loop with `size / 2`. Integer division means `7 / 2` is 3. The
        loop will run for i=0, 1, 2. The middle element (at index 3) is never
        touched, which is correct! It doesn't need to be swapped with anything.
    2.  Can you write a function `void swap(int *a, int *b)` that takes pointers
        to two integers and swaps their values? Then, you could call this
        `swap(&arr[i], &arr[symmetric_index])` inside your `reverseArray`
        function to make the code even cleaner. This is an excellent pointer exercise.
    ================================================================================
*/
```

Chapter 5

Pointers

```c
/*
    ================================================================================
    Tutorial: Pointers - The Absolute Basics
    ================================================================================

    Welcome to Pointers! This is one of the most powerful and defining features
    of the C language. It can seem intimidating, but let's break it down.

    A pointer is simply a variable that holds a memory address as its value.
    Instead of storing a number like `42` or a character like 'r', it stores
    the location where another variable lives in the computer's memory.

    This first example shows how to declare a pointer and how to get the
    memory address of a regular variable.

    Concepts Covered:
    - Declaring a pointer variable (e.g., `int *my_pointer;`).
    - The "address-of" operator (&) to get a variable's memory location.
    - The '%p' format specifier for printing memory addresses.
*/

#include <stdio.h>

int main() {
    // There are two `#include <stdio.h>` lines in the original file.
    // While harmless, it's redundant. We only need one.

    // Declare some regular variables of different types.
    int a = 42;
    double d = 58.394;
    char c = 'r';

    // --- Declaring and Assigning Pointers ---

    // This declares a variable named 'addressOfA' that is a "pointer to an int".
    // The asterisk (*) indicates that it's a pointer.
    // We then use the address-of operator (&) to get the memory address of 'a'
    // and we assign that address as the value of our pointer.
    int *addressOfA = &a;

    // We do the same for the other data types.
    double *addressOfD = &d;
    char *addressOfC = &c;


    // --- Printing Memory Addresses ---

    // The `%p` format specifier is used with `printf` to print memory addresses
    // in a standard hexadecimal format.
    printf("The value of 'a' is %d, and it is stored at memory address: %p\n", a, addressOfA);
    printf("The value of 'd' is %lf, and it is stored at memory address: %p\n", d, addressOfD);
    printf("The value of 'c' is '%c', and it is stored at memory address: %p\n", c, addressOfC);

    // You can also print the address directly without using a pointer variable:
    printf("\nWe can also print the address of 'a' directly: %p\n", &a);

    return 0;
}
```

```
/*
    ===============================================================================
    Further Exploration:
    ===============================================================================
    1.  What is the value *of the pointer variable itself*? It's a memory address.
        What is the *address of the pointer variable*? A pointer is a variable,
        so it has its own memory address too! You can find it with `&addressOfA`.
        Try printing it out!
    2.  What is the size of a pointer? Try `printf("%zu", sizeof(addressOfA));`.
        Do this for `addressOfD` and `addressOfC` as well. You will likely find
        that all pointers have the same size on your system (typically 4 or 8
        bytes), because they all just store a memory address, regardless of the
        type of data they point to.
    ===============================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Pointers - Dereferencing ("Going to the Address")
    ================================================================================

    If a pointer holds a memory address, how do we see what's *at* that address?
    This action is called "dereferencing", and it's done with the same asterisk (*)
    we use to declare a pointer.

    When used on an existing pointer variable, `*my_pointer` means
    "go to the address stored in `my_pointer` and get the value there".

    This is the key to reading and *writing* data through pointers.

    This file merges concepts from four different examples.

    Concepts Covered:
    - The dereference operator (*) to read a value.
    - Using a dereferenced pointer to modify a value.
    - A practical example: passing a pointer to a function to modify a variable.
    - The relationship between pointers and arrays.
*/

#include <stdio.h>

// Function prototype that takes a pointer as an argument.
void timesTwo(int *num_ptr);

int main(void) {
    // --- Part 1: Basic Dereferencing ---

    int i = 42;
    int *i_ptr = &i; // i_ptr now holds the address of i.

    double a = 3.14;
    double *a_ptr = &a;

    // To read the value *at* the address, we dereference the pointer.
    printf("The variable 'i' lives at address %p.\n", i_ptr);
    printf("The value stored at that address is %d.\n\n", *i_ptr); // Dereference!

    printf("The variable 'a' lives at address %p.\n", a_ptr);
    printf("The value stored at that address is %lf.\n\n", *a_ptr);

    // --- Part 2: Modifying Values via Dereferencing ---

    printf("The original value of 'i' is %d.\n", i);
    printf("Now, we will modify 'i' using the pointer...\n");

    // We are not changing the pointer `i_ptr`, but the value at the address it points to.
    *i_ptr = 50;

    // If we now check the original variable 'i', we see it has changed!
    printf("The new value of 'i' is %d.\n\n", i);


    // --- Part 3: Practical Use - Modifying a Value in a Function ---

    int n = 10;
    printf("The value of 'n' before the function call is %d.\n", n);

    // We pass the ADDRESS of 'n' to the function.
    timesTwo(&n);

    // Because the function had the address, it could change 'n' directly.
    printf("The value of 'n' after the function call is %d.\n\n", n);


    // --- Part 4: The Relationship Between Pointers and Arrays ---

    int my_array[] = {100, 200, 300};
    int *array_ptr = my_array; // Notice: no '&' needed for an array!

    // The name of an array, when used by itself, decays into a pointer
    // to its first element. So, `my_array` is equivalent to `&my_array[0]`.

    printf("The first element is %d.\n", *array_ptr); // Prints 100

    // We can use pointer arithmetic to access other elements.
    // `*(array_ptr + 1)` means "go to the address of the first element, move
    // forward by the size of one int, and get the value there".
    printf("The second element is %d.\n", *(array_ptr + 1)); // Prints 200

    // This is why array access `my_array[1]` is equivalent to `*(my_array + 1)`.
    // The array bracket notation is just more convenient "syntactic sugar".

    return 0;
}

/*
    Function: timesTwo
    Purpose: Takes a pointer to an integer, and doubles the value stored at
             that integer's address.
*/
void timesTwo(int *num_ptr) {
    printf(" -> Inside function: The received value is %d.\n", *num_ptr);
    // Here, we dereference the pointer to get the value, multiply it by 2,
    // and then use the pointer again to store the new value back in the
    // original memory location.
    *num_ptr = *num_ptr * 2;
    printf(" -> Inside function: The value has been doubled to %d.\n", *num_ptr);
}
```

```c
/*
    ================================================================================
    Tutorial: Pointers - Modifying Variables via Functions
    ================================================================================

    This example reinforces a key use case for pointers: allowing a function to
    modify a variable from its calling scope. Here, we pass a pointer to an 'age'
    variable to a function that changes the age based on some conditions.

    This is a very common pattern in C programming.

    Concepts Covered:
    - Passing a pointer to a function.
    - Dereferencing the pointer within the function to read the original value.
    - Dereferencing the pointer to write a new value back to the original variable.
*/

#include <stdio.h>

// Function prototype: takes a pointer to an integer.
void applyElixir(int *age_ptr);

int main(void) {
    int age;
    // We can use a pointer to read input with scanf.
    // `&age` gives the address of 'age', which is what scanf needs.
    // `age_ptr` also holds the address of 'age'. So they are interchangeable here.
    int *age_ptr = &age;

    printf("Enter your current age: ");
    scanf("%d", age_ptr); // Same as `scanf("%d", &age);`

    printf("Your current age is %d.\n", age);

    // Call the function, passing the address of the 'age' variable.
    applyElixir(age_ptr); // Same as `applyElixir(&age);`

    // The 'age' variable in main has now been changed by the function.
    printf("You drank the elixir... your new age is %d!\n", age);

    return 0;
}

/*
    Function: applyElixir
    Purpose: Modifies an age value based on a set of rules.
    Parameters:
        - age_ptr: A pointer to the integer 'age' variable from main().
*/
void applyElixir(int *age_ptr) {
    // We dereference the pointer `*age_ptr` to get the current age value
    // to use in our comparison.
    if (*age_ptr > 21) {
        // If the age is over 21, subtract 10.
        // We dereference the pointer again to assign the new value.
        *age_ptr = *age_ptr - 10;
    } else {
        // Otherwise, double the age.
        *age_ptr = *age_ptr * 2;
    }
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  The `applyElixir` function modifies the age "in-place". Can you rewrite it
        so that it doesn't take a pointer, but instead `return`s the new age?
        `int calculateNewAge(int current_age);`
        You would then call it from main like this: `age = calculateNewAge(age);`.
        Both are valid patterns, and choosing between them is a design decision.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Pointers - Pointer Arithmetic
    ================================================================================

    Pointer arithmetic is a powerful feature that allows you to move a pointer
    forwards or backwards in memory. This is especially useful for navigating
    arrays.

    When you add an integer `n` to a pointer, you are not adding `n` bytes to
    the address. Instead, you are moving the pointer forward by `n` *elements*
    of the type it points to. The compiler automatically knows the size of the
    data type (e.g., 4 bytes for an `int`) and does the correct calculation.

