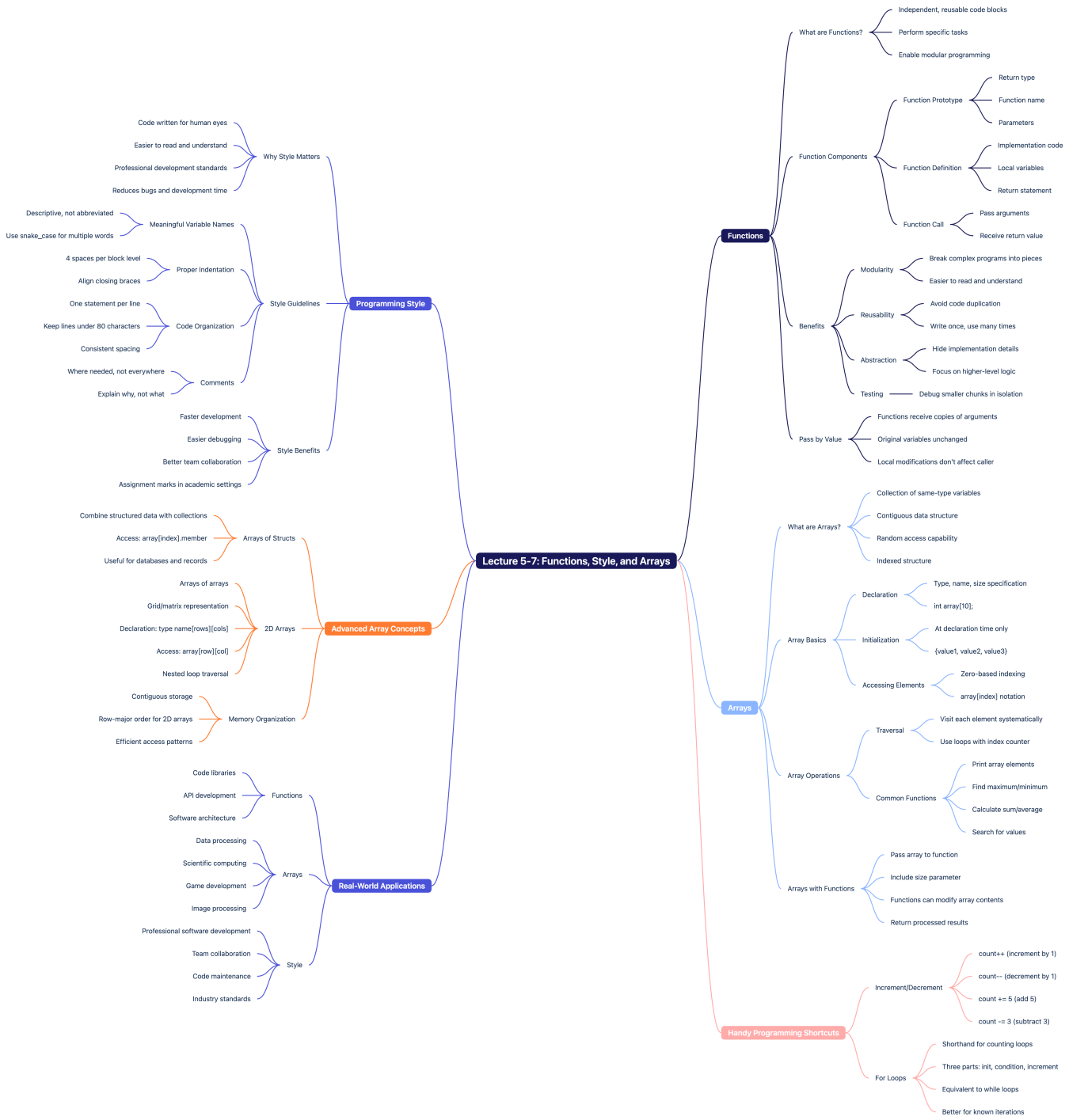
Lecture 3-6: Functions, Style, and Arrays - Building Professional Programs



Why These Topics Matter

These concepts transform basic programs into professional, maintainable software. Functions let you break complex problems into manageable pieces, good style makes code readable and collaborative, and arrays provide powerful ways to organize and process data. Together, they form the foundation of modern software development practices that you'll use throughout your programming career.

