Programming in C (DI03011011)

Complete Lecture Deck – All 30 Lectures

Milav Dabgar

Lecturer, GP Palanpur

Table of Contents I

1. Lecture 1.1: Algorithms and writing steps
2. Lecture 1.2: Flowcharts definition and symbols
3. Lecture 1.3: Structure of C program
4. Lecture 1.4: Character set, tokens, keywords, variables
5. Lecture 1.5: Data Types in C
6. Lecture 2.1: Types of Operators: Arithmetic, relational, assignment
7. Lecture 2.2: Types of Operators: Logical, conditional, increment, decrement

Table of Contents II

8 Lecture 2.3: Operator Precedence, Associativity and Evaluation

9 Lecture 2.4: Programming exercise based on operators

10 Lecture 2.5: Input and Output Functions

11 Lecture 3.1: Introduction to decision control and if, if-else

12 Lecture 3.2: if-else-if ladder and nested if-else

13 Lecture 3.3: switch case statement

14 Lecture 3.4: Introduction to branching and while loop

15 Lecture 3.5: Do-While Loop

Table of Contents III

16 Lecture 3.6: For Loop

17 Lecture 3.7: Nested for loop

18 Lecture 3.8: break, continue, goto statements

19 Lecture 4.1: Introduction to an Array & 1D Array

20 Lecture 4.2: 1D Array Memory and Display

21 Lecture 4.3: Programming exercises based on 1D array

22 Lecture 4.4: Two-dimensional array declaration

23 Lecture 4.5: Two-dimensional array memory and display

Table of Contents IV

24 Lecture 4.6: Concept of Pointer

25 Lecture 5.1: Introduction to String

26 Lecture 5.2: Standard Library String Functions

27 Lecture 5.3: Introduction to structures

28 Lecture 5.4: Introduction to Function

29 Lecture 5.5: Library Functions (Math & Character)

30 Lecture 5.6: Introduction to File Operations

Algorithms and Writing Steps

Welcome to the Basics of C Programming!
Unit 1: Basics of C Programming
Lecture 1: Algorithms, Steps, and Flowcharts

Agenda for Today

1. Definition of an Algorithm
2. Characteristics of a Good Algorithm
3. Steps to Write an Algorithm
4. Introduction to Flowcharts and Symbols
5. Example 1: Addition of Two Numbers
6. Example 2: Even or Odd Number
7. Summary and Next Steps

What is an Algorithm?

An algorithm is a step-by-step procedure to solve a specific problem.
It is written in simple, human-readable language (often called pseudo-code).
Algorithms are strictly independent of any specific programming language.
Think of it as a recipe: it provides exact instructions to go from raw ingredients (inputs) to the final dish (output).

Characteristics of a Good Algorithm

1. **Finiteness:** The algorithm must terminate after a finite number of steps.
2. **Definiteness:** Each step must be clearly and precisely defined with no ambiguity.
3. **Input:** An algorithm must have zero or more well-defined inputs.
4. **Output:** It must produce at least one output (the solution to the problem).
5. **Effectiveness:** Every step must be basic enough to be carried out practically and exactly.

Steps for Writing an Algorithm

Step 1: Understand the Problem. Read the requirement carefully.
Step 2: Identify Inputs. Determine what data is needed from the user.
Step 3: Identify Outputs. Determine what the final result should be.
Step 4: Design the Logic. Formulate the mathematical or logical steps to transform inputs into outputs.
Step 5: Write the Steps. Write them down starting with 'START' and ending with 'STOP'.
Step 6: Test/Dry Run. Mentally execute the steps with dummy data to verify.

What is a Flowchart?

A flowchart is a graphical or visual representation of an algorithm.
It uses standardized geometric shapes to denote different types of instructions.
Arrows (flowlines) connect the shapes to show the flow of execution or control.
Flowcharts make it easier to understand complex logic, loops, and conditional statements at a glance.

Basic Flowchart Symbols

Oval (Terminal): Represents the START and STOP points.
Parallelogram (Input/Output): Used for reading inputs (e.g., READ A) or printing outputs (e.g., PRINT Sum).
Rectangle (Process): Used for calculations and assignments (e.g., Sum = A + B).
Diamond (Decision): Used for conditions (e.g., Is A ¿ B?). It always has two outgoing paths: Yes/True and No/False.
Arrows (Flowlines): Show the direction of the process flow.

Algorithm: Sum of Two Numbers

Problem: Find the sum of two numbers provided by the user.
Algorithm Steps:
1. START
2. READ first number as A
3. READ second number as B
4. CALCULATE Sum = A + B
5. PRINT Sum
6. STOP

Flowchart: Sum of Two Numbers

Let's visualize the previous algorithm:

- **Oval:** START
- **Arrow down**
- **Parallelogram:** Input A, B
- **Arrow down**
- **Rectangle:** Sum $=$ A $+$ B
- **Arrow down**
- **Parallelogram:** Print Sum
- **Arrow down**
- **Oval:** STOP

Translating to C Code: Sum of Two Numbers

```c
#include <stdio.h>

int main() {
    int a, b, sum; // Step 2 & 3 preparation

    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b); // Step 2 & 3: Read inputs

    sum = a + b; // Step 4: Calculate

    printf("Sum = %d\n", sum); // Step 5: Print

    return 0; // Step 6: STOP
}
```

Algorithm: Check Even or Odd Number

Problem: Check if a given number is even or odd.
Algorithm Steps:
1. START
2. READ number as N
3. IF (N modulo 2 == 0) THEN
4. PRINT "Number is Even"
5. ELSE
6. PRINT "Number is Odd"
7. END IF
8. STOP

Flowchart: Check Even or Odd Number

- **Oval:** START
- **Parallelogram:** Read N
- **Diamond:** Is N % 2 == 0?
- **Arrow Right (Yes):** Parallelogram -¿ Print "Even"
- **Arrow Down (No):** Parallelogram -¿ Print "Odd"
- Both paths converge to:
- **Oval:** STOP

Translating to C Code: Even or Odd

```c
#include <stdio.h>

int main() {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);

    // Step 3: Decision Diamond
    if (n % 2 == 0) {
        printf("Number is Even\n"); // Yes Path
    } else {
        printf("Number is Odd\n");  // No Path
    }

    return 0;
}
```

Algorithm & Flowchart: Area of a Circle

Algorithm:

1. START
2. READ Radius as R
3. CALCULATE Area = 3.14159 R R
4. PRINT Area
5. STOP

Flowchart Logic:

Oval(START) -¿ Parallelogram(Input R) -¿ Rectangle(Area = 3.14159 R R) -¿ Parallelogram(Output Area) -¿ Oval(STOP)

C Code: Area of a Circle

```c
#include <stdio.h>

int main() {
    float r, area; // using float for decimal numbers

    printf("Enter radius: ");
    scanf("%f", &r);

    area = 3.14159 * r * r;

    printf("Area of circle is: %f\n", area);

    return 0;
}
```

Lecture Summary

Algorithms are language-independent, step-by-step instructions to solve a problem.
Good algorithms are finite, definite, and effective.
Flowcharts are graphical representations of algorithms using standardized symbols.
Ovals = Start/Stop, Parallelograms = Input/Output, Rectangles = Processes, Diamonds = Decisions.
Translating logic to C involves mapping these steps to C syntax.

Next Lecture Preview

In the next lecture, we will dive deeper into the building blocks of C programming:

1. **Character Set and Tokens:** What makes up a C program?
2. **Data Types:** `int`, `float`, `char` (How C stores information)
3. **Constants and Variables:** Naming rules and storage

Lecture 2: Flowcharts - Definition, Symbols, and Examples
Understanding how to visually represent algorithms before writing C code.

Agenda for Today

Recap of Algorithms
Definition of a Flowchart
Advantages of Using Flowcharts
Standard Flowchart Symbols (Terminal, Input/Output, Process, Decision, Flowlines)
Guidelines for Drawing Flowcharts
Examples: Simple Arithmetic, Conditional Logic
Summary & Next Steps

Recap: What is an Algorithm?

An **algorithm** is a step-by-step procedure to solve a specific problem.
Characteristics of a good algorithm:

- **Unambiguous**: Each step must be clear and precise.
- **Finiteness**: It must terminate after a finite number of steps.
- **Input / Output**: It should take zero or more inputs and produce at least one output.

While algorithms are written in plain English, they can be hard to visualize for complex problems.

1.1.2 Definition of Flowchart

A **flowchart** is a graphical or visual representation of an algorithm.
It uses different shapes to represent different types of actions or steps in a process.
Arrows (flowlines) connect the shapes to show the sequence or 'flow of control'.
It acts as a blueprint for the programmer to write the actual code.

Why Use Flowcharts?

Better Communication: Visuals are easier to understand and discuss with a team.
Effective Analysis: Helps identify logical errors or missing steps before coding begins.
Proper Documentation: Serves as permanent documentation for how a program works.
Efficient Coding: Translating a flowchart into C code is straightforward because the logic is already solved.

Standard Symbols: Terminal and I/O

Oval (Terminal Symbol):

- Used to indicate the **START** and **STOP** (or END) of a flowchart.
- Every flowchart must have one START and at least one STOP.

Parallelogram (Input/Output Symbol):

- Used for any input (reading data) or output (printing results).
- Example text inside: `Read A, B` or `Print Sum`.

Standard Symbols: Process and Decision

Rectangle (Processing Symbol):

- Used for arithmetic operations and data manipulation.
- Example text inside: `Sum = A + B` or `Count = Count + 1`.

Diamond (Decision Symbol):

- Used to represent a condition or question that has two possible answers (usually Yes/True or No/False).
- It creates a branch in the flow.
- Example text inside: `Is A > B?` or `N % 2 == 0`.

Standard Symbols: Flowlines and Connectors

Arrows (Flowlines):

- Indicate the exact sequence in which the instructions are executed.
- Generally, flow is from top to bottom and left to right.

Circle (On-page Connector):

- Used to connect different parts of a complex flowchart on the same page without drawing long, confusing lines.
- Typically contains a letter or number (e.g., 'A') to match the connection point.