    `pointer + n` is equivalent to `address_of_pointer + n * sizeof(type)`

    Concepts Covered:
    - The equivalence of array names and pointers to the first element.
    - Using `*(ptr + n)` to access the nth element.
    - Incrementing a pointer (`ptr++`) to move to the next element.
    - Modifying a pointer by a larger value (`ptr += 3`).
*/

#include <stdio.h>

void printArray(int arr[], int size);

int main(void) {
    int array[] = {10, 20, 30, 40, 50, 60};
    int size = 6;
    int *ptr = array; // ptr now points to the first element (array[0]).

    printf("Original array: ");
    printArray(array, size);
    printf("Pointer 'ptr' currently points to address %p, which holds the value %d.\n\n", ptr,
    ↪  *ptr);

    // --- Accessing elements using pointer arithmetic ---
    // `*(ptr + 2)` accesses the element at index 2 (value 30).
    printf("Accessing element at index 2 using *(ptr + 2): %d\n", *(ptr + 2));

    // --- Modifying elements using pointer arithmetic ---
    printf("Modifying element at index 2 to be 33...\n");
    *(ptr + 2) = 33;
    printf("Array after modification: ");
    printArray(array, size);
    printf("\n");

    // --- Moving the pointer itself ---
    // `ptr++` moves the pointer to the next element in the array.
    printf("Moving pointer with ptr++...\n");
    ptr++; // ptr now points to the second element (array[1]).
    printf("Pointer 'ptr' now points to address %p, which holds the value %d.\n", ptr, *ptr);
    printf("Modifying this element to be 22...\n");
    *ptr = 22; // This changes array[1].
    printf("Array after modification: ");
    printArray(array, size);
    printf("\n");
    // We can also move the pointer by more than one step.
    printf("Moving pointer with ptr += 3...\n");
    ptr += 3; // ptr was at index 1, now it moves to index 1 + 3 = 4.
    printf("Pointer 'ptr' now points to address %p, which holds the value %d.\n", ptr, *ptr);
    printf("Modifying this element to be 55...\n");
    *ptr = 55; // This changes array[4].
    printf("Array after modification: ");
    printArray(array, size);
    printf("\n");

    return 0;
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  What happens if you try to dereference a pointer that has moved past the
        end of the array (e.g., after `ptr += 3;`, add another `ptr++` and then
        try to print `*ptr`)? This is a dangerous operation that reads from
        memory you don't own, leading to undefined behavior.
    2.  You can also use the subtraction operator. If `ptr` points to `array[4]`,
        what do you think `*(ptr - 2)` would give you?
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Pointers - The Classic Swap Function
    ================================================================================

    This program demonstrates one of the most essential and classic uses for
    pointers: a function that swaps the values of two variables.

    If you were to write a `swap` function that took regular integers (`int a, int b`)
    instead of pointers, it would fail. The function would only receive *copies*
    of the original values, and swapping those copies would have no effect on the
    original variables in `main`.

    To make the swap permanent, the function needs to know the *memory addresses*
    of the original variables, which requires pointers.

    Concepts Covered:
    - A canonical use case for pointers.
    - The logic of swapping two values using a temporary variable.
    - Passing the addresses of two different variables to a single function.
*/

#include <stdio.h>

// Function prototype for our swap function.
// It takes two "pointers to int" as arguments.
void swap(int *ptr_a, int *ptr_b);

int main() {
    int a = 9;
    int b = 1;

    printf("Before swap: a = %d, b = %d\n", a, b);

    // We call the swap function, passing the memory addresses of 'a' and 'b'.
    swap(&a, &b);

    printf("After swap:  a = %d, b = %d\n", a, b);

    return 0;
}

/*
    Function: swap
    Purpose: Swaps the integer values stored at two different memory addresses.
    Parameters:
        - ptr_a: A pointer to the first integer variable.
        - ptr_b: A pointer to the second integer variable.
*/
void swap(int *ptr_a, int *ptr_b) {
    int temp; // A temporary variable is essential for a swap.

    // 1. Store the value from the first address (`*ptr_a`) in `temp`.
    temp = *ptr_a;

    // 2. Copy the value from the second address (`*ptr_b`) into the first address.
    *ptr_a = *ptr_b;

    // 3. Copy the value we originally stored in `temp` into the second address.
    *ptr_b = temp;
}

/*
    ===============================================================================
    Further Exploration:
    ===============================================================================
    1.  As a challenge, try to write the incorrect version of swap:
        `void failedSwap(int a, int b)`. Put the swap logic inside and call it
        from main. You will see that it does not work, which is the best way to
        understand why pointers are needed here.
    2.  We used this swap logic in the `04_Arrays/06_reversing_an_array.c`
        example. Can you go back to that file and modify it to use this new,
        reusable `swap()` function?
    ===============================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Pointers - Pointers to Pointers (Double Pointers)
    ================================================================================

    This is an advanced topic, so let's take it slow.
    - A normal pointer (like `int *`) stores the address of a variable.
    - A pointer-to-a-pointer (like `int **`) stores the address of *another pointer*.

    Why would you need this?
    1.  To create an array of pointers (as shown in this example). This is very
        common for storing an array of strings.
    2.  For a function that needs to change where a pointer points (i.e., change
        the address stored in the pointer variable itself).

    This example creates an "array of pointers", where each element of the array
    is a pointer to another array.

    Concepts Covered:
    - Declaring a pointer-to-a-pointer (`type **`).
    - Creating an array of pointers.
    - The logic of double dereferencing.
*/

#include <stdio.h>

// This function takes a pointer-to-a-pointer-to-a-short.
// `t` is an address of a pointer.
void setToZero(short **t);
void printArrays(short **t);

int main() {
    short a[] = {1245, 1924, 234};
    short b[] = {24, 256};

    // `t` is an array of pointers. Each element in `t` is a `short *`.
    // t[0] is a pointer that holds the address of the first element of `a`.
    // t[1] is a pointer that holds the address of the first element of `b`.
    short *t[2] = {a, b};

    printf("--- Before ---\n");
    printArrays(t);

    // When we pass `t` to a function, the array name `t` decays into a pointer
    // to its first element. The first element of `t` is a `short *`.
    // Therefore, a pointer to that element is a `short **`.
    setToZero(t);

    printf("\n--- After ---\n");
    printArrays(t);

    return 0;
}

/*
    Function: setToZero
    Purpose: Sets the elements of the arrays pointed to by the pointer array to zero.
    Parameters:
        - t: A pointer to the first element of the array of pointers from main.
```

```
*/
void setToZero(short **t) {
    // Let's break down the complex syntax: `*(*t) = 0;`
    //
    // 1. `t`: A pointer to a `short *`. Its value is the address of `t[0]` from main.
    // 2. `*t`: We dereference `t` once. This gives us the value *at* `t[0]`, which
    //          is the pointer to array `a`. So, `*t` is the same as `a`.
    // 3. `*(*t)`: We dereference the result again. Since `*t` is `a`, `*(*t)` is the
    //            same as `*a`, which refers to the first element of `a`.
    // So, `*(*t) = 0;` is equivalent to `a[0] = 0;`.

    *(*t) = 0; // Sets a[0] to 0.

    // Let's look at `*((*t) + 1) = 0;`
    // 1. `*t` is `a`.
    // 2. `(*t) + 1` is pointer arithmetic, same as `a + 1`, which points to a[1].
    // 3. `*((*t) + 1)` dereferences that, so it's the same as `a[1]`.
    *((*t) + 1) = 0; // Sets a[1] to 0.
    *((*t) + 2) = 0; // Sets a[2] to 0.

    // Now let's look at `*(*(t + 1)) = 0;`
    // 1. `t + 1`: Pointer arithmetic. Moves from pointing at `t[0]` to pointing at `t[1]`.
    // 2. `*(t + 1)`: Dereferences that. This gives us the value *at* `t[1]`, which is the pointer
    ↪  `b`.
    // 3. `*(*(t + 1))`: Dereferences `b`, giving us `b[0]`.
    *(*(t + 1)) = 0;     // Sets b[0] to 0.
    *(*(t + 1) + 1) = 0; // Sets b[1] to 0.
}

// A helper function to see the results.
void printArrays(short **t) {
    // This is just for demonstration; in a real program, we would also need
    // to pass the size of each individual array to do this safely in a loop.
    printf("Content of array 'a': %d, %d, %d\n", t[0][0], t[0][1], t[0][2]);
    printf("Content of array 'b': %d, %d\n", t[1][0], t[1][1]);
}
```

Chapter 6

Strings

```c
/*
    ================================================================================
    Tutorial: Strings - Finding the Longest Word
    ================================================================================

    This program reads a specified number of words from the user and finds the
    length of the longest word among them.

    This is a common string processing task that combines loops, string handling,
    and a max-finding algorithm.

    Concepts Covered:
    - Reading strings from input.
    - Using `strlen()` to get the length of a string.
    - The algorithm for finding a maximum value, applied to string lengths.
*/

#include <stdio.h>
#include <string.h> // We need this header for the strlen() function.

int main(void) {
    int num_words = 0;
    // This buffer will hold each word as we read it.
    // Size 101 can hold a 100-character word plus the null terminator.
    char current_word[101];
    int max_length_so_far = 0;

    printf("How many words will you enter? ");
    scanf("%d", &num_words);

    printf("Please enter %d words:\n", num_words);

    for (int i = 0; i < num_words; i++) {
        scanf("%s", current_word);

        // Use the standard strlen() function to get the length of the word just read.
        int current_length = strlen(current_word);

        // This is the standard max-finding algorithm.
        // If the length of the new word is greater than the max we've seen so far...
        if (current_length > max_length_so_far) {
            // ...then we have a new maximum!
            max_length_so_far = current_length;
        }
    }

    printf("The length of the longest word entered is: %d\n", max_length_so_far);

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  This program only tells you the *length* of the longest word, but not
        what the word itself is. Can you modify it to store the longest word?
        You would need another character array, maybe `char longest_word[101];`,
        and inside the `if` block, you would use the `strcpy()` function (also
        from `<string.h>`) to copy the `current_word` into `longest_word`.
    2.  What if two words have the same, maximum length? The current program
        correctly reports their length. The modification in #1 would only store
        the *first* longest word it finds. How could you change it to store the
        *last* longest word?
    ===============================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Strings - Searching for a Character
    ================================================================================

    This program searches for the first occurrence of the letter 't' (or 'T')
    in a given word. It then reports if the letter was found in the first or
    second half of the word.