Discussion: What makes a program "professional" versus just "working"? Think about apps you use daily - what makes them well-designed?

1. Functions: The Computer's Specialized Workers

The Power of Modular Programming

Think of functions as specialized workers in a factory. Instead of having one person try to build an entire car from scratch, you have different workers for different tasks - one person installs the engine, another handles the electrical system, and another paints the body. Each worker knows exactly how to do their specific job, and you can call on them whenever you need that particular skill. It's like having a team of experts rather than trying to be an expert at everything yourself.

Discussion: What tasks in your daily life could be broken down into smaller, specialized functions? (Making breakfast, getting ready for class, etc.)

Function Anatomy: The Blueprint System

```
// Function prototype - like a job description posted on a bulletin board
// Tells the computer "there will be a function called maximum that takes
two integers and returns one"
int maximum(int x, int y);

// Function definition - the actual worker doing the job
// This is where we write the instructions for what the function should do
int maximum(int x, int y) {
    int max = x;  // Start by assuming the first number is the largest
    if (x < y) {  // Check if the second number is actually bigger
        max = y;  // If so, update our assumption
    }
    return max;   // Send back the result of our work
}

// Function call - like calling a worker to do their job
// We give them the materials (10 and 7) and they give us back the result
int result = maximum(10, 7);
```

Exercise: What will this function return: `maximum(15, 15)`? What about `maximum(-5, 3)`?

Different Types of Workers

Value-Returning Functions are like workers who produce something tangible. When you ask them to do a job, they give you back a product. It's like asking a baker to make bread - you give them ingredients and they give you back a loaf.

```
// This function is like a calculator that adds two numbers
// It takes two inputs and gives you back their sum
int add_numbers(int a, int b) {
```

```
    return a + b;  // The "product" this worker produces
}
```

Void Functions are like workers who perform actions but don't produce physical products. They're more like service workers - they do something useful but don't give you something to take away. It's like asking someone to clean your room - they do the work, but you don't get a physical object back.

```
// This function is like a printer that prints a certain number of stars
// It does the printing work but doesn't give you anything back
void print_stars(int count) {
    for (int i = 0; i < count; i++) {  // Repeat the printing count times
        printf("*");  // Print one star each time
    }
    printf("\n");  // Move to the next line when done
}
```

Challenge: Design a function that takes a temperature and prints whether it's hot, cold, or moderate.

The Copy Machine Effect: Pass by Value

```
// This function tries to change a number, but it's working with a copy
void increment(int x) {
    x = x + 1;  // Only modifies the local copy, not the original
    printf("Inside function: %d\n", x);  // Shows the modified copy
}

int main(void) {
    int num = 5;  // Original number
    increment(num);  // Function gets a copy of 5, not the original
    printf("In main: %d\n", num);  // Original is still 5
    return 0;
}
```

Think of this like making a photocopy of an important document. You can write all over the copy, but the original document remains unchanged. When you pass a variable to a function, the function gets its own copy to work with, leaving your original data safe.

Question: Why might this "copy machine" behavior be useful in programming?

2. Programming Style: The Art of Readable Code

Why Style Matters

Good programming style is like writing clear, legible handwriting. Just as messy handwriting makes it hard for others to read your notes, poor code style makes it difficult for other programmers (including future you) to understand what your program does. It's the difference between a well-organized kitchen where everything has its place and a chaotic one where you can never find the spatula.

Discussion: What makes text easy or hard to read? How does this apply to code?

The Style Transformation

```
// BAD STYLE - like writing a sentence without spaces or punctuation
// It works but is nearly impossible to read and understand
int x,y,z;if(x>y){z=x;}else{z=y;}

// GOOD STYLE - clear, organized, and easy to understand
// Each variable has a meaningful name and proper spacing
int first_number;
int second_number;
int maximum_value;

if (first_number > second_number) {
    maximum_value = first_number;
} else {
    maximum_value = second_number;
}
```

Exercise: Rewrite this messy code with good style:

```
int a,b,c;if(a>b&&a>c){printf("a is biggest");}else if(b>c){printf("b is biggest");}else{printf("c is biggest");}
```