Guidelines for Drawing Flowcharts

1. Ensure the flowchart has a logical start and end.
2. The general flow should be from top to bottom or left to right.
3. Avoid intersecting flowlines; use connectors if necessary.
4. Only one flowline should exit a process symbol (rectangle).
5. Exactly two flowlines (Yes/No) should exit a decision symbol (diamond).
6. Write clear and concise instructions inside the symbols.

1.1.3 Example 1: Add Two Numbers

Algorithm:

1. Start
2. Read two numbers, A and B
3. Calculate Sum = A + B
4. Print Sum
5. Stop

Flowchart Mental Map:

- Oval: `START` ↓
- Parallelogram: `Read A, B` ↓
- Rectangle: `Sum = A + B` ↓
- Parallelogram: `Print Sum` ↓
- Oval: `STOP`

Example 2: Check Even or Odd Number

Algorithm:

1. Start
2. Read an integer N
3. If (N % 2 == 0) then goto step 4, else goto step 5
4. Print 'N is Even' and goto step 6
5. Print 'N is Odd'
6. Stop

Flowchart Note: The Decision Diamond is used here.

- Inside Diamond: `Is N % 2 == 0?`
- `Yes` arrow → Parallelogram: `Print Even`
- `No` arrow → Parallelogram: `Print Odd`

Example 3: Find Largest of Three Numbers

Algorithm Snippet:

1. Read A, B, C
2. Is A ¿ B?
   - YES: Is A ¿ C? (If YES, print A is max; If NO, print C is max)
   - NO: Is B ¿ C? (If YES, print B is max; If NO, print C is max)

Flowchart Structure:

- Requires nested decision diamonds.
- First diamond checks `A > B`.
- Its 'Yes' branch leads to a second diamond `A > C`.
- Its 'No' branch leads to a third diamond `B > C`.

Summary of Lecture 2

Flowcharts are essential visual tools for algorithm design.
Standard symbols must be used: Oval (Terminal), Parallelogram (I/O), Rectangle (Process), and Diamond (Decision).
Arrows represent the flow of control.
Drawing a flowchart helps prevent logical errors before writing C code.
We practiced converting basic algorithms (like Sum and Even/Odd) into flowchart structures.

Next Lecture Preview

Now that we know how to design logic, we will start writing actual code!
Next time, we will cover:

- History and Importance of C language
- Structure of a standard C Program
- C Tokens (Keywords, Identifiers, Constants)
- Writing our very first C program!

Structure of a C Program

Understanding the fundamental building blocks and sections of every C program.

Agenda for Today

- General Structure of a C Program
- Documentation Section
- Link Section
- Definition Section
- Global Declaration Section
- The main() Function
- Subprogram Section
- Example Program Breakdown

Overview of C Program Structure

A C program can be viewed as a group of building blocks called functions.
A typical C program consists of the following sections:

1. **Documentation Section**
2. **Link Section**
3. **Definition Section**
4. **Global Declaration Section**
5. **main() Function Section**
6. **Subprogram Section**

1. Documentation Section

The Documentation section consists of a set of **comments** giving the name of the program, the author, and other details.

- It helps in understanding what the program does.
- Comments are ignored by the compiler.
- **Single-line comments:** `// comment here`
- **Multi-line comments:** `/ comment here /`

2. Link Section

The Link section provides instructions to the compiler to link functions from the system library.

- Also known as the **Preprocessor Directives** section.
- Typically includes header files.
- Syntax: `#include <filename.h>`
- Example: `#include <stdio.h>` allows us to use `printf()` and `scanf()`.

3. Definition Section

The Definition section defines all symbolic constants.

- Used to assign a constant value to a name.
- Syntax: `#define CONSTANT_NAME value`
- Example: `#define PI 3.14159`

Whenever `PI` is used in the program, it is replaced by `3.14159` before compilation.

4. Global Declaration Section

Variables that are used in more than one function are called **global variables**.

- Declared outside of all functions, usually before `main()`.
- They can be accessed and modified by any function in the program.
- This section also contains user-defined function declarations (prototypes).

5. The main() Function Section

Every C program must have one `main()` function. Execution always begins here.
Contains two parts:
1. **Declaration Part:** Declares local variables used within `main()`.
2. **Executable Part:** Contains the statements that perform the actual task.

```c
int main() {
    // Declaration part
    int a = 10;
    // Executable part
    printf("%d", a);
    return 0;
}
```

6. Subprogram Section

Contains all **user-defined functions** that are called in the `main()` function.

- User-defined functions are generally placed immediately after the `main()` function, although they may appear in any order.
- They help in breaking down a complex problem into smaller, manageable pieces (modularity).

Example: Putting it all together

```c
/* Documentation: Program to calculate area of circle */
#include <stdio.h>          // Link section
#define PI 3.14             // Definition section
float area;                 // Global declaration
void calcArea(float r);     // Function prototype
int main() {                // main() function
    float radius = 5.0;
    calcArea(radius);
    printf("Area: %f\n", area);
    return 0;
}
void calcArea(float r) { // Subprogram section
    area = PI * r * r;
}
```

Syntax Rules in C

When writing the structure, you must follow basic syntax rules:

- **Case Sensitivity:** C is highly case-sensitive. `printf` is different from `PRINTF`.
- **Semicolons:** Every executable statement must end with a semicolon `;`.
- **Braces:** Code blocks for functions, loops, and conditionals must be enclosed in curly braces {}.
- **Main Return:** It is standard practice for `main()` to return an integer (`return 0;`), indicating successful execution.

Why Follow This Structure?

- **Readability:** A standardized structure makes it easier for others (and yourself) to understand the code.
- **Maintainability:** Easier to find bugs or update features when code is organized logically.
- **Compilation Efficiency:** The compiler expects directives before code, and function declarations before they are called.

Summary

- A C program consists of 6 primary sections: Documentation, Link, Definition, Global Declaration, main(), and Subprogram.
- `main()` is the mandatory starting point of execution.
- `#include` is used to link library functions.
- Comments are essential for understanding the logic behind the code.
- Strict syntax rules (like semicolons and case-sensitivity) must be followed.

Next Lecture Preview

In our next lecture, we will cover:

- The life cycle of a C program.
- How a text file becomes an executable program.
- Preprocessing, Compilation, Assembly, and Linking steps.

Unit 1: Basics of C Programming
Understanding the basic building blocks of C code

Agenda

C Character Set
What are C Tokens?
Keywords and Identifiers
Constants in C
Understanding Variables
Rules for Naming Variables

The C Character Set

The character set is the set of characters allowed in a C program.
Letters: Uppercase (A-Z) and Lowercase (a-z). C is case-sensitive.
Digits: Decimal digits 0 through 9.
Special Characters: Symbols like ~@ # % ^& * ( ) _ - + = { } [ ] ; : ' " , . ¡ ¿ / ? \— !
White Spaces: Blank space, horizontal tab, carriage return, new line, and form feed.

Introduction to C Tokens

In a C program, the smallest individual units are known as C Tokens.
A token is to a C program what a word is to an English sentence.
The C compiler breaks the source code into tokens before processing it.
Tokens are separated by white spaces.

Types of C Tokens

C has six distinct types of tokens:

1. Keywords: e.g., int, while
2. Identifiers: e.g., main, amount
3. Constants: e.g., 10, 15.5
4. Strings: e.g., "Hello"
5. Special Symbols: e.g., { } [ ]
6. Operators: e.g., + - * /

Keywords in C

Keywords are reserved words that have standard, predefined meanings in C.
Their meaning cannot be changed; they cannot be used as variable names.
All keywords must be written in lowercase.
Standard C (C89) has 32 keywords. Examples: int, float, char, if, else, for, while, return, void.

Identifiers

Identifiers are user-defined names given to entities like variables, arrays, and functions.
They are used to identify a particular item in the program.
While keywords are defined by the C language, identifiers are defined by the programmer.
Example: In 'int age;', 'int' is a keyword, and 'age' is an identifier.

Rules for Naming Identifiers and Variables

1. First character must be an alphabet (A-Z or a-z) or an underscore (_).
2. Cannot start with a digit. (e.g., '1stRank' is invalid, 'rank1' is valid).
3. Only letters, digits, and underscores are allowed. No other special characters.
4. Cannot be a keyword (e.g., 'int float;' is invalid).
5. No spaces are allowed (e.g., 'first name' is invalid, use 'first_name').
6. Case sensitive: 'Age' and 'age' are different variables.

Constants in C

Constants refer to fixed values that the program may not alter during its execution.
They are also called literals.
Types of constants include:

- Integer Constants: 10, -50, 0
- Real/Floating-point Constants: 3.14, -0.99
- Character Constants: 'A', 'z', '5' (enclosed in single quotes)
- String Constants: "Hello World" (enclosed in double quotes)

What is a Variable?

A variable is a name given to a storage area in the computer's memory.
Unlike constants, the value stored in a variable can be changed during program execution.
Each variable in C has a specific data type, which determines the size and layout of its memory.
Variables act as containers for storing data.

Declaring and Initializing Variables

Declaration tells the compiler the variable's name and type.
Initialization assigns an initial value to the variable.

Putting it all together: Tokens in Action

Let's identify the tokens in a simple program.

Common Mistakes with Variables

1. Using uninitialized variables (leads to garbage values).
2. Using keywords as variable names (e.g., int float = 10;).
3. Forgetting the semicolon (;) at the end of declaration.
4. Redeclaring the same variable in the same scope.
5. Case errors (e.g., declaring 'int Sum;' but using 'sum = 5;').

Summary

C code is built from the C Character Set.
Tokens are the smallest individual units (Keywords, Identifiers, Constants, Strings, Symbols, Operators).
Keywords are reserved; Identifiers are user-defined names.
Variables are named memory locations whose values can change.
Strict rules apply when naming variables (start with letter/_, no spaces, no special chars).