    This is a great example of a linear search algorithm applied to a string.

    Concepts Covered:
    - Iterating through a string with a 'for' loop.
    - Character comparison.
    - Handling case-insensitivity with `tolower()`.
    - Using a "flag" variable to track if an item was found.
    - Using `break` to exit a loop early.
*/

#include <stdio.h>
#include <string.h> // For strlen()
#include <ctype.h>  // For tolower()

int main(void) {
    char word[51];
    int found_flag = 0; // A "flag" to track if we've found the letter. 0=false, 1=true.

    printf("Enter a word: ");
    scanf("%s", word);

    int length = strlen(word);

    // We can loop through the string using a 'for' loop, which is often
    // cleaner than a 'while' loop for this kind of task.
    for (int i = 0; i < length; i++) {
        // To handle case-insensitivity, we can convert the current character
        // to lowercase before comparing it. The tolower() function is from <ctype.h>.
        if (tolower(word[i]) == 't') {
            printf("Found 't' at index %d.\n", i);

            // The original logic for determining the "half" was a bit complex.
            // Let's simplify: if the index is less than half the length, it's
            // in the first half.
            // Note on integer division: For length 5, `5 / 2` is 2.
            // Indices 0, 1 are in the first half. Index 2 (the middle) is not.
            if (i < length / 2) {
                printf("Output: 1 (first half)\n");
            } else {
                printf("Output: 2 (second half)\n");
            }

            // We found the letter, so we set our flag to true...
            found_flag = 1;
            // ...and use 'break' to exit the loop immediately. We only care
            // about the *first* occurrence.
            break;
        }
    }
    // After the loop has finished, we check the flag.
    // If the flag is still false, it means the loop completed without ever
    // finding the character.
    if (found_flag == 0) {
        printf("The letter 't' was not found.\n");
        printf("Output: -1\n");
    }

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  The definition of "half" can be tricky. In our version, for a word with
        an odd length like "seven" (length 5), the middle character (at index 2)
        is considered part of the "second half". Is this the behavior you want?
        How would you change the condition to include the middle character in the
        first half?
    2.  Can you modify this program to count *all* occurrences of 't', not just
        the first one? You would need a counter variable and you would remove
        the `break` statement.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Strings - Comparing Strings Alphabetically
    ================================================================================

    This program compares two words entered by the user to determine their
    alphabetical order.

    The original code implements the comparison logic manually. We will keep
    that as a reference but introduce the standard C library function `strcmp()`
    from `<string.h>`, which is the correct and standard way to compare strings.

    Concepts Covered:
    - The logic of lexicographical (alphabetical) comparison.
    - Using the `strcmp()` function to compare strings.
    - Interpreting the return value of `strcmp()`.
*/

#include <stdio.h>
#include <string.h> // For strcmp()

int main(void) {
    char word1[50];
    char word2[50];

    printf("Please enter a word: ");
    scanf("%s", word1);
    printf("And another word: ");
    scanf("%s", word2);

    /*
        Method 1: Manual Comparison (as in the original code)
        This is how comparison works under the hood. The loop finds the first
        character where the two strings differ. The comparison of those two
        characters then determines the alphabetical order of the words.

        int i = 0;
        while (word1[i] != '\0' && word2[i] != '\0' && word1[i] == word2[i]) {
            i++;
        }
        // After the loop, `i` is the index of the first differing character.
        // We can then compare `word1[i]` and `word2[i]`.
    */

    // Method 2: The Standard Library Function `strcmp()` (Preferred)
    // The `strcmp()` function (short for "string compare") does all the
    // manual work for us. It returns:
    //   - A negative number (< 0) if word1 comes before word2.
    //   - A positive number (> 0) if word1 comes after word2.
    //   - Zero (0) if the strings are identical.
    int comparison_result = strcmp(word1, word2);

    printf("\n--- Result ---\n");
    if (comparison_result < 0) {
        printf("\"%s\" comes before \"%s\" alphabetically.\n", word1, word2);
    } else if (comparison_result > 0) {
        printf("\"%s\" comes after \"%s\" alphabetically.\n", word1, word2);
    } else {
        printf("You entered the same word, \"%s\", twice.\n", word1);
    }

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  `strcmp()` is case-sensitive. 'A' comes before 'a'. There is another
        function, often called `stricmp()` or `strcasecmp()`, that performs a
        case-insensitive comparison. Its availability can depend on the system,
        but it's good to know it exists.
    2.  How would you sort an *array* of strings? You would need a sorting
        algorithm (like bubble sort, which we'll see later) and you would use
        `strcmp()` for the comparisons. You would also need to swap the strings
        if they are out of order, which requires the `strcpy()` function. This
        is a classic, challenging C exercise!
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Strings and Arrays - Word Length Frequency Counter
    ================================================================================

    This program reads a number of words and then calculates the frequency of
    each word length. For example, it counts how many 3-letter words, 4-letter
    words, etc., were entered by the user.

    This is a fantastic example of a common programming pattern: using an
    array as a frequency counter or histogram.

    Concepts Covered:
    - Using an array as a frequency counter.
    - The importance of initializing an array to zero.
    - Using one variable's value (word length) as an index for another array.
    - Preventing out-of-bounds array access.
*/

#include <stdio.h>
#include <string.h> // For strlen()

// It's good practice to use constants for array sizes.
#define MAX_WORD_LENGTH 10
#define MAX_WORDS 50

int main(void) {
    int num_words_to_read = 0;
    char current_word[MAX_WORD_LENGTH + 1]; // +1 for the null terminator

    // This is our frequency array. The index will represent the word length.
    // `length_counts[3]` will store the number of 3-letter words.
    // We make it size `MAX_WORD_LENGTH + 1` to safely handle words of length
    // up to MAX_WORD_LENGTH (indices 0 through 10).
    // Initializing with `{0}` sets all elements to zero, which is crucial.
    int length_counts[MAX_WORD_LENGTH + 1] = {0};

    printf("How many words do you want to analyze (up to %d)? ", MAX_WORDS);
    scanf("%d", &num_words_to_read);

    if (num_words_to_read > MAX_WORDS || num_words_to_read <= 0) {
        printf("Invalid number of words.\n");
        return 1;
    }

    printf("Enter %d words (max length %d each):\n", num_words_to_read, MAX_WORD_LENGTH);

    for (int i = 0; i < num_words_to_read; i++) {
        scanf("%s", current_word);

        int current_length = strlen(current_word);
        // The original code printed here, which can be noisy.
        // printf("Read word: \"%s\", length: %d\n", current_word, current_length);

        // --- The Core Logic ---
        // We use the length of the word as an index into our frequency array.
        // But first, we must check if the length is within the bounds of our array!
        if (current_length <= MAX_WORD_LENGTH) {
            // If the word has length `L`, we increment the counter at index `L`.
            length_counts[current_length]++;
        } else {
            printf("Warning: The word \"%s\" is too long and will be ignored.\n", current_word);
        }
    }

    // --- Printing the Results ---
    printf("\n--- Word Length Frequency ---\n");
    // We loop through our frequency array to print the results.
    for (int j = 0; j <= MAX_WORD_LENGTH; j++) {
        // We only print the counts for lengths that we actually saw.
        if (length_counts[j] > 0) {
            printf("There are %d word(s) with %d letter(s).\n", length_counts[j], j);
        }
    }

    return 0;
}
```

```c
/*
    ================================================================================
    Tutorial: Strings - Pointers and String Literals
    ================================================================================

    This program determines the name of a fictional tree based on its height
    and number of leaflets.

    While the filename mentions concatenation, this program's core concept is
    actually about **character pointers** and **string literals**.

    A "string literal" is a sequence of characters enclosed in double quotes,
    like "hello world". When you write this in your code, the compiler stores
    this string in a special, often read-only, part of the program's memory.

    Concepts Covered:
    - Character pointers (`char *`).
    - String literals.
    - Assigning the address of a string literal to a character pointer.
    - Using `if-else if` to implement a set of rules.
*/

#include <stdio.h>

int main(void) {
    int height = 0;
    int leaflets = 0;

    // Here, `tree_name` is a POINTER to a character.
    // We initialize it to point to the beginning of the string literal "uncertain".
    // This doesn't copy the string; it just stores the memory address of the 'u'.
    char *tree_name = "uncertain";

    printf("Enter the tree's height: ");
    scanf("%d", &height);
    printf("Enter the number of leaflets: ");
    scanf("%d", &leaflets);

    // The following `if-else if` chain checks the rules.
    // If a rule matches, we simply change the `tree_name` pointer to point to
    // a *different* string literal. No strings are being copied or modified.
    if (height <= 5 && leaflets >= 8) {
        tree_name = "Tinuviel";
    } else if (height >= 10 && leaflets >= 10) {
        tree_name = "Calaelen";
    } else if (height <= 8 && leaflets <= 5) {
        tree_name = "Falarion";
    } else if (height >= 12 && leaflets <= 7) {
        tree_name = "Dorthonion";
    }
    // If none of the conditions match, `tree_name` keeps its original value, "uncertain".