Style Guidelines Table

Principle	Bad Example	Good Example	Why It Matters
Meaningful Names	`int x, y, z;`	`int student_count, grade, age;`	Names should tell the story
Proper Indentation	`if(x>0) {y=x*2;}else{y=0;}`	`if (x > 0) { y = x * 2; } else { y = 0; }`	Shows code structure clearly
One Statement Per Line	`x=5;y=10;z=x+y;`	`x = 5; y = 10; z = x + y;`	Easier to read and debug
Consistent Spacing	`if(x>0&&y<10)`	`if (x > 0 && y < 10)`	Makes operators stand out
Appropriate Comments	`// adds numbers`	`// Calculate total score from all assignments`	Explains the "why", not the "what"

Challenge: Find three style problems in this code:

```
int main(void){int x=5,y=10;if(x<y){printf("x is smaller");}else{printf("y
is smaller");}return 0;}
```

3. Arrays: The Computer's Filing Cabinet

Organizing Related Data

Arrays are like filing cabinets where each drawer contains related information. Instead of having separate variables scattered around for each piece of data, you have one organized storage system. It's like the difference between having loose papers scattered on your desk versus having them neatly organized in folders. When you need to find something, you just need to know which folder (array) and which position in the folder (index).

Discussion: What collections of related data do you work with daily? (Grades, phone contacts, shopping lists, etc.)

Basic Array Operations: The Filing System

```
// Declaration and initialization - like setting up a new filing cabinet
// We create 5 drawers and put specific files in each one
int scores[5] = {85, 92, 78, 96, 88};

// Accessing elements - like opening a specific drawer
// Remember: arrays start counting at 0, not 1!
printf("First score: %d\n", scores[0]);  // Gets the first score (85)
scores[2] = 95;  // Updates the third score to 95

// Array traversal - like checking every drawer systematically
for (int i = 0; i < 5; i++) {  // Go through each drawer
    printf("Score %d: %d\n", i + 1, scores[i]);  // Print each score
}
```

Exercise: What will this print?

```
int numbers[4] = {10, 20, 30, 40};
printf("%d %d %d\n", numbers[0], numbers[2], numbers[3]);
```

Arrays and Functions: The Team Approach

```
// Function to print array - like a worker who knows how to display all
files
void print_array(int size, int array[]) {
    for (int i = 0; i < size; i++) {  // Check each position
        printf("%d ", array[i]);  // Print each number with a space
    }
```

```
    printf("\n");  // Move to next line when done
}

// Function to find maximum - like a worker who can find the highest value
int find_maximum(int size, int array[]) {
    int max = array[0];  // Start by assuming first number is the largest
    for (int i = 1; i < size; i++) {  // Check all other numbers
        if (array[i] > max) {  // If we find a bigger one
            max = array[i];  // Update our record
        }
    }
    return max;  // Give back the largest number found
}

// Usage - like being the manager who coordinates the workers
int main(void) {
    int numbers[] = {3, 7, 2, 9, 1};  // Our data
    int size = 5;  // How much data we have

    print_array(size, numbers);  // Ask the display worker to show
everything
    int max = find_maximum(size, numbers);  // Ask the finder worker to
find the max
    printf("Maximum: %d\n", max);  // Show the result

    return 0;
}
```

Challenge: Write a function that finds the average of all numbers in an array.

Arrays of Structs: The Advanced Filing System

```
// Define a structure - like creating a custom file format
// Each student file will have these three pieces of information
struct student {
    int id;           // Student ID number
    char grade;       // Letter grade (A, B, C, etc.)
    double mark;      // Numerical mark (can have decimals)
};

int main(void) {
    struct student class[3];  // Create a filing cabinet with 3 student
files

    // Initialize first student - like filling out the first file
    class[0].id = 1001;      // Student ID
    class[0].grade = 'A';    // Letter grade
    class[0].mark = 85.5;    // Numerical mark

    // Print all students - like reviewing all files in the cabinet
    for (int i = 0; i < 3; i++) {  // Go through each student file
```

```
        printf("Student %d: Grade %c, Mark %.1lf\n",
               class[i].id, class[i].grade, class[i].mark);
    }

    return 0;
}
```

Question: What other information might you want to store about students? How would you modify the struct?

4. 2D Arrays: The Computer's Spreadsheet

Modeling Real-World Grids

2D arrays are like spreadsheets or game boards - they organize data in rows and columns. Think of a chess board: you need to specify both which row and which column to find a specific piece. It's like having a filing cabinet where each drawer contains multiple folders, and you need both the drawer number and folder number to find what you're looking for.