Next Lecture

Topic: Data Types in C

- Understanding Predefined Data Types (int, char, float, double)
- Size and range of data types
- Format specifiers used in C

Data Types in C

Unit 1: Basics of C Programming
Lecture 5

Previous Lecture Recap

- Constants in C (Integer, Real, Character, String)
- Variables and rules for naming them
- Variable declaration and initialization
- Introduction to keywords

Agenda

1. What is a Data Type?
2. Classification of Data Types in C
3. Predefined (Primary) Data Types (`int`, `char`, `float`, `double`)
4. `sizeof` Operator and Data Ranges
5. User-Defined Data Types (`typedef` and `enum`)

What is a Data Type?

A **Data Type** specifies two things about a variable:
1. The **type** of data it can hold (e.g., numbers, text).
2. The **size** (in bytes) it will occupy in memory.

Without data types, the C compiler wouldn't know how much memory to allocate or how to interpret the binary bits stored in that memory.

Classification of Data Types

C supports several categories of data types:

1. **Predefined (Primary/Fundamental) Data Types**
   - Built into the language (e.g., `int`, `char`, `float`, `double`)
2. **User-Defined Data Types**
   - Created by the programmer (e.g., `typedef`, `enum`)
3. **Derived Data Types**
   - Derived from primary types (e.g., Arrays, Pointers, Structures - covered later)
4. **Void Data Type**
   - Represents 'no value' or 'empty'

Predefined Data Types: Integer (`int`)

- Used to store whole numbers (without decimal points).
- **Keyword:** `int`
- **Size:** Typically 2 or 4 bytes (depends on the compiler/architecture).
- **Format Specifier:** `%d`

Modifiers can be applied:

- `short int`
- `long int`
- `unsigned int` (only positive numbers)

Predefined Data Types: Floating-Point

- Used to store real numbers (numbers with a fractional/decimal part).

1. `float` (Single Precision)
   - Size: 4 bytes
   - Precision: ~6 decimal places
   - Format Specifier: `%f`
2. `double` (Double Precision)
   - Size: 8 bytes
   - Precision: ~15 decimal places
   - Format Specifier: `%lf`

Predefined Data Types: Character (`char`)

- Used to store a single character (letter, digit, or symbol).
- **Keyword:** `char`
- **Size:** 1 byte
- **Format Specifier:** %c

Note: Internally, C stores characters as integers based on their ASCII values.

Code Example: Predefined Types

```c
#include <stdio.h>
int main() {
    int age = 20;
    float height = 5.9f;
    double salary = 45000.50;
    char grade = 'A';
    printf("Age: %d\n", age);
    printf("Height: %.1f ft\n", height);
    printf("Salary: %.2lf\n", salary);
    printf("Grade: %c\n", grade);
    return 0;
}
```

Finding Size with `sizeof()`

C provides a built-in operator called `sizeof()` to determine the exact memory size (in bytes) of a data type or variable on your specific machine.

```c
#include <stdio.h>
int main() {
    printf("Size of int: %lu bytes\n", sizeof(int));
    printf("Size of char: %lu bytes\n", sizeof(char));
    printf("Size of float: %lu bytes\n", sizeof(float));
    return 0;
}
```

User-Defined Data Types

Sometimes predefined types are not descriptive enough. C allows programmers to define their own data types for better code readability and maintenance.
We will cover two main tools for this:
1. `typedef` (Type Definition)
2. `enum` (Enumeration)

The typedef Keyword

typedef is used to give a new, meaningful name (alias) to an existing data type.

Syntax:

```
typedef existing_type new_type_name;
```

Example:

```
typedef unsigned long int UINT32;
typedef float Distance;
UINT32 accountBalance = 1500000;
Distance distanceToMoon = 384400.0;
```

The enum Keyword (Enumeration)

enum is a user-defined data type consisting of named integer constants. It helps in assigning names to integral constants, making a program easier to read.

Syntax:

```
enum enum_name { value1, value2, value3 };
```

By default, value1 is 0, value2 is 1, and so on.

Code Example: Using enum

```c
#include <stdio.h>
enum WeekDay {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
};
int main() {
    enum WeekDay today = WEDNESDAY;
    printf("Day number: %d\n", today);
    // Output will be: Day number: 3
    return 0;
}
```

Summary

- **Data types** determine the size and nature of data a variable can hold.
- **Primary types**: `int` (integers), `float`/`double` (real numbers), `char` (characters).
- `sizeof()` helps find memory size allocated on your system.
- `typedef` allows you to alias existing types for readability.
- `enum` lets you create a set of named integer constants for clarity.

Next Lecture Preview

Operators and Expressions in C

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- How to perform complex calculations in C

Types of Operators in C

Unit 2: Operator & Expressions and input/output Functions
Lecture 6

Agenda

1. Introduction to Operators
2. Arithmetic Operators
3. Relational Operators
4. Assignment Operators
5. Increment and Decrement Operators
6. Summary and Next Steps

Introduction to Operators

An **operator** is a symbol that tells the compiler to perform specific mathematical or logical functions.
C is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Increment and Decrement Operators
- Conditional (Ternary) Operator
- Bitwise Operators

Operators act on **operands**. For example, in `a + b`, + is the operator, while `a` and `b` are operands.

Arithmetic Operators

Arithmetic operators perform standard mathematical operations on numerical values.

- + (Addition): Adds two operands
- - (Subtraction): Subtracts second operand from the first
- * (Multiplication): Multiplies both operands
- / (Division): Divides numerator by denominator
- % (Modulus): Remainder of after an integer division

Note: The modulus operator (%) can only be used with integer operands.

Example: Arithmetic Operators

```c
#include <stdio.h>
int main() {
    int a = 10, b = 3;
    int add, sub, mul, div, mod;
    add = a + b;
    sub = a - b;
    mul = a * b;
    div = a / b;
    mod = a % b;
    printf("Addition: %d\n", add);          /* 13 */
    printf("Subtraction: %d\n", sub);       /* 7 */
    printf("Multiplication: %d\n", mul);    /* 30 */
    printf("Division: %d\n", div);          /* 3 */
    printf("Modulus: %d\n", mod);           /* 1 */
    return 0;
}
```

Relational Operators

Relational operators compare two operands. They return `1` (true) if the relationship is true, and `0` (false) if it is false.

- == (Equal to)
- != (Not equal to)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)
- <= (Less than or equal to)

These are primarily used in decision making and loops (which we will cover in Unit 3).

Example: Relational Operators

```c
#include <stdio.h>
int main() {
    int a = 5, b = 10;
    printf("a == b is %d\n", a == b); /* 0 */
    printf("a != b is %d\n", a != b); /* 1 */
    printf("a > b is %d\n", a > b);   /* 0 */
    printf("a < b is %d\n", a < b);   /* 1 */
    printf("a >= b is %d\n", a >= b); /* 0 */
    printf("a <= b is %d\n", a <= b); /* 1 */
    return 0;
}
```

Assignment Operators

Assignment operators are used to assign a value to a variable. The most common is the simple assignment operator =.
C also supports **compound assignment operators** (shorthand notation):

- = (Simple assignment: `c = a + b`)
- += (Add and assign: `c += a` is equivalent to `c = c + a`)
- -= (Subtract and assign)
- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulus and assign)

Example: Assignment Operators

```c
#include <stdio.h>
int main() {
    int a = 10;
    int c;
    c = a;        /* c is 10 */
    printf("c = %d\n", c);
    c += a;       /* c = c + a -> 10 + 10 = 20 */
    printf("c += a -> %d\n", c);
    c -= a;       /* c = c - a -> 20 - 10 = 10 */
    printf("c -= a -> %d\n", c);
    c *= a;       /* c = c * a -> 10 * 10 = 100 */
    printf("c *= a -> %d\n", c);
    return 0;
}
```

Increment and Decrement Operators

C provides two special unary operators for incrementing and decrementing variables by 1.

- ++ (Increment): Increases integer value by one.
- -- (Decrement): Decreases integer value by one.

They can be used in two forms:
1. **Prefix** (++a or --a): Value is modified first, then used in the expression.
2. **Postfix** (a++ or a--): Value is used in the expression first, then modified.

Example: Prefix vs Postfix

```c
#include <stdio.h>
int main() {
    int x = 5, y;
    /* Postfix increment */
    y = x++;
    printf("Postfix: x = %d, y = %d\n", x, y);
    /* Output: x = 6, y = 5 */
    x = 5; /* Reset x */
    /* Prefix increment */
    y = ++x;
    printf("Prefix: x = %d, y = %d\n", x, y);
    /* Output: x = 6, y = 6 */
    return 0;
}
```

Other Important Operators (Overview)

In addition to Arithmetic, Relational, and Assignment operators, C has:

- **Logical Operators** (`&&`, `||`, `!`): Used to combine multiple relational expressions.
- **Conditional Operator** (`? :`): A shorthand for `if-else` statements. Takes three operands.
- **Bitwise Operators** (`&`, `|`, `^`, `<<`, `>>`): Perform operations at the bit level.

We will explore logical and conditional operators in more detail when we discuss control structures.

Summary

- **Operators** perform actions on operands.
- **Arithmetic operators** (+, −, *, /, %) are for math. Remember % needs integers.
- **Relational operators** (==, >, <) compare values and return 1 (true) or 0 (false).
- **Assignment operators** (=, +=, etc.) store values into variables right-to-left.
- **Increment/Decrement** (++, −−) change a variable by 1, with distinct prefix and postfix behaviors.

Next Lecture Preview

In the next lecture, we will cover:

- Logical Operators (`&&`, `||`, `!`)
- Conditional Operator (`?  :`)
- Expressions and their evaluation
- Operator Precedence and Associativity

Lecture 7: Logical, Conditional, Increment & Decrement Operators

Unit 2: Operator & Expressions and Input/Output Functions
GTU Diploma Semester 3 (DI03011011)

Agenda for Today

1. Recap of Arithmetic, Relational, and Assignment Operators
2. Logical Operators (`&&`, `||`, `!`)
3. Short-Circuit Evaluation
4. Conditional (Ternary) Operator (`?:`)
5. Increment (++) and Decrement (−−) Operators
6. Prefix vs. Postfix Evaluation

Recap: Previous Operators

Arithmetic Operators: +, −, ∗, /, %
Relational Operators: <, <=, >, >=, ==, != (Return 1 for true, 0 for false)
Assignment Operators: =, +=, −=, ∗=, /=, %=
These form the basic building blocks for calculations and simple comparisons.