    // The `%s` format specifier tells printf to go to the address stored in
    // `tree_name` and print characters until it finds a null terminator.
    printf("The tree's name is: %s\n", tree_name);

    return 0;
}
```

```
/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  What is the difference between `char *name = "My Name";` and
        `char name[] = "My Name";`? This is a crucial and classic C question.
        - `char *name` creates a pointer to a read-only string literal. You
          can re-point `name` to something else, but you should NOT try to
          modify the content (e.g., `name[0] = 'm';` is undefined behavior).
        - `char name[]` creates an ARRAY and initializes it by copying the
          string literal into it. The array is writable, so you CAN modify
          its contents. However, you cannot re-point `name` itself to a
          different string.
    ================================================================================
*/
```

Chapter 7

Algorithms

```c
/*
    ================================================================================
    Tutorial: Algorithms - Linear Search
    ================================================================================

    This program demonstrates a fundamental searching algorithm called a
    **Linear Search**.

    A linear search is the most straightforward search strategy. It sequentially
    checks each element of a list until a match is found or the whole list has
    been searched.

    Characteristics of Linear Search:
    - Simplicity: It's very easy to understand and implement.
    - Data Requirement: It works on any list of data, sorted or unsorted.
    - Performance: It can be slow for large lists. In the worst case, if the
      item is the last element or not in the list at all, you have to check
      every single element. This is described as O(n) or "linear" time complexity.

    Concepts Covered:
    - The linear search algorithm.
    - Using a 'while' loop that terminates on one of two conditions.
    - Using a "flag" variable to track search success.
*/

#include <stdio.h>

int main(void) {
    int list[] = {6, -2, 5, 12, 7, 3, 8, 18, -10, 1};
    int n = 10;
    int item_to_find;
    int i = 0;
    int found_flag = 0; // 0 for false, 1 for true

    printf("Here is the list of numbers: ");
    for (int j = 0; j < n; j++) {
        printf("%d ", list[j]);
    }
    printf("\nWhich number are you looking for? ");
    scanf("%d", &item_to_find);

    // The core of the linear search.
    // The `while` loop continues as long as we haven't found the item AND
    // we haven't run out of elements to check (i < n).
    while (found_flag == 0 && i < n) {
        printf("Checking index %d (value %d)...\n", i, list[i]);
        if (item_to_find == list[i]) {
            // We found a match!
            found_flag = 1;
        } else {
            // No match, so we move to the next element.
            i++;
        }
    }

    // After the loop, we check the flag to see if the search was successful.
    if (found_flag == 0) { // Or just `if (!found_flag)`
        printf("%d is not a member of this list.\n", item_to_find);
    } else {
        printf("I found %d at index %d in the list.\n", item_to_find, i);
    }

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  A `for` loop can also be used for a linear search. Can you rewrite this
        program to use a `for` loop instead of a `while` loop? You would still
        need a `found_flag` and a `break` statement to exit the loop early once
        the item is found.
    2.  What if the list contains duplicates? This implementation will always
        find the *first* occurrence. How would you modify it to find the *last*
        occurrence? (Hint: you couldn't exit the loop early).
    3.  The next example will show a "bisection" or "binary" search, which is
        much faster but requires the data to be sorted first.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Algorithms - Binary Search (Bisection Search)
    ================================================================================

    This program demonstrates a much more efficient searching algorithm called a
    **Binary Search** or **Bisection Search**.

    The core idea is to repeatedly divide the search interval in half. If the
    value of the search key is less than the item in the middle of the interval,
    narrow the interval to the lower half. Otherwise, narrow it to the upper half.
    This is continued until the value is found or the interval is empty.

    **CRITICAL REQUIREMENT: Binary search ONLY works on a SORTED list.**

    Characteristics of Binary Search:
    - Efficiency: It is extremely fast. For a list of `n` items, it takes at
      most `log2(n)` comparisons. For a list of 1,000,000 items, a linear search
      might take 1,000,000 steps, while a binary search will take at most 20!
      This is "logarithmic" or O(log n) time complexity.
    - Data Requirement: The data MUST be sorted.

    Concepts Covered:
    - The binary search algorithm.
    - Manipulating interval boundaries (lower and upper bounds).
    - The power of "divide and conquer" algorithms.
*/

#include <stdio.h>

int main(void) {
    // Note that this list is already sorted in ascending order.
    int list[] = {-10, -3, 2, 5, 8, 14, 77, 106, 759, 900};
    int n = 10;
    int item_to_find;
    int found_flag = 0;

    // We need three variables to track our search space:
    int lower_bound = 0;        // The starting index of our search space.
    int upper_bound = n - 1;    // The ending index of our search space.
    int midpoint;               // The calculated middle index.

    printf("Which number are you looking for? ");
    scanf("%d", &item_to_find);

    // The loop continues as long as we haven't found the item AND our search
    // space is still valid (the lower bound hasn't crossed the upper bound).
    while (found_flag == 0 && lower_bound <= upper_bound) {
        // Calculate the middle index of the current search space.
        midpoint = lower_bound + (upper_bound - lower_bound) / 2; // Avoids overflow for huge
        ↪  arrays
        printf("Searching between index %d and %d... Midpoint is %d (value %d)\n", lower_bound,
        ↪  upper_bound, midpoint, list[midpoint]);

        if (item_to_find == list[midpoint]) {
            // We found it!
            found_flag = 1;
        } else if (item_to_find < list[midpoint]) {
            // The item must be in the lower half.
            // We discard the top half by moving our upper bound.
            printf(" -> Item is smaller, tossing the top half.\n");
            upper_bound = midpoint - 1;
        } else { // item_to_find > list[midpoint]
            // The item must be in the upper half.
            // We discard the bottom half by moving our lower bound.
            printf(" -> Item is larger, tossing the bottom half.\n");
            lower_bound = midpoint + 1;
        }
    }

    if (found_flag == 0) {
        printf("Number %d was not found in the array.\n", item_to_find);
    } else {
        // Note: `midpoint` will hold the correct index if the item was found.
        printf("Number %d was found at index %d in the array.\n", item_to_find, midpoint);
    }

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  Try searching for a number that isn't in the list (e.g., 6) and watch how
        the search space (lower_bound and upper_bound) shrinks until it becomes
        invalid (`lower_bound > upper_bound`), causing the loop to terminate.
    2.  What happens if the list is not sorted? Try changing the order of the
        elements in the `list` array and see how the algorithm fails. This will
        prove why the sorted requirement is so important.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Algorithms - Bubble Sort
    ================================================================================

    This program demonstrates a simple but inefficient sorting algorithm called
    **Bubble Sort**.

    The core idea is to repeatedly step through the list, compare adjacent
    elements, and swap them if they are in the wrong order. This process is
    repeated until the list is sorted. The smaller elements "bubble" to the
    top of the list.

    Characteristics of Bubble Sort:
    - Simplicity: It is one of the simplest sorting algorithms to understand.
    - Performance: It is very slow for anything other than small or mostly-sorted
      lists. Its time complexity is O(n^2) or "quadratic" time. This means if
      you double the size of the list, the sorting time roughly quadruples.

    Concepts Covered:
    - The bubble sort algorithm.
    - Using nested loops for sorting.
    - A simple in-place swap.
    - An optimized version of bubble sort.
*/

#include <stdio.h>

void printArray(int arr[], int size);

int main(void) {
    int list[] = {759, 14, 2, 900, 106, 77, -10, 8, -3, 5};
    int n = 10;
    int temp_swap;

    printf("Unsorted list: \n");
    printArray(list, n);
    printf("\n");

    // The outer loop (`j`) controls how many passes we make over the array.
    // After the first pass, the largest element is guaranteed to be at the end.
    // After the second pass, the second-largest is in place, and so on.
    // We need n-1 passes in the worst case.
    for (int j = 0; j < n - 1; j++) {
        // --- Optimization ---
        // We can add a flag to check if any swaps were made during a pass.
        // If a full pass completes with no swaps, the array is already sorted,
        // and we can exit early.
        int swapped = 0;

        // The inner loop (`i`) does the actual comparison and swapping.
        // It "bubbles" the largest unsorted element towards the end of the array.
        // The `n - 1 - j` is a further optimization: we don't need to re-check
        // the elements that are already sorted at the end of the array.
        for (int i = 0; i < n - 1 - j; i++) {
            if (list[i] > list[i + 1]) {
                // The elements are in the wrong order, so we swap them.
                temp_swap = list[i];
                list[i] = list[i + 1];
                list[i + 1] = temp_swap;
                swapped = 1; // Set the flag because we made a swap.
            }
        }

        printf("After pass %d: ", j + 1);
        printArray(list, n);

        // If the 'swapped' flag is still 0, it means the list is sorted.
        if (swapped == 0) {
            printf("Array is sorted! Exiting early.\n");
            break; // Exit the outer loop.
        }
    }

    printf("\nFinal sorted list: \n");
    printArray(list, n);

    return 0;
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```c
/*
    ================================================================================
    Tutorial: Algorithms - Finding the Minimum Value (Streaming)
    ================================================================================

    This program finds the smallest integer from a list of numbers provided by
    the user.