Discussion: What real-world situations use grids or tables? (Seating charts, game boards, spreadsheets, etc.)

2D Array Basics: The Grid System

```
// Declaration - like creating a 3x4 grid or spreadsheet
int grid[3][4];  // 3 rows, 4 columns

// Declaration with initialization - like filling out a spreadsheet with
data
int matrix[3][4] = {
    {1, 2, 3, 4},   // First row: 1, 2, 3, 4
    {5, 6, 7, 8},   // Second row: 5, 6, 7, 8
    {9, 10, 11, 12} // Third row: 9, 10, 11, 12
};

// Accessing elements - like finding a specific cell in a spreadsheet
matrix[1][2] = 99;  // Row 1, Column 2 gets the value 99
```

Exercise: What value is stored in `matrix[0][3]`? What about `matrix[2][1]`?

2D Array Traversal: The Systematic Search

```
#define ROWS 3
#define COLS 4

// Function to print 2D array - like displaying a spreadsheet
void print_2d_array(int array[ROWS][COLS]) {
    for (int row = 0; row < ROWS; row++) {      // Go through each row
```

```
        for (int col = 0; col < COLS; col++) {  // Go through each column
in that row
            printf("%3d ", array[row][col]);    // Print the value with
spacing
        }
        printf("\n");  // New line after each row
    }
}

// Calculate row sums - like adding up each row in a spreadsheet
void row_sums(int array[ROWS][COLS]) {
    for (int row = 0; row < ROWS; row++) {  // For each row
        int sum = 0;  // Start with zero for this row
        for (int col = 0; col < COLS; col++) {  // Add up all columns in
this row
            sum += array[row][col];
        }
        printf("Row %d sum: %d\n", row, sum);  // Show the total for this
row
    }
}
```

Challenge: Write a function that finds the maximum value in a 2D array.

5. Handy Shortcuts: The Programmer's Toolkit

Making Code More Concise

These shortcuts are like learning keyboard shortcuts on your computer - they make you faster and more efficient. Instead of typing out long expressions, you can use these compact versions that do the same thing.

```
// Increment/Decrement operators - like counting up or down
count++;        // Same as count = count + 1 (add 1)
count--;        // Same as count = count - 1 (subtract 1)
count += 5;     // Same as count = count + 5 (add 5)
count -= 3;     // Same as count = count - 3 (subtract 3)

// For loops - like a counting machine
for (int i = 0; i < 10; i++) {  // Count from 0 to 9
    printf("%d ", i);  // Print each number
}

// Equivalent while loop - the same counting, just written differently
int i = 0;
while (i < 10) {  // Keep going while i is less than 10
    printf("%d ", i);  // Print the current number
    i++;  // Move to the next number
}
```

Exercise: Rewrite this while loop as a for loop:

```
int j = 5;
while (j >= 0) {
    printf("%d ", j);
    j--;
}
```

6. Practical Applications: Real-World Problem Solving

Grade Calculator: The Fair Teacher

```
#include <stdio.h>

// Function prototypes - like a table of contents for our functions
double calculate_average(int size, int grades[]);
char determine_letter_grade(double average);
void print_grade_report(int size, int grades[]);

int main(void) {
    int grades[5];  // Store 5 grades

    printf("Enter 5 grades: ");
    for (int i = 0; i < 5; i++) {  // Get each grade from the user
        scanf("%d", &grades[i]);
    }

    print_grade_report(5, grades);  // Generate and display the report

    return 0;
}

// Calculate the average - like finding the middle ground
double calculate_average(int size, int grades[]) {
    int sum = 0;  // Start with zero
    for (int i = 0; i < size; i++) {  // Add up all grades
        sum += grades[i];
    }
    return (double)sum / size;  // Divide by number of grades to get
average
}

// Convert numerical average to letter grade - like applying a grading
policy
char determine_letter_grade(double average) {
    if (average >= 90) return 'A';      // 90+ is an A
    else if (average >= 80) return 'B'; // 80-89 is a B
    else if (average >= 70) return 'C'; // 70-79 is a C
    else if (average >= 60) return 'D'; // 60-69 is a D
    else return 'F';                    // Below 60 is an F
```

```
}

// Print the complete grade report - like generating a report card
void print_grade_report(int size, int grades[]) {
    printf("Grades: ");
    for (int i = 0; i < size; i++) {  // Show all individual grades
        printf("%d ", grades[i]);
    }

    double avg = calculate_average(size, grades);  // Calculate the
average
    char letter = determine_letter_grade(avg);     // Get the letter grade

    printf("\nAverage: %.1lf\n", avg);  // Show the numerical average
    printf("Letter Grade: %c\n", letter);  // Show the letter grade
}
```