Introduction to Logical Operators

Logical operators are used to combine multiple relational expressions or conditions.
They return `1` (true) or `0` (false) depending on the logic.

- **Logical AND (`&&`):** Returns 1 if ALL operands are true (non-zero).
- **Logical OR (`||`):** Returns 1 if ANY operand is true (non-zero).
- **Logical NOT (`!`):** Reverses the logical state of its operand (turns non-zero to 0, and 0 to 1).

Logical AND (&&) Operator

Truth Table for `A && B`:

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

Logical OR (——) Operator

Truth Table for `A || B`:

A	B	A —— B
0	0	0
0	1	1
1	0	1
1	1	1

Logical NOT (!) Operator

Truth Table for `!A`:

A	!A
0	1
1	0

Short-Circuit Evaluation

C optimizes logical expressions by evaluating them from left to right and stopping as soon as the result is known.

- **For `&&`:** If the left side is `0` (false), the entire expression must be false. The right side is **not evaluated**.
- **For `||`:** If the left side is `1` (true), the entire expression must be true. The right side is **not evaluated**.

Conditional (Ternary) Operator (?:)

The conditional operator is the only **ternary operator** in C (it takes three operands).
It provides a shorthand way of writing an `if-else` statement.
Syntax:

```
condition ? expression1 : expression2;
```

- If `condition` is true (non-zero), `expression1` is evaluated and becomes the result.
- If `condition` is false (zero), `expression2` is evaluated and becomes the result.

Conditional Operator Example

Let's find the maximum of two numbers using the ternary operator.

Increment (++) and Decrement (–) Operators

These are unary operators used to increase or decrease the value of a variable by 1.

- **Increment (++):** Adds 1 to the operand (`x++` or `++x` is equivalent to `x = x + 1`).
- **Decrement (--):** Subtracts 1 from the operand (`x--` or `--x` is equivalent to `x = x - 1`).

They can be used in two forms:

1. **Prefix:** `++x` or `--x`
2. **Postfix:** `x++` or `x--`

Prefix vs. Postfix

The difference between prefix and postfix matters when the operator is part of a larger expression.

- **Prefix (`++x`):** Change the value first, then use the new value in the expression.
- **Postfix (`x++`):** Use the current value in the expression first, then change the value afterwards.

Decrement Operator Example

The decrement operator works exactly the same way, but subtracts 1.

Summary

Logical Operators (`&&`, `||`, `!`): Evaluate to 1 or 0 and allow complex condition building.
Short-Circuiting: C stops evaluating logical expressions as soon as the outcome is guaranteed.
Conditional Operator (`?:`): A compact, inline if-else statement.
Increment/Decrement (`++`, `--`): Modify variables by 1. Remember the critical difference between prefix (change then use) and postfix (use then change).

Next Lecture Preview

In our next lecture, we will cover:

- Bitwise Operators (`&`, `|`, `^`, `<<`, `>>`)
- Special Operators (`sizeof`, comma `,`)
- Operator Precedence and Associativity (BODMAS for C!)

Lecture 8: Operator Precedence & Associativity

Welcome to Lecture 8!
Today's focus:

1. Operator Precedence (BODMAS in C)
2. Associativity of Operators (Tie-breakers)
3. Step-by-step Evaluation of Arithmetic Expressions
4. Evaluation of Logical Expressions (Short-circuiting)

Agenda

Introduction to Expressions
What is Operator Precedence?
Precedence Table Overview
What is Associativity?
Left-to-Right vs Right-to-Left Associativity
Evaluation of Arithmetic Expressions
Evaluation of Logical Expressions
Summary & Q&A

Introduction to Expressions

An **expression** is a combination of variables, constants, and operators written according to the syntax of C.
When an expression is executed, it evaluates to a single value.
Example:

```
result = a + b * c;
```

Question: Which operation happens first? Addition (+) or Multiplication (*)?
The C compiler uses rules of **Precedence** and **Associativity** to resolve this.

Operator Precedence

Operator Precedence determines which operator is evaluated first when an expression contains multiple different operators.
Operators with higher precedence are evaluated before operators with lower precedence.
Think of it as the 'BODMAS' or 'PEMDAS' rule for C.
Example:
`10 + 5 * 2`
Here, * has higher precedence than +.
So, `5 * 2 = 10` is evaluated first, then `10 + 10 = 20`.

Common Precedence Levels

Let's look at the basic order (from Highest to Lowest):

1. Parentheses ( ) - Always highest priority!
2. Unary Operators: ++, --, !, &, *
3. Multiplicative: *, /, %
4. Additive: +, -
5. Relational: <, <=, >, >=
6. Equality: ==, !=
7. Logical AND: &&
8. Logical OR: ||
9. Assignment: =, +=, -=, etc.

Associativity of Operators

What happens when an expression has multiple operators of the **SAME** precedence?
Example: `10 / 2 * 5`
Both / and * have the same precedence.
Associativity dictates the direction of evaluation.
Associativity can be:

- **Left-to-Right (L to R)**: Evaluates from the left side.
- **Right-to-Left (R to L)**: Evaluates from the right side.

Associativity Rules

Most operators in C are **Left-to-Right**.
Example of Left-to-Right:
Arithmetic: *, /, %, +, -
`10 / 2  5 => (10 / 2)  5 =¿ 5 * 5 = 25`
Exceptions (Right-to-Left):
1. Unary Operators (!, ++, --, etc.)
2. Assignment Operators (=, +=, etc.)

Example of Right-to-Left Assignment:
`a = b = c = 5; =¿ a = (b = (c = 5));`

Evaluation Example 1

Expression: `x = 5 + 2 * 3 - 8 / 4;`
Step 1: Evaluate * and / (Highest precedence here, L to R)
`2 * 3` becomes 6
`8 / 4` becomes 2
Expression becomes: `x = 5 + 6 - 2;`
Step 2: Evaluate + and - (Same precedence, L to R)
`5 + 6` becomes 11
Expression becomes: `x = 11 - 2;`
Step 3: `11 - 2` becomes 9
Step 4: Assignment = (Lowest precedence here)
`x` is assigned 9.

Code Example: Arithmetic Evaluation

```c
#include <stdio.h>

int main() {
    int result1 = 10 + 5 * 2;
    int result2 = (10 + 5) * 2; // Overriding precedence

    printf("Without parentheses: %d\n", result1);
    printf("With parentheses: %d\n", result2);

    return 0;
}
```

Output:
Without parentheses: 20
With parentheses: 30

Evaluation of Logical Expressions

Logical expressions involve && (AND), || (OR), and ! (NOT).
Precedence: ! ¿ && ¿ ||.
Example:
`a > 5 || b < 3 && c == 10`
Here, && is evaluated BEFORE ||.
So it's treated as: `a > 5 || (b < 3 && c == 10)`
Relational operators (>, <, ==) have higher precedence than logical operators!

Short-Circuit Evaluation

C optimizes logical expressions using **Short-Circuiting**.
For Logical AND (&&):
If the left operand is FALSE (0), the result is FALSE. The right operand is **not evaluated**.
Example: `(5 < 2) && (x++)` =¿ `x++` never runs!
For Logical OR (||):
If the left operand is TRUE (non-zero), the result is TRUE. The right operand is **not evaluated**.
Example: `(5 > 2) || (y++)` =¿ `y++` never runs!

Code Example: Short-Circuiting

```c
#include <stdio.h>

int main() {
    int x = 5, y = 5;

    // && Short-circuit
    int res1 = (0) && (++x > 0);

    // || Short-circuit
    int res2 = (1) || (++y > 0);

    printf("res1 = %d, x = %d\n", res1, x); // x is still 5
    printf("res2 = %d, y = %d\n", res2, y); // y is still 5

    return 0;
}
```

Type Conversion in Expressions

What happens if an expression has mixed data types (e.g., `int` and `float`)?
C automatically performs **Implicit Type Conversion** (Type Promotion).
Rule: Lower data types are promoted to higher data types during evaluation to prevent data loss.
Hierarchy: `char -¿ int -¿ float -¿ double`
Example:
`int a = 5; float b = 2.5;`
`a + b =¿ 5` (int) is promoted to `5.0` (float).
Result is `7.5` (float).

Summary

Precedence dictates the order of operations among different operators.
Associativity dictates the order (L-to-R or R-to-L) when precedence is the same.
Parentheses () have the highest precedence and can override default rules.
Logical operators utilize **short-circuit evaluation** for efficiency.
Expressions with mixed types undergo automatic type promotion.

Next Lecture Preview

Next time, we will cover:

Formatted Input/Output Functions

- Reading data from the user using `scanf()`
- Formatting output using `printf()`
- Understanding format specifiers (`%d`, `%f`, `%c`)
- Managing field width and precision

Programming Exercise Based on Operators

Practical implementations and exercises utilizing various C operators.

Agenda

- Arithmetic Operators: Swapping and Conversions
- Ternary Operator: Finding maximums
- Bitwise Operators: Odd/Even checking & XOR swap
- Increment/Decrement Operators
- Logical & Relational Operators: Leap year logic
- `sizeof` operator applications
- Common operator pitfalls

Exercise 1: Swap Two Numbers (Without Third Variable)

We can swap two integer variables using only addition and subtraction operations. This demonstrates the use of arithmetic and assignment operators without needing a temporary variable.

Exercise 2: Maximum of Two Numbers (Ternary Operator)

The conditional (ternary) operator ? : provides a concise way to perform simple if-else logic. Let's use it to find the larger of two numbers.

Exercise 3: Odd or Even Check (Using Bitwise AND)

While the modulus operator (%) is commonly used to check for odd/even, we can also use the bitwise AND operator (&) for a highly efficient check.

Exercise 4: Understanding Increment and Decrement

A common point of confusion is the difference between prefix (`++x`) and postfix (`x++`) operators. Let's see them in an expression.

Exercise 5: Calculating Simple Interest

This exercise combines multiple arithmetic operators. The formula for simple interest is `(P * R * T) / 100`.

Exercise 6: Leap Year Check (Logical & Relational)

A year is a leap year if it is divisible by 4, but century years must be divisible by 400. This requires logical AND (&&) and OR (||).