    The original code read all numbers into an array first. However, to find the
    minimum value, you don't actually need to store the entire list! You can
    process the numbers as they arrive, keeping track of the smallest one you've
    seen so far. This "streaming" approach uses much less memory and is more
    efficient if you don't need the full list for other purposes.

    We will refactor the code to use this more efficient, streaming approach.

    Concepts Covered:
    - The "streaming" approach to finding a minimum value.
    - The importance of correct initialization.
    - Variable-Length Arrays (VLAs) and why they should often be avoided.
*/

#include <stdio.h>
#include <limits.h> // For INT_MAX

// This helper function is fine. It clearly returns the smaller of two integers.
int min(int a, int b);

int main(void) {
    int num_to_read = 0;
    int current_num = 0;
    // We need a variable to store the minimum value found so far.
    // A great way to initialize it is with the largest possible integer value.
    // This guarantees that the very first number the user enters will be smaller
    // and will become the first "real" minimum.
    // INT_MAX is a constant from the <limits.h> library.
    int min_so_far = INT_MAX;

    printf("How many integers do you want to compare? ");
    scanf("%d", &num_to_read);

    if (num_to_read <= 0) {
        printf("Nothing to compare.\n");
        return 0;
    }

    printf("Enter %d integers:\n", num_to_read);

    for (int i = 0; i < num_to_read; i++) {
        scanf("%d", &current_num);
        // Compare the newly entered number with our running minimum.
        // Update the minimum if the new number is smaller.
        min_so_far = min(min_so_far, current_num);
    }

    printf("The smallest integer entered was: %d\n", min_so_far);

    return 0;
}
```

```c
/*
    A Note on Variable-Length Arrays (VLAs):
    The original code used `int array[len];` where `len` was a variable read from
    the user. This is called a Variable-Length Array. While some C compilers
    support this, it is NOT part of the modern C standard (C11/C18) and can be
    risky. If a user enters a very large number for `len`, it can cause a "stack
    overflow" and crash your program. For this problem, storing the array was
    unnecessary, so the streaming approach is much better and safer.
*/

int min(int a, int b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

```c
/*
    ================================================================================
    Tutorial: Algorithms - Counting Duplicate Characters
    ================================================================================

    This program counts the number of unique characters that are repeated in a word.
    For example, in the word "erroneousnesses", the letters 'e', 'r', 'o', and 's'
    are all repeated, so the program should output 4.

    The algorithm used here is quite clever and demonstrates a powerful technique:
    1.  **Sort:** The characters of the word are sorted alphabetically. This has
        the effect of grouping all identical letters together.
        "erroneousnesses" -> "eeeennoorrssssu"
    2.  **Count Groups:** Iterate through the sorted string and count the number
        of times a new group of duplicate letters begins.

    Concepts Covered:
    - Combining sorting and searching to solve a problem.
    - Using Bubble Sort on a character array.
    - A custom algorithm to count groups of duplicates in a sorted list.
*/

#include <stdio.h>
#include <string.h> // For strlen()

int main(void) {
    char word[51];
    char temp_swap;
    int num_duplicate_groups = 0;

    printf("Enter a word to find the number of repeated character types: ");
    scanf("%s", word);

    int n = strlen(word);

    // --- Step 1: Sort the String using Bubble Sort ---
    // This arranges the word so all identical letters are adjacent.
    for (int j = 0; j < n - 1; j++) {
        for (int i = 0; i < n - 1 - j; i++) {
            if (word[i] > word[i + 1]) {
                temp_swap = word[i];
                word[i] = word[i + 1];
                word[i + 1] = temp_swap;
            }
        }
    }

    printf("Sorted word: %s\n", word);

    // --- Step 2: Count the Groups of Duplicates ---
    // We iterate up to n-1 so we can always check `word[i+1]`.
    for (int i = 0; i < n - 1; i++) {
        // If the current character is the same as the next one, we've found
        // the beginning of a group of duplicates.
        if (word[i] == word[i + 1]) {
            num_duplicate_groups++;

            // This is a crucial inner loop. Once we've counted a group (like the
            // first 'e' in 'eeeee...'), we need to skip past all the other
            // identical characters in that same group so we don't count it again.
            while (i < n - 1 && word[i] == word[i + 1]) {
                i++;
            }
        }
    }

    printf("Number of unique characters that are repeated: %d\n", num_duplicate_groups);

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  This approach is clever but not the most efficient because Bubble Sort is
        slow. A different approach would be to use a frequency array, like we
        saw in the `04_word_lengths_frequency.c` example. You could have an
        array of size 26 (for the English alphabet) and increment the count for
        each letter you see. Afterwards, you would loop through the frequency
        array and count how many elements have a value greater than 1.
    2.  Try that alternative approach! It will be much faster for long strings.
    ================================================================================
*/
```

Chapter 8

Recursion

```c
/*
    ================================================================================
    Tutorial: Recursion - Factorial Example
    ================================================================================

    This program calculates the factorial of a number, and it serves as a classic
    introduction to **Recursion**.

    A recursive function is a function that calls itself. To prevent it from
    calling itself forever, it must have two parts:
    1.  A **Base Case**: A simple condition that stops the recursion. For factorial,
        the base case is `0! = 1`.
    2.  A **Recursive Step**: The part of the function that calls itself, but with
        a "smaller" or "simpler" input that moves it closer to the base case.
        For factorial, this is `n * factorial(n-1)`.

    This file shows both the recursive and the more common iterative (loop-based)
    solutions to demonstrate the differences.

    Concepts Covered:
    - The definition of recursion.
    - Base cases and recursive steps.
    - Comparing a recursive solution to an iterative one.
*/

#include <stdio.h>

// Function prototypes
long long factorial_iterative(int n);
long long factorial_recursive(int n);

int main(void) {
    int n;
    printf("Please enter a non-negative integer: ");
    scanf("%d", &n);

    if (n < 0) {
        printf("%d is negative! Aborting...\n", n);
    } else {
        // We use `%lld` to print a `long long` integer, which is needed because
        // factorials grow extremely quickly.
        printf("Iterative solution: %d! = %lld\n", n, factorial_iterative(n));
        printf("Recursive solution: %d! = %lld\n", n, factorial_recursive(n));
    }

    return 0;
}

/*
    Function: factorial_iterative
    Purpose: Calculates n! using a standard 'for' loop.
    This approach is generally more efficient in C for this problem.
*/
long long factorial_iterative(int n) {
    long long result = 1;
    // We loop from 1 up to n, multiplying the result by each number.
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}

/*
    Function: factorial_recursive
    Purpose: Calculates n! using recursion.
    This approach is often more elegant and closer to the mathematical definition.
*/
long long factorial_recursive(int n) {
    // 1. Base Case: The condition that stops the recursion.
    // If n is 0, we know the answer is 1, so we just return it.
    if (n == 0) {
        return 1;
    }
    // 2. Recursive Step: The function calls itself with a smaller input.
    // We calculate `n * factorial(n-1)`. The call to `factorial(n-1)` will
    // continue until it hits the base case of `factorial(0)`.
    else {
        return n * factorial_recursive(n - 1);
    }
}

/*
    How does `factorial_recursive(4)` work?
    - It returns 4 * factorial_recursive(3)
      - which returns 3 * factorial_recursive(2)
        - which returns 2 * factorial_recursive(1)
          - which returns 1 * factorial_recursive(0)
            - which returns 1 (Base Case!)
    The results are then multiplied back up the chain: 1*1 -> 2*1 -> 3*2 -> 4*6 = 24.
*/
```

```c
/*
    ================================================================================
    Tutorial: Recursion - Fibonacci Numbers
    ================================================================================

    This program calculates the Nth number in the Fibonacci sequence, another
    classic example of recursion.

    The Fibonacci sequence is defined as:
    F(1) = 0
    F(2) = 1
    F(n) = F(n-1) + F(n-2) for n > 2

    This definition translates very directly into a recursive function.

    **A VERY IMPORTANT NOTE ON EFFICIENCY:**
    While this is a great example for learning recursion, this specific
    implementation is extremely inefficient! It recalculates the same values
    many, many times. For example, to calculate `fibonacci(5)`, it calculates
    `fibonacci(3)` twice, `fibonacci(2)` three times, etc. For larger numbers
    (try anything over 40), this becomes incredibly slow. A loop-based (iterative)
    solution is much more efficient for this particular problem.

    Concepts Covered:
    - Recursion with multiple base cases.
    - Recursion with multiple recursive calls in a single return statement.
    - An example of an inefficient recursive algorithm.
*/

#include <stdio.h>

int fibonacci(int n);

int main(void) {
    int N, fib;
    printf("Which Fibonacci number would you like (e.g., 1st, 2nd, 3rd...)? ");
    scanf("%d", &N);

    if (N <= 0) {
        printf("%d is not a positive number. Aborting!\n", N);
    } else {
        fib = fibonacci(N);
        printf("The %dth Fibonacci number is %d.\n", N, fib);
    }

    return 0;
}

/*
    Function: fibonacci
    Purpose: Calculates the Nth Fibonacci number using recursion.
*/
int fibonacci(int n) {
    // Base Case 1: The first Fibonacci number is 0.
    if (n == 1) {
        return 0;
    }
    // Base Case 2: The second Fibonacci number is 1.
    else if (n == 2) {
        return 1;
    }
    // Recursive Step: The Nth number is the sum of the previous two.
    // The function makes two recursive calls to itself.
    else {
        return (fibonacci(n - 1) + fibonacci(n - 2));
    }
}

/*
    How does `fibonacci(5)` work?
    - It returns fibonacci(4) + fibonacci(3)
      - fibonacci(4) returns fibonacci(3) + fibonacci(2)
        - fibonacci(3) returns fibonacci(2) + fibonacci(1) -> returns 1 + 0 = 1
        - fibonacci(2) returns 1 (Base Case)
        - So, fibonacci(4) returns 1 + 1 = 2
      - fibonacci(3) returns fibonacci(2) + fibonacci(1) -> returns 1 + 0 = 1
    - Finally, fibonacci(5) returns 2 + 1 = 3.
    Notice that `fibonacci(3)` was calculated twice! This is the source of the inefficiency.
*/
```

```c
/*
    ================================================================================
    Tutorial: Recursion - Sum of Digits
    ================================================================================

    This program calculates the sum of the digits of an integer using a very
    elegant recursive solution. This is a great example of thinking about a
    problem in a recursive way.

    The core idea is to recognize that the sum of digits of a number (e.g., 123)
    is equal to `(the last digit) + (the sum of digits of the rest of the number)`.
    sum(123) = 3 + sum(12)
    sum(12)  = 2 + sum(1)
    sum(1)   = 1 + sum(0)

    This pattern leads directly to a recursive solution.

    Concepts Covered:
    - A non-obvious but elegant application of recursion.
    - Using the modulo (%) and division (/) operators to decompose a number.
*/

#include <stdio.h>

int sumOfDigits(int n);

int main(void) {
    int n, sum;
    printf("Enter an integer to sum its digits: ");
    scanf("%d", &n);

    // Let's handle negative numbers gracefully.
    int input_num = n;
    if (n < 0) {
        n = -n; // Work with the positive version for the calculation.
    }

    sum = sumOfDigits(n);

    printf("The sum of the digits of %d is %d.\n", input_num, sum);

    return 0;
}

/*
    Function: sumOfDigits
    Purpose: Recursively calculates the sum of the digits of an integer.
*/
int sumOfDigits(int n) {
    // Base Case: If the number is 0, the sum of its digits is 0.
    // This stops the recursion.
    if (n == 0) {
        return 0;
    }
    // Recursive Step:
    else {
        // Here's the mathematical trick:
        // 1. `n % 10` (modulo 10) gives you the last digit of the number.
        //    (e.g., 123 % 10 is 3)
        // 2. `n / 10` (integer division by 10) effectively removes the last digit.
        //    (e.g., 123 / 10 is 12)

        // We return the last digit PLUS the sum of the digits of the rest of the number.
        return (n % 10) + sumOfDigits(n / 10);
    }
}

/*
    How does sumOfDigits(123) work?
    - It returns 3 + sumOfDigits(12)
      - which returns 2 + sumOfDigits(1)
        - which returns 1 + sumOfDigits(0)
          - which returns 0 (Base Case!)
    The results are then added back up the chain: 1 + 0 = 1 -> 2 + 1 = 3 -> 3 + 3 = 6.
*/
```