You're a fair teacher who wants to give students clear feedback on their performance. You collect their grades, calculate their average, and apply a consistent grading policy to determine their final letter grade.

Question: How could this system be made more flexible? (Different grading scales, weighted averages, etc.)

Student Database: The Organized Registrar

```
#include <stdio.h>
#define MAX_STUDENTS 50

// Define student structure - like creating a standardized student record
form
struct student {
    int id;              // Student ID number
    char name_initial;   // First letter of student's name
    double gpa;          // Grade Point Average
    int year;            // Year of study (1st, 2nd, 3rd, etc.)
};

// Function prototypes - our team of specialized workers
void input_student(struct student *s);
void print_student(struct student s);
void print_all_students(int count, struct student database[]);
double calculate_class_average(int count, struct student database[]);

int main(void) {
    struct student class_database[MAX_STUDENTS];  // Our filing cabinet
    int student_count = 0;  // How many students we have so far
    int choice;  // User's menu choice

    do {
        // Show the menu - like a reception desk with options
        printf("\n1. Add student\n2. Print all\n3. Class average\n0.
```

```
Exit\n");
        printf("Choice: ");
        scanf("%d", &choice);

        switch (choice) {  // Handle each menu option
            case 1:  // Add a new student
                if (student_count < MAX_STUDENTS) {  // Check if we have
room
                    input_student(&class_database[student_count]);  // Get
student info
                    student_count++;  // Count this new student
                } else {
                    printf("Database full!\n");  // No more room
                }
                break;
            case 2:  // Show all students
                print_all_students(student_count, class_database);
                break;
            case 3:  // Calculate and show class average
                if (student_count > 0) {  // Only if we have students
                    double avg = calculate_class_average(student_count,
class_database);
                    printf("Class average GPA: %.2lf\n", avg);
                }
                break;
        }
    } while (choice != 0);  // Keep going until user chooses to exit

    return 0;
}

// Get student information from user - like filling out a registration
form
void input_student(struct student *s) {
    printf("Enter student ID: ");
    scanf("%d", &s->id);
    printf("Enter name initial: ");
    scanf(" %c", &s->name_initial);  // Space before %c skips whitespace
    printf("Enter GPA: ");
    scanf("%lf", &s->gpa);
    printf("Enter year: ");
    scanf("%d", &s->year);
}

// Display one student's information - like showing a student record
void print_student(struct student s) {
    printf("ID: %d, Name: %c, GPA: %.2lf, Year: %d\n",
           s.id, s.name_initial, s.gpa, s.year);
}

// Show all students - like printing a class roster
void print_all_students(int count, struct student database[]) {
    printf("\n=== Student Database ===\n");
    for (int i = 0; i < count; i++) {  // Go through each student
```

```
            printf("%d. ", i + 1);  // Student number
            print_student(database[i]);  // Show this student's info
        }
    }

    // Calculate class average GPA - like finding the overall class
    performance
    double calculate_class_average(int count, struct student database[]) {
        double sum = 0.0;  // Start with zero
        for (int i = 0; i < count; i++) {  // Add up all GPAs
            sum += database[i].gpa;
        }
        return sum / count;  // Divide by number of students
    }
```

You're a university registrar managing student records. You can add new students, view all records, and calculate class statistics. The system is organized and efficient, just like a real university database.

Challenge: Add a function to find the student with the highest GPA.