Exercise 7: Using the sizeof Operator

The `sizeof` operator is a compile-time operator used to determine the size, in bytes, of a variable or data type.

Exercise 8: Convert Days to Years, Weeks, and Days

This exercise uses division (/) and modulus (%) operators to break down a total number of days into larger units.

Common Pitfalls with Operators

- **Confusing = and ==**: Using assignment (=) instead of equality check (==) in `if` conditions.
- **Integer Division**: `5 / 2` yields 2, not `2.5`. Use `5.0 / 2` for floats.
- **Undefined Behavior**: Modifying a variable multiple times in the same expression (e.g., `a = a++ + ++a;`).
- **Precedence Ignorance**: Not using parentheses when mixing logical and arithmetic operators, leading to unexpected evaluation orders.

Exercise 9: Swapping Using Bitwise XOR

Another way to swap two variables without a third variable is by using the Bitwise XOR (^) operator. This avoids potential integer overflow issues present in the arithmetic approach.

Exercise 10: Uppercase to Lowercase (Bitwise OR)

We can manipulate ASCII values using operators. Adding 32 converts an uppercase ASCII letter to lowercase, which is equivalent to a bitwise OR with 32.

Lecture Summary

- Applied **Arithmetic Operators** for unit conversions and in-place swapping.
- Leveraged the **Ternary Operator** for concise conditional assignments.
- Utilized **Bitwise Operators** for low-level data manipulation (odd/even, XOR swap).
- Clarified the behavior of **Prefix and Postfix Increment/Decrement**.
- Combined **Logical & Relational Operators** to evaluate complex conditions like leap years.
- Discussed common operator pitfalls to avoid logical bugs.

Lecture 10: Input and Output Functions

Unit 2: Operator & Expressions and Input/Output Functions
Topics Covered:

- Introduction to I/O in C
- Unformatted Input/Output: `getch()`, `putch()`, `gets()`, `puts()`
- Formatted Input/Output: `scanf()`, `printf()`

Course: C Programming (DI03011011)

Agenda for Today

- **What is Input/Output (I/O)?**
- **Standard I/O Library (`<stdio.h>`)**
- **Unformatted I/O Functions**
- Character I/O: `getch()`, `putch()`
- String I/O: `gets()`, `puts()`
- **Formatted I/O Functions**
- Formatted Output: `printf()`
- Formatted Input: `scanf()`
- **Summary & Practice**

Introduction to Input and Output

What is I/O?

- **Input:** Providing data to the program (e.g., typing on a keyboard).
- **Output:** The program displaying data to the user (e.g., printing on the screen).

The Standard I/O Header

- In C, standard I/O functions are part of the C Standard Library.
- We must include the header file `<stdio.h>` to use these functions.
- Some older console-specific functions like `getch()` require `<conio.h>` (Console I/O).

Formatted vs. Unformatted I/O

Unformatted I/O Functions

- Deal with characters and strings directly.
- Cannot format the data (like controlling decimal places or padding).
- Examples: `getch()`, `putch()`, `gets()`, `puts()`, `getchar()`, `putchar()`.

Formatted I/O Functions

- Allow you to specify the exact format of the input or output.
- Use format specifiers (like `%d`, `%f`, `%c`).
- Examples: `scanf()`, `printf()`.

Character Input: getch()

The `getch()` Function

- **Purpose:** Reads a single character from the keyboard.
- **Key Feature:** It does *not* echo (display) the character on the screen, and it does *not* wait for the user to press Enter.
- **Header:** Requires `<conio.h>`.

Syntax

```
int getch(void);
```

Note: Often used at the end of programs in older IDEs to keep the output window open until a key is pressed.

Character Output: putch()

The `putch()` Function

- **Purpose:** Writes a single character to the console screen.
- **Header:** Requires `<conio.h>`.
- (Alternatively, `putchar()` from `<stdio.h>` is the standard C equivalent).

Syntax

```
int putch(int c);
```

Example

```
#include <stdio.h>
#include <conio.h>
int main() {
    char c = 'A';
    putch(c);
    return 0;
}
```

String Input: gets()

The `gets()` Function

- **Purpose:** Reads a full line of text (a string) from the keyboard until the Enter key (newline) is pressed.
- **Advantage:** Unlike `scanf("%s")`, it can read strings containing spaces.

Syntax

```
char *gets(char *str);
```

¿ **WARNING:** `gets()` is considered dangerous because it does not check the size of the array, which can lead to buffer overflows. It was removed in later C standards (C11), but is still taught for historical context.

String Output: puts()

The `puts()` Function

- **Purpose:** Writes a string to the console screen and automatically appends a newline character (`\n`) at the end.
- **Header:** `<stdio.h>`

Syntax

```
int puts(const char *str);
```

Example

```
#include <stdio.h>
int main() {
    char greeting[] = "Hello, GTU Students!";
    puts(greeting); // Automatically moves to the next line after
        printing
    return 0;
}
```

Formatted Output: printf()

The `printf()` Function

- **Purpose:** The most versatile output function in C. Used to print strings, numbers, and characters in a formatted way.
- **Syntax:**

‘c
int printf(const char *format, ...);
‘

Format String Components

1. **Plain Text:** Printed exactly as typed.
2. **Format Specifiers:** Start with % (e.g., `%d`, `%f`) and are replaced by the values of variables.
3. **Escape Sequences:** Start with \ (e.g., `\n` for newline, `\t` for tab).

Common Format Specifiers

Specifiers for `printf` and `scanf`

Specifier	Data Type	Example Output
`%d` or `%i`	`int`	`42`
`%f`	`float`	`3.141593`
`%lf`	`double`	`3.14159265`
`%c`	`char`	`'A'`
`%s`	String (`char` array)	`"Hello"`
`%u`	`unsigned int`	`42`
`%x` / `%X`	Hexadecimal `int`	`2a` / `2A`

Formatting Output with printf()

Controlling Width and Precision

You can control exactly how data is displayed.

```c
#include <stdio.h>
int main() {
    int num = 45;
    float pi = 3.14159;
    // Field width of 5, right-aligned
    printf("Number: |%5d|\n", num);    // Output: |   45|
    // Field width of 5, left-aligned (minus sign)
    printf("Number: |%-5d|\n", num);   // Output: |45   |
    // Precision to 2 decimal places
    printf("Pi: %.2f\n", pi);          // Output: 3.14
    return 0;
}
```

Formatted Input: scanf()

The `scanf()` Function

- **Purpose:** Reads formatted data from the standard input (keyboard).
- **Syntax:**

‘c
int scanf(const char *format, ...);
‘

- **The Address-of Operator (`&`):**
- `scanf()` needs to know *where* in memory to store the input.
- We must use the `&` operator before standard variables (like `int`, `float`, `char`).
- *Exception:* Arrays (like strings) do not need the `&` because the array name acts as an address.

Example: Using scanf()

```c
#include <stdio.h>
int main() {
    int age;
    float gpa;
    char name[50];
    printf("Enter your first name: ");
    scanf("%s", name);  // No & needed for arrays/strings
    printf("Enter your age and GPA: ");
    // Reading multiple values at once
    scanf("%d %f", &age, &gpa); // & is REQUIRED here
    printf("\n--- Profile ---\n");
    printf("Name: %s\nAge: %d\nGPA: %.2f\n", name, age, gpa);
    return 0;
}
```

Comparing gets() and scanf()

Problem with `scanf("%s")`

If the user types `John Doe`:

- `scanf("%s", name);` will only read `John`. It stops at the space.

Solution using `gets()`

- `gets(name);` will read the entire line `John Doe` until Enter is pressed.

Modern C Alternative

Because `gets()` is unsafe, modern C programmers use `fgets()`:

```
fgets(name, sizeof(name), stdin);
```

(We will cover `fgets` in detail in the file I/O unit.)

Summary

Quick Recap

- **I/O Functions** bridge the gap between the user and the program.
- **Unformatted Functions** (`getch`, `putch`, `gets`, `puts`) handle raw characters and strings.
- `gets()` reads spaces but is unsafe; `puts()` automatically adds a newline.
- **Formatted Functions** (`scanf`, `printf`) offer precise control over how data is read or displayed using format specifiers (`%d`, `%f`, `%s`).
- **The `&` Operator** is mandatory in `scanf` for standard variables to provide the memory address.

Lecture 11: Decision Control and if, if-else

Unit 3: Decision Control & Looping

- Introduction to Decision Control
- The `if` Statement
- The `if-else` Statement

Agenda for Today

1. What is Decision Control?
2. Control Flow in C
3. The `if` Statement: Syntax and Logic
4. Examples of `if` Statements
5. The `if-else` Statement: Syntax and Logic
6. Examples of `if-else` Statements
7. Common Pitfalls and Best Practices

1. Introduction to Decision Control

Sequential Execution:
By default, instructions in a C program are executed sequentially, one after another, from top to bottom.
What if we want to change this?
Real-world problems require decision making. For example:

- *If* it is raining, take an umbrella.
- *If* a student's marks are ¿= 35, they pass; *otherwise*, they fail.

Decision control instructions allow the computer to take different paths depending on a logical condition.

Types of Decision Control in C

C provides several ways to make decisions:

1. `if` **statement** (Simple decision)
2. `if-else` **statement** (Two-way decision)
3. **Nested** `if-else` (Multi-way decision)
4. `else-if` **ladder** (Multi-way decision)
5. `switch` **statement** (Multi-way selection)
6. **Conditional operator (`?:`)** (Inline decision)

Today, we focus exclusively on `if` and `if-else`.

The `if` Statement

The `if` statement evaluates a test expression (condition).
Syntax:

```
if (test_expression) {
    // block of code to be executed if the condition is true
}
```

- If `test_expression` evaluates to non-zero (true), the block is executed.
- If it evaluates to zero (false), the block is skipped.