Chapter 9

Memory Management

```c
/*
    ===============================================================================
    Tutorial: Memory Management - The `sizeof` Operator
    ===============================================================================

    This program introduces the `sizeof` operator, a fundamental tool for
    understanding and managing memory in C.

    `sizeof` is a compile-time operator that returns the size, in bytes, of a
    variable or a data type. A "byte" is the standard unit of data storage in
    a computer (typically 8 bits).

    The exact size of data types (like `int`, `double`, etc.) can vary between
    different computer architectures (e.g., 32-bit vs. 64-bit systems). `sizeof`
    allows you to write portable code that doesn't rely on hard-coded "magic numbers"
    for type sizes.

    This program calculates the total memory required for a list of items of
    different data types and then formats the result in a human-readable way
    (Bytes, Kilobytes, and Megabytes).

    Concepts Covered:
    - The `sizeof` operator.
    - The size in bytes of common data types.
    - Memory usage calculation.
    - Integer arithmetic for formatting byte counts into KB/MB.
*/

#include <stdio.h>

int main(void) {
    int num_item_types = 0;
    int quantity = 0;
    char type_code; // 'i' for int, 'd' for double, 'c' for char
    long long total_bytes = 0; // Use long long to handle potentially large sizes.

    printf("How many different item types are there? ");
    scanf("%d", &num_item_types);

    printf("For each type, enter the quantity and the type code (i, d, or c).\n");

    for (int i = 0; i < num_item_types; i++) {
        printf("Enter quantity and type for item #%d: ", i + 1);
        scanf("%d %c", &quantity, &type_code);

        if (type_code == 'i') {
            total_bytes += sizeof(int) * quantity;
        } else if (type_code == 'd') {
            total_bytes += sizeof(double) * quantity;
        } else if (type_code == 'c') {
            total_bytes += sizeof(char) * quantity;
        } else {
            printf("Invalid tracking code type '%c'.\n", type_code);
        }
    }

    printf("\n--- Total Memory Required ---\n");
    printf("Raw bytes: %lld\n", total_bytes);
    // Format the byte count into a more human-readable format.
    // 1 KB = 1000 Bytes, 1 MB = 1000 KB = 1,000,000 Bytes
    if (total_bytes >= 1000000) {
        int megabytes = total_bytes / 1000000;
        int kilobytes = (total_bytes % 1000000) / 1000;
        int bytes = total_bytes % 1000;
        printf("Formatted: %d MB, %d KB, and %d B\n", megabytes, kilobytes, bytes);
    } else if (total_bytes >= 1000) {
        int kilobytes = total_bytes / 1000;
        int bytes = total_bytes % 1000;
        printf("Formatted: %d KB and %d B\n", kilobytes, bytes);
    } else {
        printf("Formatted: %lld B\n", total_bytes);
    }

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  On your system, what are the sizes of `short`, `long`, `float`, and
        `long long`? Write a simple program to print the `sizeof` each one.
    2.  What is the size of a pointer, like `int *`? You'll find it's likely
        4 or 8, depending on whether you're on a 32-bit or 64-bit system.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Memory Management - Dynamic Memory Allocation
    ================================================================================

    This program introduces one of C's most powerful and advanced features:
    **dynamic memory allocation**.

    So far, the size of all our arrays had to be a fixed constant known when we
    compile the program. Dynamic allocation allows us to request a block of
    memory of a specific size *while the program is running*.

    This is essential when you don't know how much memory you'll need until
    runtime (e.g., the user tells you how many items they want to enter).

    The two key functions are from `<stdlib.h>`:
    - `malloc(size)`: "Memory Allocate". It requests a block of `size` bytes
      from the operating system (from a memory pool called the "heap"). It
      returns a `void *` pointer to the start of that block, or `NULL` if the
      allocation fails.
    - `free(pointer)`: Releases the block of memory pointed to by `pointer`
      back to the system, so it can be used again.

    **THE GOLDEN RULE:** For every call to `malloc()`, there must be exactly
    one call to `free()`. Forgetting to `free` memory causes a "memory leak".

    Concepts Covered:
    - The need for dynamic allocation.
    - The `malloc()` and `free()` functions.
    - Casting the `void *` returned by `malloc()`.
    - The "off-by-one" error with null terminators (a very common bug!).
    - Checking for `malloc` failure.
*/

#include <stdio.h>
#include <stdlib.h> // Required for malloc() and free()

char *allocateString(int num_chars);

int main(void) {
    int lengthLight, lengthDark;
    char *strLight, *strDark;

    printf("Enter the desired length for the light and dark side strings: ");
    scanf("%d %d", &lengthLight, &lengthDark);

    // Allocate memory for the strings based on user input.
    strLight = allocateString(lengthLight);
    strDark = allocateString(lengthDark);

    // It's crucial to check if malloc succeeded. If the system is out of
    // memory, it will return NULL.
    if (strLight == NULL || strDark == NULL) {
        printf("Error: Memory allocation failed.\n");
        return 1; // Exit with an error code.
    }

    printf("Enter the light side setting (max %d chars): ", lengthLight);
    scanf("%s", strLight);
    printf("Enter the dark side setting (max %d chars): ", lengthDark);
    scanf("%s", strDark);

    printf("Local settings: %s - %s\n", strLight, strDark);

    // **CRUCIAL STEP:** Free the memory when you are done with it.
    // If you don't, your program will have a memory leak.
    free(strLight);
    free(strDark);

    return 0;
}

/*
    Function: allocateString
    Purpose: Dynamically allocates enough memory to hold a string of a given
             number of characters.
*/
char *allocateString(int num_chars) {
    // --- A VERY COMMON BUG AND ITS FIX ---
    // The original code was `malloc(numChars * sizeof(char))`.
    // The problem is that a string needs one extra byte for the null terminator ('\0').
    // If you ask for 5 chars, you need 6 bytes of space. `scanf` will write the
    // null terminator at the 6th byte, which is outside the allocated memory.
    // This is a "buffer overflow" and a major security vulnerability.
    //
    // The FIX is to always allocate `num_chars + 1` bytes.
    // `sizeof(char)` is always 1 by definition, so it's optional, but good for clarity.
    char *ptr = (char *) malloc((num_chars + 1) * sizeof(char));

    return ptr;
}
```