Matrix Operations: The Mathematical Assistant

```
#include <stdio.h>
#define ROWS 3
#define COLS 3

// Function prototypes - our mathematical toolkit
void input_matrix(int matrix[ROWS][COLS]);
void print_matrix(int matrix[ROWS][COLS]);
void add_matrices(int a[ROWS][COLS], int b[ROWS][COLS], int result[ROWS]
[COLS]);
int find_matrix_max(int matrix[ROWS][COLS]);
void print_diagonal(int matrix[ROWS][COLS]);

int main(void) {
    int matrix_a[ROWS][COLS];  // First matrix
    int matrix_b[ROWS][COLS];  // Second matrix
    int sum_matrix[ROWS][COLS];  // Result of addition

    printf("Enter matrix A:\n");
    input_matrix(matrix_a);  // Get first matrix from user

    printf("Enter matrix B:\n");
    input_matrix(matrix_b);  // Get second matrix from user

    printf("\nMatrix A:\n");
    print_matrix(matrix_a);  // Show first matrix

    printf("\nMatrix B:\n");
    print_matrix(matrix_b);  // Show second matrix
```

```
    add_matrices(matrix_a, matrix_b, sum_matrix);  // Add the matrices
    printf("\nA + B:\n");
    print_matrix(sum_matrix);  // Show the result

    printf("\nMaximum in A: %d\n", find_matrix_max(matrix_a));  // Find
highest value

    printf("\nDiagonal of A: ");
    print_diagonal(matrix_a);  // Show diagonal elements

    return 0;
}

// Get matrix input from user - like filling out a spreadsheet
void input_matrix(int matrix[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {      // For each row
        for (int j = 0; j < COLS; j++) {  // For each column in that row
            printf("Enter element [%d][%d]: ", i, j);
            scanf("%d", &matrix[i][j]);
        }
    }
}

// Display matrix nicely - like formatting a spreadsheet for printing
void print_matrix(int matrix[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {      // For each row
        for (int j = 0; j < COLS; j++) {  // For each column
            printf("%4d ", matrix[i][j]);  // Print with nice spacing
        }
        printf("\n");  // New line after each row
    }
}

// Add two matrices - like adding corresponding cells in two spreadsheets
void add_matrices(int a[ROWS][COLS], int b[ROWS][COLS], int result[ROWS]
[COLS]) {
    for (int i = 0; i < ROWS; i++) {      // For each row
        for (int j = 0; j < COLS; j++) {  // For each column
            result[i][j] = a[i][j] + b[i][j];  // Add corresponding
elements
        }
    }
}

// Find maximum value in matrix - like finding the highest number in a
spreadsheet
int find_matrix_max(int matrix[ROWS][COLS]) {
    int max = matrix[0][0];  // Start with first element
    for (int i = 0; i < ROWS; i++) {      // Check each row
        for (int j = 0; j < COLS; j++) {  // Check each column
            if (matrix[i][j] > max) {     // If we find a bigger number
                max = matrix[i][j];       // Update our record
            }
        }
```

```
    }
    return max;  // Return the highest value found
}

// Print diagonal elements - like reading the diagonal of a square
void print_diagonal(int matrix[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {  // For each row
        printf("%d ", matrix[i][i]);  // Print element where row = column
    }
    printf("\n");
}
```

You're a mathematical assistant helping with matrix operations. You can input matrices, add them together, find maximum values, and extract special elements like diagonals. This is the foundation of many scientific and engineering calculations.

Exercise: What would the diagonal of this matrix be?

```
1  2  3
4  5  6
7  8  9
```

Summary: The Professional Programming Toolkit

Key Concepts Table

Concept	What It Does	Real-World Analogy	Why It Matters
Functions	Break problems into manageable pieces	Specialized workers in a factory	Code reuse, organization, debugging
Style	Make code readable and maintainable	Clear handwriting and organization	Collaboration, debugging, professionalism
Arrays	Store collections of related data	Filing cabinets and spreadsheets	Efficient data organization and processing
2D Arrays	Model grids and tables	Game boards and spreadsheets	Real-world data representation

The Professional Programming Mindset

Professional programming is about more than just making code work - it's about making code that others can understand, modify, and maintain. Functions help you organize your thoughts into logical pieces, good style makes your intentions clear, and arrays help you work with real-world data efficiently. These skills transfer to every programming language and every programming job you'll ever have.

Final Discussion: How do these concepts work together? How might you use functions, arrays, and good style to solve a real problem you face?

These concepts form the foundation of professional software development, enabling you to write code that's not just functional, but elegant, maintainable, and collaborative.