Example: Simple if Statement

```c
#include <stdio.h>
int main() {
    int age;
    printf("Enter your age: ");
    scanf("%d", &age);
    if (age >= 18) {
        printf("You are eligible to vote.\n");
    }
    printf("End of program.\n");
    return 0;
}
```

Important Rules for `if` Statements

1. **Parentheses are mandatory:** `if condition` is a syntax error. It must be `if (condition)`.
2. **No semicolon after the condition:**
   - WRONG: `if (x > 5);`
   - This creates an empty statement, meaning the block below it is no longer controlled by the `if`.
3. **Braces can be omitted for a single statement:**

```c
if (x > 5)
    printf("x is greater than 5");
```

Best Practice: Always use braces to avoid logic errors during future edits.

The `if-else` Statement

What if we want to execute a different block of code when the condition is false?

Syntax:

```
if (test_expression) {
    // Executed if condition is true
} else {
    // Executed if condition is false
}
```

- It provides a strict either/or choice.
- Exactly one of the two blocks will execute.

Example: `if-else` Statement

```c
#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);
    if (number % 2 == 0) {
        printf("%d is an EVEN number.\n", number);
    } else {
        printf("%d is an ODD number.\n", number);
    }
    return 0;
}
```

Common Pitfall: = vs ==

One of the most frequent errors in C:

```c
int x = 5;
// WRONG: Assignment operator
if (x = 10) {
    printf("This will ALWAYS print!");
}
// CORRECT: Relational operator
if (x == 10) {
    printf("This will only print if x is 10.");
}
```

Why? `x = 10` assigns 10 to `x`, and the expression evaluates to 10 (non-zero = true).

Example: Checking Leap Year (Logic)

Let's determine if a simple year is a leap year (divisible by 4, assuming standard simple rules for now):

```c
#include <stdio.h>
int main() {
    int year;
    printf("Enter year: ");
    scanf("%d", &year);
    if (year % 4 == 0) {
        printf("%d is a leap year.\n", year);
    } else {
        printf("%d is not a leap year.\n", year);
    }
    return 0;
}
```

Logical Operators in Conditions

We often need to check multiple conditions simultaneously. We use logical operators:

- && (Logical AND): True if BOTH operands are true.
- || (Logical OR): True if AT LEAST ONE operand is true.
- ! (Logical NOT): Reverses the truth value.

```
if (age >= 18 && hasID == 1) {
    printf("Entry permitted.\n");
}
```

Example: Largest of Two Numbers

```c
#include <stdio.h>
int main() {
    int a, b;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    if (a > b) {
        printf("Largest is %d\n", a);
    } else {
        // If they are equal, b is printed, which is fine.
        printf("Largest is %d\n", b);
    }
    return 0;
}
```

Summary

- **Decision Control** allows programs to choose execution paths.
- `if` **statement:** Executes a block of code if a condition is true.
- `if-else` **statement:** Provides an alternative block if the condition is false.
- **Condition Evaluation:** True is non-zero, False is zero.
- **Caution:** Always distinguish between = (assignment) and == (comparison).

if-else-if ladder and nested if-else

Unit 3: Decision Control & Looping
Handling multiple and complex conditions in C

Agenda for Today

The need for multiple conditions
The if-else-if ladder statement
Syntax, Flowchart, and Examples of if-else-if
The nested if-else statement
Syntax, Flowchart, and Examples of nested if-else
Comparing if-else-if ladder and nested if-else

Handling Multiple Conditions

Real-world problems often require choosing from more than two options. Examples:

- Assigning grades (A, B, C, D, Fail) based on marks.
- Categorizing a person's age (Child, Teenager, Adult, Senior).
- Finding the largest among three or more numbers.

Using just simple 'if' or single 'if-else' statements for these is inefficient and makes code hard to read.

The if-else-if Ladder

The if-else-if ladder is used to execute exactly one block of code among many alternatives.
It evaluates conditions sequentially from top to bottom.
As soon as a true condition is found, its corresponding block is executed.
All remaining conditions in the ladder are skipped.
If none of the conditions are true, the final 'else' block (if present) is executed.

Syntax: if-else-if Ladder

```
if (condition1) {
    // statements if condition1 is true
} else if (condition2) {
    // statements if condition2 is true
} else {
    // statements if all conditions are false
}
```

Flowchart of if-else-if Ladder

1. Start -¿ Condition 1?
2. If True -¿ Execute Block 1 -¿ Exit Ladder
3. If False -¿ Condition 2?
4. If True -¿ Execute Block 2 -¿ Exit Ladder
5. If False -¿ Check next condition...
6. If all False -¿ Execute Default Else Block -¿ Exit Ladder

Example: Grading System (if-else-if)

```c
#include <stdio.h>
int main() {
    int marks = 85;
    if (marks >= 90) {
        printf("Grade: A\n");
    } else if (marks >= 80) {
        printf("Grade: B\n");
    } else {
        printf("Grade: C\n");
    }
    return 0;
}
```

The nested if-else Statement

An 'if' or 'if-else' statement can be placed completely inside another 'if' or 'else' block.
This is called nesting.
Nesting is useful when a decision depends on a previous decision.
It allows checking sub-conditions only if a primary condition is met.

Syntax: nested if-else

```
if (condition1) {
    if (condition2) {
        // statements if both condition1 and condition2 are true
    } else {
        // statements if condition1 is true but condition2 is false
    }
} else {
    // statements if condition1 is false
}
```

Flowchart of nested if-else

1. Start -¿ Condition 1?
2. If True -¿ Condition 2? (Nested)
   - If True -¿ Execute Block A
   - If False -¿ Execute Block B
3. If False (Condition 1 is False) -¿ Execute Block C
4. All paths eventually converge and continue.

Example: Greatest of 3 Numbers (Nested if)

```c
#include <stdio.h>
int main() {
    int a = 10, b = 20, c = 15;
    if (a > b) {
        if (a > c) printf("a is greatest\n");
        else printf("c is greatest\n");
    } else {
        if (b > c) printf("b is greatest\n");
        else printf("c is greatest\n");
    }
    return 0;
}
```

Dangling Else Problem

In nested if statements without curly braces, an 'else' associates with the nearest preceding 'if'.
This can lead to logic errors if not careful.

Ladder vs. Nested: Which to use?

if-else-if Ladder:

- Best when checking a single variable against multiple distinct values or ranges.
- Readability remains high even with many conditions.

Nested if-else:

- Best when conditions are interdependent (one condition is a prerequisite for another).
- Deep nesting (more than 3 levels) reduces readability drastically.

Summary

The **if-else-if ladder** allows selecting one block among many, evaluating top-to-bottom.
The **nested if-else** involves placing conditionals inside other conditionals for dependent logic.
Always use curly braces $\{\}$ to prevent the dangling else problem and improve readability.
Choosing the right structure depends on the problem's logic (independent ranges vs. dependent checks).

Lecture 13: switch case statement
Topic: 3.2.5 switch case statement

Agenda for Today

Introduction to the switch statement
Syntax and Flow of Execution
The role of break and default keywords
Rules for switch and case labels
Practical Examples: Days, Calculator, Vowels
Fall-through Behavior
Difference between switch and else-if ladder

Introduction to switch case

The `switch` statement is a multi-way branch statement.
It provides an easy way to dispatch execution to different parts of code based on the value of a single variable or expression.
It is often used as a more readable alternative to a long `else-if` ladder when testing a single variable against multiple constant values.
It improves code clarity and often executes faster than multiple `if-else` checks.

Syntax of switch Statement

```
switch (expression) {
    case constant1:
        // statements
        break;
    case constant2:
        // statements
        break;
    ...
    default:
        // statements
}
```

The expression is evaluated once.
Its value is compared with the values of each case label.

How switch Executes

1. The `expression` inside the `switch()` is evaluated.
2. The control jumps to the `case` label whose constant value matches the expression's result.
3. All statements following the matching `case` are executed until a `break` statement is encountered.
4. If no `case` matches, the control jumps to the `default` label (if present).
5. Once `break` is executed, control exits the entire `switch` block.

The break Statement

The `break` statement is crucial in a `switch` block.
It terminates the execution of the `switch` statement and transfers control to the statement immediately following the `switch` block.
If `break` is omitted, execution continues into the next `case` sequentially, regardless of whether its constant matches.
This behavior is known as **fall-through**.

Example: Days of the Week

```c
#include <stdio.h>
int main() {
    int day = 3;
    switch (day) {
        case 1: printf("Monday\n"); break;
        case 2: printf("Tuesday\n"); break;
        case 3: printf("Wednesday\n"); break;
        case 4: printf("Thursday\n"); break;
        default: printf("Invalid day\n");
    }
    return 0;
}
```

Output: `Wednesday`

The default Case

The `default` case is optional but highly recommended.
It is executed only if none of the `case` constants match the switch expression.
It acts like the `else` block in an `if-else` ladder.
It can be placed anywhere inside the `switch`, but is conventionally placed at the very end.
If placed at the end, it does not strictly need a `break` statement, but adding one is a good habit.

Rules for switch and case Labels

1. The `switch` expression must evaluate to an integer or character type (`int`, `char`, `short`, `long`). Floating-point types (`float`, `double`) are **not allowed**.
2. The `case` labels must be **constants** or **constant expressions** (e.g., `case 5:`, `case 'A':`, `case 1+2:`). Variables are not allowed.
3. Two `case` labels cannot have the same value (no duplicates).
4. Relational or logical expressions (like `case x > 5:`) are not permitted.

Example: Simple Calculator

```c
#include <stdio.h>
int main() {
    char op = '*';
    int a = 10, b = 5;
    switch (op) {
        case '+': printf("Sum: %d\n", a + b); break;
        case '-': printf("Diff: %d\n", a - b); break;
        case '*': printf("Prod: %d\n", a * b); break;
        case '/': printf("Quot: %d\n", a / b); break;
        default:  printf("Invalid Operator\n");
    }
    return 0;
}
```

Output: `Prod: 50`

Fall-through Behavior

If a `case` matches and there is no `break` statement at the end of it, execution continues into the next `case` block automatically.
This happens even if the next `case` condition does not match.
Usually, this is a logical error caused by forgetting `break`.
However, sometimes programmers use fall-through **intentionally** to execute the same block of code for multiple case values.