Chapter 10

Structures

```c
/*
    ==============================================================================
    Tutorial: Structures - Defining Your Own Data Types
    ==============================================================================

    Welcome to Structures, or `struct`s. This is a powerful feature that allows
    you to create your own custom data types by grouping other variables together.

    Think of a `struct` as a "blueprint" for a new type of variable. If you want
    to represent a student, you need to store their name, age, grade, etc. A
    `struct` lets you bundle all of this related information into a single,
    neat package.

    Concepts Covered:
    - The `struct` keyword for defining a new structure type.
    - Declaring a variable of your new `struct` type.
    - Initializing a `struct` variable with data.
    - Accessing the members of a `struct` using the dot operator (`.`).
*/

#include <stdio.h>

// Here, we define a new structure named 'student'.
// This acts as a template. It doesn't allocate any memory yet, it just
// defines what a 'student' variable will look like.
struct student {
    // These are the "members" or "fields" of the structure.
    char firstName[30];
    char lastName[30];
    int birthYear;
    double aveGrade;
};

int main(void) {
    // Now we declare a variable of our new type, `struct student`.
    // This creates a variable named `me` in memory that has space for all the
    // members we defined in the blueprint.
    // We can initialize it directly using curly braces, with the values in order.
    struct student me = {"Petra", "Bonfert-Taylor", 1989, 3.5};

    // Let's create another student variable.
    struct student you = {"Remi", "Sharrock", 2005, 3.5};

    // To access the data inside a struct variable, we use the dot operator (`.`).
    printf("Names: %s %s, %s %s\n", me.firstName, me.lastName, you.firstName, you.lastName);

    // You can access any member of the struct.
    printf("Year of birth for %s: %d\n", me.firstName, me.birthYear);
    printf("Average grade for %s: %.2lf\n", you.firstName, you.aveGrade);

    return 0;
}

/*
    ==============================================================================
    Further Exploration:
    ==============================================================================
    1.  Define a new struct to represent a `car`. It could have members like
        `char make[20]`, `char model[20]`, `int year`, and `double price`.
    2.  Create a couple of `car` variables, initialize them, and print out
        their details.
    3.  You can also assign one struct variable to another, like `struct student new_student =
    ↪   me;`.
        This copies the values of all members. Try it out!
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Structures - Populating Structs with User Input
    ================================================================================

    This program builds on the previous example. Instead of initializing the
    `struct` with hard-coded values, we declare an uninitialized `struct` variable
    and then fill its members with data provided by the user.

    The key takeaway is how we use the dot operator (`.`) to access each member
    and pass it to `scanf`.

    Concepts Covered:
    - Declaring an uninitialized `struct` variable.
    - Using the dot operator (`.`) to access members for writing/input.
    - The difference in `scanf` usage for arrays vs. other types.
*/

#include <stdio.h>

struct student {
    char firstName[30];
    char lastName[30];
    int birthYear;
    double aveGrade;
};

int main(void) {
    // We declare a `struct student` variable named 'learner'.
    // At this point, the values of its members are garbage (undefined).
    struct student learner;

    printf("Enter the student's first name: ");
    // IMPORTANT: When scanning a string into a character array like `learner.firstName`,
    // you do NOT use the '&' address-of operator. This is because the name of an
    // array already "decays" into a pointer to its first element.
    scanf("%s", learner.firstName);

    printf("Enter the student's last name: ");
    scanf("%s", learner.lastName);

    printf("Enter the student's year of birth: ");
    // For simple numeric types like `int` and `double`, you DO need the '&'
    // to pass the memory address of that specific member to scanf.
    scanf("%d", &learner.birthYear);

    printf("Enter the student's average grade: ");
    scanf("%lf", &learner.aveGrade);

    printf("\n--- Student Record ---\n");
    printf("Name: %s %s\n", learner.firstName, learner.lastName);
    printf("Year of birth: %d\n", learner.birthYear);
    printf("Average grade: %.2lf\n", learner.aveGrade);

    return 0;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  After populating the `learner` struct, declare a new struct variable,
        `struct student partner;`, and then try to copy the data using a single
        assignment: `partner = learner;`. Print the details of `partner` to
        confirm that all the members were copied successfully.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Structures - Structs and Memory Layout (Padding)
    ================================================================================

    This program uses the `sizeof` operator to inspect the amount of memory
    used by a `struct` and its individual members.

    This often reveals a surprising and important concept in C: **structure padding**.

    You might expect the total size of the `struct` to be the exact sum of the
    sizes of its members. However, it is often larger. The compiler adds invisible,
    unused bytes ("padding") between members to ensure that each member is
    aligned on a memory address that is a multiple of its size. This is done
    for performance reasons, as accessing aligned data is much faster on most
    computer architectures.

    Concepts Covered:
    - Using `sizeof` on a `struct` type and its members.
    - Structure padding and memory alignment.
    - The `%zu` format specifier for printing `size_t` types (the type returned by `sizeof`).
*/

#include <stdio.h>

// We use smaller array sizes here to make the padding more obvious.
struct student {
    char firstName[5]; // 5 bytes
    char lastName[5];  // 5 bytes
    int birthYear;     // Usually 4 bytes
    double aveGrade;   // Usually 8 bytes
};

int main(void) {
    struct student me;

    // The `%zu` format specifier is the correct, portable way to print a value
    // of type `size_t`, which is what `sizeof` returns.
    printf("Size of firstName is %zu bytes.\n", sizeof(me.firstName));
    printf("Size of lastName is %zu bytes.\n", sizeof(me.lastName));
    printf("Size of birthYear is %zu bytes.\n", sizeof(me.birthYear));
    printf("Size of aveGrade is %zu bytes.\n", sizeof(me.aveGrade));

    size_t sum_of_members = sizeof(me.firstName) + sizeof(me.lastName) +
                            sizeof(me.birthYear) + sizeof(me.aveGrade);

    printf("\nSum of the sizes of individual members: %zu bytes.\n", sum_of_members);

    // Now, let's see the size of the entire structure.
    // It will likely be larger than the sum of its parts!
    printf("Total size of the 'struct student' is %zu bytes.\n", sizeof(struct student));

    printf("\nThe difference is due to memory padding added by the compiler for alignment.\n");

    return 0;
}

/*
    Why the padding?
    Imagine memory as a series of numbered boxes. A `double` (8 bytes) is most
    efficiently accessed if it starts at a memory address divisible by 8.
    - firstName[5] takes 5 bytes.
    - lastName[5] takes another 5 bytes. Total = 10 bytes.
    - The next member, `birthYear` (an int), needs to be aligned. Let's say it
      starts at address 12 (the compiler adds 2 bytes of padding). It takes 4 bytes. Total = 16.
    - The next member, `aveGrade` (a double), needs to be aligned to an address
      divisible by 8. The current address is 16, which is divisible by 8, so no
      padding is needed here. It takes 8 bytes. Total = 24.
    - The compiler might even add padding at the *end* of the struct to ensure
      that in an array of these structs, each one starts on a properly aligned
      boundary. So the final size might be 24 or even 32.

    The exact rules for padding are compiler-dependent, but the principle is the same.
*/
```

```c
/*
    ================================================================================
    Tutorial: Structures - Structs and Functions (Pass-by-Value vs. Pass-by-Pointer)
    ================================================================================

    This program demonstrates the two primary ways to work with `struct`s and
    functions. It highlights when to pass the whole `struct` and when to pass a
    pointer to it.

    1.  **Pass-by-Value (`printPoint`)**:
        - A *copy* of the entire `struct` is made and given to the function.
        - The function can read the data, but any changes it makes are to the
          copy and will NOT affect the original `struct` in `main`.
        - This is simple and safe for read-only operations, but can be inefficient
          if the `struct` is very large, as copying takes time and memory.
        - We use the **dot operator (`.`)** to access members.

    2.  **Pass-by-Pointer (`readPoint`)**:
        - Only the *memory address* of the `struct` is passed to the function.
        - This is very efficient as no data is copied.
        - The function can now modify the original `struct`'s data directly. This
          is essential for functions that need to write or change data, like `readPoint`.
        - We use the **arrow operator (`->`)** as a convenient shortcut to
          dereference the pointer and access a member.

    Concepts Covered:
    - Passing a `struct` to a function by value.
    - Passing a pointer to a `struct` to a function.
    - The dot operator (`.`) for direct access.
    - The arrow operator (`->`) for access via a pointer.
*/

#include <stdio.h>

struct point {
    int x;
    int y;
};

// Function prototypes
void printPoint(struct point pt);
void readPoint(struct point *ptr);

int main(void) {
    struct point z;

    // We pass the ADDRESS of `z` to readPoint so it can be filled with data.
    readPoint(&z);

    printf("\nBack in main, the point is: ");
    // We pass `z` itself (a copy) to printPoint, since it only needs to read the data.
    printPoint(z);

    return 0;
}

/*
    Function: readPoint
    Purpose: Reads x and y coordinates from the user into a `point` struct.
    Method: Pass-by-Pointer.
*/
void readPoint(struct point *ptr) {
    printf("\nEnter a new point: \n");
    printf("x-coordinate: ");

    // The Arrow Operator (`->`):
    // This is a shortcut. The expression `ptr->x` is exactly equivalent to
    // `(*ptr).x`, which means:
    // 1. Dereference the pointer `ptr` to get the whole `struct`.
    // 2. Use the dot operator on that `struct` to access the `x` member.
    // The arrow operator is much cleaner and easier to read.
    scanf("%d", &ptr->x);

    printf("y-coordinate: ");
    scanf("%d", &ptr->y);
}

/*
    Function: printPoint
    Purpose: Prints the coordinates of a `point` struct.
    Method: Pass-by-Value.
*/
void printPoint(struct point pt) {
    // Here, `pt` is a copy of the `z` variable from main.
    // Since we have the struct directly, we use the dot operator.
    printf("(%d, %d)\n", pt.x, pt.y);
}
```

```c
/*
    ================================================================================
    Tutorial: Structures - Arrays of Structs
    ================================================================================

    Now that we know how to create a `struct`, we can create an array of them!
    This is an extremely common and powerful pattern for managing a collection
    of related data objects, like a class roster of students, a fleet of cars,
    or, in this case, the vertices of a polygon.

    This program reads the vertices of a polygon from the user, stores them in
    an array of `point` structs, and then prints them back out.

    This example merges the logic from two previous files into a more general
    and robust program.

    Concepts Covered:
    - Declaring an array of `struct`s.
    - Passing an array of `struct`s to a function.
    - Looping through an array of `struct`s to process each element.
*/

#include <stdio.h>

struct point {
    int x;
    int y;
};

// Function Prototypes
void printPoint(struct point pt);
void readPoint(struct point *ptr);
void printPolygon(struct point polygon[], int num_vertices);
void readPolygon(struct point polygon[], int num_vertices);


int main(void) {
    int num_vertices;

    printf("How many vertices does the polygon have? ");
    scanf("%d", &num_vertices);

    // Basic input validation
    if (num_vertices <= 0) {
        printf("A polygon must have at least 1 vertex.\n");
        return 1; // Exit with an error
    }

    // A Note on Variable-Length Arrays (VLAs):
    // `struct point polygon[num_vertices];` creates a VLA, whose size is
    // determined at runtime. While convenient, they are not part of the most
    // recent C standards and can be risky if `num_vertices` is very large.
    // A more common and safer approach in production code would be to either
    // declare an array with a fixed maximum size, or use dynamic allocation (`malloc`).
    struct point polygon[num_vertices];

    readPolygon(polygon, num_vertices);
    printPolygon(polygon, num_vertices);
    return 0;
}

/*
    Function: readPolygon
    Purpose: Reads the coordinates for all vertices of a polygon.
    It loops and calls readPoint for each vertex.
*/
void readPolygon(struct point polygon[], int num_vertices) {
    printf("\nEnter the %d vertices for the polygon:\n", num_vertices);
    for (int i = 0; i < num_vertices; i++) {
        printf("Vertex #%d:\n", i + 1);
        // We pass the address of the current struct in the array, `&polygon[i]`,
        // to the readPoint function.
        readPoint(&polygon[i]);
    }
}

/*
    Function: printPolygon
    Purpose: Prints the coordinates of all vertices of a polygon.
*/
void printPolygon(struct point polygon[], int num_vertices) {
    printf("\nThe vertices of your polygon are:\n");
    for (int i = 0; i < num_vertices; i++) {
        // We pass the struct itself, `polygon[i]`, by value to printPoint.
        printPoint(polygon[i]);
    }
}

/*
    Function: readPoint
    Purpose: Reads a single point's coordinates from the user.
*/
void readPoint(struct point *ptr) {
    printf("  x-coordinate: ");
    scanf("%d", &ptr->x);
    printf("  y-coordinate: ");
    scanf("%d", &ptr->y);
}

/*
    Function: printPoint
    Purpose: Prints a single point in (x, y) format.
*/
void printPoint(struct point pt) {
    printf("  (%d, %d)\n", pt.x, pt.y);
}
```

```c
/*
    ================================================================================
    Tutorial: Structures - Returning a Struct from a Function
    ================================================================================

    This program introduces a new and powerful concept: functions that return
    an entire `struct`.

    So far, our functions that modify structs have taken a pointer to the original
    struct. An alternative design pattern is to have the function take a `struct`
    by value (as a copy), perform some calculations, and then return a *new*,
    modified `struct`.

    This program defines a `date` struct and a function `advanceDay` that takes
    one date and returns a new `date` struct representing the following day.

    Concepts Covered:
    - Declaring a function that returns a `struct`.
    - Returning a `struct` variable from a function.
    - A practical algorithm for date manipulation.
    - The limitations of the algorithm (e.g., leap years).
*/

#include <stdio.h>

struct date {
    int year;
    int month;
    int day;
};

// Function Prototypes
void readDate(struct date *date_ptr);
void printDate(struct date d);
struct date advanceDay(struct date d);

int main(void) {
    struct date today, tomorrow;

    printf("Enter today's date (YYYY MM DD): ");
    readDate(&today);

    printf("Today's date is: ");
    printDate(today);

    // The advanceDay function returns a whole new struct, which we assign to 'tomorrow'.
    tomorrow = advanceDay(today);

    printf("Tomorrow's date is: ");
    printDate(tomorrow);

    return 0;
}

void readDate(struct date *date_ptr) {
    scanf("%d %d %d", &date_ptr->year, &date_ptr->month, &date_ptr->day);
}

void printDate(struct date d) {
    // `%02d` is a format specifier that pads a number with a leading zero if
    // it is less than 2 digits wide. Perfect for dates and times.
    printf("%04d/%02d/%02d\n", d.year, d.month, d.day);
}

/*
    Function: advanceDay
    Purpose: Calculates the date of the day after the one provided.
    Note: This is a simplified version that does not account for leap years.
*/
struct date advanceDay(struct date d) {
    struct date next_day;
    // An array to hold the number of days in each month. Index 0 is unused.
    int days_in_month[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

    // Case 1: It's not the end of the month.
    // We simply increment the day and keep the month and year the same.
    if (d.day < days_in_month[d.month]) {
        next_day.day = d.day + 1;
        next_day.month = d.month;
        next_day.year = d.year;
    }
    // Case 2: It IS the end of the month, but not the end of the year.
    else if (d.month < 12) {
        next_day.day = 1;
        next_day.month = d.month + 1;
        next_day.year = d.year;
    }
    // Case 3: It's December 31st.
    else { // d.month == 12
        next_day.day = 1;
        next_day.month = 1;
        next_day.year = d.year + 1;
    }

    return next_day;
}

/*
    ================================================================================
    Further Exploration:
    ================================================================================
    1.  The current `advanceDay` function doesn't handle leap years (where
        February has 29 days). A year is a leap year if it is divisible by 4,
        unless it is divisible by 100 but not by 400. Can you add this logic?
        You would need to check `d.year` inside the `advanceDay` function and
        adjust `days_in_month[2]` accordingly.
    ================================================================================
*/
```

```c
/*
    ================================================================================
    Tutorial: Structures - Dynamic Allocation of Structs
    ================================================================================

    This program is a capstone example that brings together many of the topics
    we've covered: structs, pointers, arrays, and dynamic memory allocation.

    It creates a dynamically-sized array of `point` structs. This is the proper,
    safe, and standard way to handle cases where you don't know the size of your
    array until runtime, and it's the preferred alternative to Variable-Length
    Arrays (VLAs).

    The process is:
    1.  Ask the user how many vertices they need.
    2.  Use `malloc` to request enough memory for that many `point` structs.
    3.  Treat the resulting block of memory as an array.
    4.  `free` the memory when we are done.

    Concepts Covered:
    - Using `malloc` to create a dynamic array of structs.
    - Using `sizeof(struct type)` to calculate the correct memory size.
    - Accessing elements in the dynamically allocated array.
    - The importance of `free`ing the allocated memory.
*/

#include <stdio.h>
#include <stdlib.h> // For malloc() and free()

struct point {
    int x;
    int y;
};

void printPoint(struct point pt);
void readPoint(struct point *ptr);
void printPolygon(struct point *polygon, int num_vertices);

int main(void) {
    // `polygon` is not an array. It's a pointer that will hold the
    // starting address of a block of memory we get from malloc.
    struct point *polygon;
    int num_vertices;

    printf("How many vertices does your polygon have? ");
    scanf("%d", &num_vertices);

    // Dynamic Allocation:
    // 1. We want space for `num_vertices` number of `point` structs.
    // 2. `sizeof(struct point)` gives us the size of a single struct.
    // 3. We multiply them to get the total number of bytes needed.
    // 4. `malloc` returns a `void *`, which we cast to `struct point *`.
    polygon = (struct point *) malloc(num_vertices * sizeof(struct point));

    // ALWAYS check if malloc was successful.
    if (polygon == NULL) {
        printf("Error: Memory allocation failed.\n");
        return 1;
    }
    // Now, we can treat `polygon` as if it were an array.
    // We can use the familiar `[]` array notation to access elements.
    for (int i = 0; i < num_vertices; i++) {
        printf("Enter vertex #%d:\n", i + 1);
        readPoint(&polygon[i]); // Pass the address of the i-th struct
    }

    printPolygon(polygon, num_vertices);

    // THE GOLDEN RULE: If you `malloc`, you must `free`.
    // This releases the entire block of memory back to the system.
    free(polygon);
    polygon = NULL; // Good practice to prevent using a "dangling pointer".

    return 0;
}

void readPoint(struct point *ptr) {
    printf("  x-coordinate: ");
    scanf("%d", &ptr->x);
    printf("  y-coordinate: ");
    scanf("%d", &ptr->y);
}

void printPolygon(struct point *polygon, int num_vertices) {
    printf("\nYour polygon's vertices are:\n");
    for (int i = 0; i < num_vertices; i++) {
        printPoint(polygon[i]);
    }
}

void printPoint(struct point pt) {
    printf("  (%d, %d)\n", pt.x, pt.y);
}
```