Intentional Fall-through: Vowel or Consonant

```c
#include <stdio.h>
int main() {
    char ch = 'E';
    switch (ch) {
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
            printf("Vowel\n");
            break;
        default:
            printf("Consonant\n");
    }
    return 0;
}
```

Output: `Vowel`

switch vs. else-if Ladder

Condition Evaluation:

- `switch` tests only equality against constants.
- `else-if` can test ranges, logical operators, and variables.

Data Types:

- `switch` allows only `int` and `char`.
- `else-if` allows any type (float, string via functions, etc.).

Readability:

- `switch` is much cleaner for multiple exact matches.
- `else-if` gets cluttered for many exact matches.

Summary

The `switch` statement is a powerful multi-way branching tool in C.
It compares a single expression against a list of integer or character constants.
The `break` keyword prevents unintended fall-through.
The `default` case handles values that do not match any case label.
Cases cannot be duplicated, and must be constants.
Intentional fall-through can simplify code when multiple inputs share the same logic.

Next Lecture Preview

Topic: 3.3 Looping Statements
Introduction to the concept of iteration.
The `while` loop syntax and structure.
Understanding loop conditions and update expressions.
Writing programs to repeat tasks efficiently.

Introduction to Branching and while Loop

Unit 3: Decision Control & Looping
Topics: Branching & Looping concepts, while Loop syntax and examples
Course: Programming in C

Agenda

Recap of Decision Making
Need for Looping Statements
Concept of Looping (Iteration)
Categories of Loops in C
Introduction to the `while` loop
Syntax and Flowchart of `while` loop
Examples of `while` loop
Common Pitfalls (Infinite Loops)

Need for Looping Statements

Imagine you need to print 'Hello World' 5 times. You could write `printf` 5 times.
What if you need to print it 1000 times? Writing 1000 `printf` statements is impractical.

Issues with sequential repetition:

1. Increases code length unnecessarily.
2. Makes code difficult to read and maintain.
3. High probability of errors during typing.

Solution: Use looping structures to execute a block of code repeatedly.

Introduction to Branching and Looping

Control structures in C dictate the flow of execution.

Branching (Decision Making):

- `if`, `if-else`, `switch`
- Executes a block of code *only once* based on a condition.

Looping (Iteration):

- `while`, `do-while`, `for`
- Executes a block of code *repeatedly* as long as a condition is true.
- Once the condition becomes false, the loop terminates.

Categories of Loops in C

Loops are broadly classified into two categories based on when the condition is checked:

1. **Entry-Controlled Loops:**
   - The test condition is checked *before* entering the loop body.
   - If the condition is false initially, the loop body is never executed.
   - Examples: `while` loop, `for` loop.
2. **Exit-Controlled Loops:**
   - The test condition is checked *after* executing the loop body.
   - The loop body is executed *at least once*, even if the condition is false initially.
   - Example: `do-while` loop.

The `while` Loop

The `while` loop is the most fundamental looping statement in C.
It is an **entry-controlled loop**.
Syntax:

```
while (test_condition) {
    // Body of the loop
    // Statements to be executed
}
```

- The `test_condition` is any valid C expression.
- The loop executes as long as `test_condition` evaluates to non-zero (true).
- If the body has only one statement, the braces $\{\}$ are optional, but recommended.

How the `while` Loop Works

Three essential components for any loop:

1. **Initialization:** Set the starting value of the loop control variable (before the loop).
2. **Condition:** The test expression evaluated before each iteration (inside `while(...)`).
3. **Update (Increment/Decrement):** Modify the loop control variable inside the loop body to eventually make the condition false.

Execution Flow:

- Step 1: Evaluate the condition.
- Step 2: If true, execute loop body and update variable. Go to Step 1.
- Step 3: If false, exit the loop and continue with the rest of the program.

Example 1: Printing 1 to 5

```c
#include <stdio.h>

int main() {
    int i = 1; // 1. Initialization

    while (i <= 5) { // 2. Condition
        printf("%d ", i);
        i++; // 3. Update
    }

    return 0;
}
```

Output:

1 2 3 4 5

Tracing the Execution (Dry Run)

Let's trace the previous example step-by-step:

Iteration	Initial `i`	Condition `i <= 5`	Output	Updated `i`
1st	1	1 ¡= 5 (True)	1	2
2nd	2	2 ¡= 5 (True)	2	3
3rd	3	3 ¡= 5 (True)	3	4
4th	4	4 ¡= 5 (True)	4	5
5th	5	5 ¡= 5 (True)	5	6
6th	6	6 ¡= 5 (False)	Loop terminates	-

Example 2: Sum of first N natural numbers

```c
#include <stdio.h>

int main() {
    int n = 5, i = 1, sum = 0;

    while (i <= n) {
        sum = sum + i;
        i++;
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

Output:

```
Sum = 15
```

Example 3: Decrementing loop (Reverse counting)

```c
#include <stdio.h>

int main() {
    int count = 5;

    while (count > 0) {
        printf("%d ", count);
        count--; // Decrementing the counter
    }

    printf("\nBlast off!\n");
    return 0;
}
```

Output:

```
5 4 3 2 1
Blast off!
```

Infinite Loops

An infinite loop never ends because its condition never evaluates to false.
Usually caused by a logical error: missing or incorrect update statement.
Example of infinite loop:

```c
int i = 1;
while (i <= 5) {
    printf("%d ", i);
    // Missing i++;
}
```

Since `i` remains 1, `i <= 5` is always true.
To stop an infinite loop: Press `Ctrl + C` in the terminal.

Common Pitfalls with `while` Loops

1. **Semicolon after while condition:**

```c
while (i <= 5); // Empty loop body executes infinitely
{
    printf("%d", i);
    i++;
}
```

2. **Uninitialized variables:** Using a loop variable without giving it an initial value results in garbage behavior.
3. **Off-by-one errors:** Using < instead of <= or vice versa, causing the loop to run one time too many or one time too few.

Summary

Loops are used to execute a block of code repeatedly.
The `while` loop is an **entry-controlled loop**.
Its condition is evaluated before every iteration.
Every loop requires three elements: **initialization, condition, and update**.
Failing to update the loop variable properly results in an **infinite loop**.
Tracing loops manually (dry run) is the best way to understand and debug them.

Next Lecture Preview

We will learn about the `do-while` **loop**.
How is it different from the `while` loop?
What is an **exit-controlled loop**?
When should we use `do-while` instead of `while`?
Practical examples comparing both loops.

Lecture 15: Do-While Loop

Understanding the exit-controlled loop in C programming.

Agenda

- Recap: The While Loop
- Introduction to Do-While Loop
- Syntax and Flowchart
- Entry-Controlled vs Exit-Controlled Loops
- Practical Examples
- Common Mistakes

Recap: The While Loop

- A **while loop** is an entry-controlled loop.
- The condition is checked **before** entering the loop body.
- If the condition is false initially, the loop body **never executes**.

```
int i = 5;
while (i < 5) {
    printf("This will not print.\n");
}
```

Introduction to Do-While Loop

- The **do-while loop** is an **exit-controlled** loop.
- The condition is evaluated **after** the loop body executes.
- **Guarantee**: The loop body will execute **at least once**, regardless of whether the condition is true or false initially.
- Highly useful for menu-driven programs or validating user input.

Syntax of Do-While Loop

```
do {
    // Statements to execute
    // Update expression
} while (condition);
```

- The `do` keyword marks the start.
- The body is enclosed in { }.
- The `while(condition)` is placed at the end.
- **Important**: Notice the **semicolon (;)** at the end of the `while` statement!

Flowchart of Do-While Loop

1. Enter loop.
2. Execute loop body.
3. Evaluate test condition.
4. If True, go back to step 2.
5. If False, exit loop and continue with the next statement in the program.

Example 1: Basic Counting

```c
#include <stdio.h>
int main() {
    int i = 1;
    do {
        printf("%d ", i);
        i++;
    } while (i <= 5);
    return 0;
}
```

Output:

1 2 3 4 5

The 'At Least Once' Guarantee

What happens if the condition is false from the very beginning?

```c
#include <stdio.h>
int main() {
    int i = 10;
    do {
        printf("i is %d\n", i);
        i++;
    } while (i < 5);
    return 0;
}
```

Output:

```
i is 10
```

Entry-Controlled vs Exit-Controlled

— Feature — `while` Loop (Entry-Controlled) — `do-while` Loop (Exit-Controlled) —

— **Condition Check** — Beginning of loop — End of loop —
— **Minimum Executions** — 0 — 1 —
— **Semicolon** — No semicolon after `while(cond)` — Semicolon required after `while(cond);` —
— **Use Case** — When you might not need to run the loop at all — When you must run the code at least once (e.g., menus) —

Example 2: Input Validation

```c
#include <stdio.h>
int main() {
    int number;
    do {
        printf("Enter a positive number: ");
        scanf("%d", &number);
    } while (number <= 0);
    printf("Thank you! You entered %d\n", number);
    return 0;
}
```

Execution Trace of Example 2

Assume the user enters -5, then 0, then 7.
1. **Iteration 1**: Prompts user. User enters -5. Condition `-5 <= 0` is **True**. Loop repeats.
2. **Iteration 2**: Prompts user. User enters 0. Condition `0 <= 0` is **True**. Loop repeats.
3. **Iteration 3**: Prompts user. User enters 7. Condition `7 <= 0` is **False**. Loop exits.
4. Proceeds to print: `Thank you!  You entered 7.`

Common Mistakes: Missing Semicolon

Forgetting the semicolon at the end of the `while` statement is a very common syntax error in C.

Incorrect:

```
do {
    // code
} while (x > 0)   // Error: Expected ';'
```

Correct:

```
do {
    // code
} while (x > 0);
```

Summary

- The **do-while loop** executes its body **before** evaluating the condition.
- It is an **exit-controlled** loop.
- It guarantees at least **one** execution of the loop body.
- Perfect for scenarios requiring at least one action, such as user menus or input validation.
- Do not forget the trailing `;` after `while(condition);`.

Next Lecture Preview

- The `for` Loop
- Structure of `for` loop (initialization, condition, update)
- Comparing `while`, `do-while`, and `for` loops

Understanding the most compact and widely used loop in C programming.

Agenda for Today

- Introduction to the `for` Loop
- Syntax and Execution Flow
- Detailed Breakdown: Initialization, Condition, Update
- Variations and Omitted Expressions
- Practical Examples: Factorial and Patterns
- Nested `for` Loops
- Summary

Introduction to the For Loop

The `for` loop provides a concise way to write loops. It is primarily used when the number of iterations is known before entering the loop.

Syntax:

```
for (initialization; condition; update) {
    // body of the loop (statements)
}
```

- It gathers the three essential loop components (initialization, condition, and update) into a single line, making the code easier to read and maintain.

Execution Flow of a For Loop

Understanding the precise order of execution is crucial for mastering the `for` loop:

1. **Initialization**: Executed exactly *once* at the very beginning.
2. **Condition Evaluation**: Checked before every iteration.
   - If `true` (non-zero), the loop body executes.
   - If `false` (zero), the loop terminates.
3. **Loop Body**: The statements inside the braces are executed.
4. **Update**: Executed immediately after the loop body finishes.
5. Go back to Step 2 (Condition Evaluation).

Basic For Loop Example

Let's look at a simple program that prints numbers from 1 to 5.

```c
#include <stdio.h>
int main() {
    int i;
    for (i = 1; i <= 5; i++) {
        printf("%d ", i);
    }
    printf("\nLoop finished.\n");
    return 0;
}
```

Output:

```
1 2 3 4 5
Loop finished.
```

Dissecting the For Loop: Initialization

1. Initialization

- Used to set the starting value of loop control variables.
- Executed only once when the loop starts.
- You can initialize multiple variables separated by commas.

```
// Initializing a single variable
for (i = 0; ...; ...)
// Initializing multiple variables
for (i = 0, j = 10; ...; ...)
```

Note: In modern C (C99 onwards), you can declare the variable here: `for (int i = 0; ...)` but we stick to C89 style declaration at the top of the block for this course.

Dissecting the For Loop: Condition

2. Condition

- A relational or logical expression that evaluates to true (non-zero) or false (zero).
- Evaluated *before* every iteration.
- If omitted, it is assumed to be true (creating an infinite loop).

```
// Simple condition
for (...; i < 10; ...)
// Compound condition
for (...; i < 10 && j > 0; ...)
```

Dissecting the For Loop: Update

3. Update (Increment/Decrement)

- Modifies the loop control variable(s).
- Executed *after* the loop body, before the next condition check.
- Can be an increment, decrement, or any arithmetic operation.

```
// Increment by 1
for (...; ...; i++)
// Decrement by 1
for (...; ...; i--)
// Complex update
for (...; ...; i = i + 2, j--)
```

Example: Multiple Variables in For Loop

We can use the comma operator to manage two variables in a single `for` loop.

```
#include <stdio.h>
int main() {
    int i, j;
    for (i = 0, j = 5; i < j; i++, j--) {
        printf("i = %d, j = %d\n", i, j);
    }
    return 0;
}
```

Output:

```
i = 0, j = 5
i = 1, j = 4
i = 2, j = 3
```

Variations: Omitted Expressions

Any or all of the three expressions in a `for` loop header can be omitted. The semicolons, however, *must* remain.

Missing Initialization and Update:

```
int i = 0;
for (; i < 5;) {
    printf("%d ", i);
    i++;
}
```

(This behaves exactly like a `while` loop!)

The Infinite Loop:

```
for (;;) {
    printf("This will print forever!\n");
}
```

(Omitted condition is treated as true).

Practical Example: Calculating Factorial

The `for` loop is perfect for calculations requiring a specific number of steps, like finding the factorial of a number.

```c
#include <stdio.h>
int main() {
    int n, i;
    long long factorial = 1;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        factorial = factorial * i;
    }
    printf("Factorial of %d = %lld\n", n, factorial);
    return 0;
}
```

Nested For Loops

A `for` loop can be placed inside the body of another `for` loop. This is known as a nested loop.

- The outer loop controls the number of times the inner loop completely executes.
- For every single iteration of the outer loop, the inner loop runs through all of its iterations.

```
for (i = 1; i <= 3; i++) {          // Outer loop (runs 3 times)
    for (j = 1; j <= 2; j++) {      // Inner loop (runs 2 times per
        outer step)
        // Executes a total of 3 * 2 = 6 times
    }
}
```

Example: Nested Loops for Star Patterns

Printing a right-angled triangle pattern.

```c
#include <stdio.h>
int main() {
    int rows = 5;
    int i, j;
    for (i = 1; i <= rows; i++) {          // Controls rows
        for (j = 1; j <= i; j++) {         // Controls columns per row
            printf("* ");
        }
        printf("\n");                      // Move to next line
    }
    return 0;
}
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Summary

- The `for` loop provides a compact syntax: `for (initialization; condition; update)`.
- Execution flow: Initialize once, then loop (Check condition, execute body, update).
- It is best used when the number of iterations is known in advance.
- Expressions in the loop header can be omitted, leading to infinite loops like `for(;;)`.
- The comma operator allows managing multiple variables simultaneously.
- `for` loops can be nested to handle multidimensional tasks (like rows and columns).

Next Lecture Preview

In our next lecture, we will cover **Jumping Statements**.
We will explore how to alter the normal flow of loops unconditionally using:

- `break`: To exit a loop immediately.
- `continue`: To skip the rest of the current iteration and move to the next.
- `goto`: To jump to a specific label (and why to avoid it!).

Unit 3: Decision Control & Looping

Lecture 17: Nested for loop
Subject: C Programming (DI03011011)
Semester: 3

Agenda

1. What is a Nested Loop?
2. Syntax of a Nested for Loop
3. Understanding the Execution Flow
4. Real-world Analogy
5. Example 1: Printing a Matrix/Grid
6. Example 2: Star Pattern Generation
7. Example 3: Multiplication Tables
8. Common Mistakes and Best Practices
9. Summary

What is a Nested Loop?

A nested loop is simply a loop placed inside the body of another loop.
The loop that encloses the other is called the **Outer Loop**.
The loop that is enclosed is called the **Inner Loop**.
Key Rule: For every single iteration of the outer loop, the inner loop executes completely from start to finish.
Nested loops are essential for working with multi-dimensional structures like 2D arrays, grids, matrices, and complex patterns.

Syntax of a Nested for Loop

```
for (initialization1; condition1; update1) {
    // Outer loop body
    for (initialization2; condition2; update2) {
        // Inner loop body
    }
    // More outer loop body (optional)
}
```

The variables used for initialization1 and initialization2 must be distinct (typically `i` for outer and `j` for inner) to avoid interference.

Understanding the Execution Flow

Step 1: The outer loop initializes its counter.
Step 2: The outer loop evaluates its condition. If true, it enters its body.
Step 3: The inner loop initializes its counter.
Step 4: The inner loop evaluates its condition. If true, it executes its body, updates, and repeats until its condition is false.
Step 5: Once the inner loop terminates, the outer loop continues its body, updates its counter, and goes back to Step 2.

Real-world Analogy: A Clock

Imagine a digital clock displaying Hours and Minutes.
Outer Loop (Hours): Goes from 0 to 23.
Inner Loop (Minutes): Goes from 0 to 59.
For every *single* hour that passes, the minutes must go through their *entire* cycle (0 to 59) before the hour increments by 1.
Hour 1 -¿ Mins: 0,1,2...59
Hour 2 -¿ Mins: 0,1,2...59

Example 1: Printing Coordinates (Grid)

```c
#include <stdio.h>

int main() {
    int r, c;
    // Outer loop for rows (1 to 3)
    for (r = 1; r <= 3; r++) {
        // Inner loop for columns (1 to 2)
        for (c = 1; c <= 2; c++) {
            printf("(%d, %d) ", r, c);
        }
        printf("\n"); // Newline after each row
    }
    return 0;
}
```

Example 1: Output and Tracing

Output:
(1, 1) (1, 2)
(2, 1) (2, 2)
(3, 1) (3, 2)
Trace:
When `r`=1, inner loop runs for c=1, then c=2.
Line breaks `\n`.
When `r`=2, inner loop resets and runs for c=1, then c=2.
Line breaks `\n`.

Example 2: Generating a Right-Angled Triangle Pattern

```c
#include <stdio.h>

int main() {
    int i, j;
    for (i = 1; i <= 4; i++) {
        // Inner loop limit depends on outer loop counter 'i'
        for (j = 1; j <= i; j++) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

Example 2: Tracing the Pattern

Iteration 1: `i=1 -¿ j` loops from 1 to 1 (Prints 1 star). Newline.
Iteration 2: `i=2 -¿ j` loops from 1 to 2 (Prints 2 stars). Newline.
Iteration 3: `i=3 -¿ j` loops from 1 to 3 (Prints 3 stars). Newline.
Iteration 4: `i=4 -¿ j` loops from 1 to 4 (Prints 4 stars). Newline.
Output:

```
*
 *

  *
```

Example 3: Multiplication Tables (1 to 5)

```c
#include <stdio.h>

int main() {
    int table, multiplier;
    for (table = 1; table <= 5; table++) {
        printf("Table of %d:\n", table);
        for (multiplier = 1; multiplier <= 10; multiplier++) {
            printf("%d x %d = %d\n", table, multiplier, (table *
                multiplier));
        }
        printf("---------\n");
    }
    return 0;
}
```

Common Mistakes in Nested Loops

1. Using the same counter variable:

```
for(int i=0; i<5; i++) { for(int i=0; i<3; i++) { ... } }
```

(This ruins the outer loop's count.)

2. Misplaced Braces: Forgetting braces $\{\}$ can cause the inner loop to act strangely if you intend to include multiple statements.

3. Infinite Loops: Modifying the outer loop's counter inside the inner loop by accident.

Summary

A nested for loop is a loop inside another loop.
The inner loop completes all its iterations for every single iteration of the outer loop.
Commonly used for 2D grids, matrices, tabular data, and printing structural patterns.
The total number of executions for the innermost statement is roughly (Outer Iterations * Inner Iterations).
Careful variable tracking (tracing) is the best way to understand and debug nested loops.

Next Lecture Preview

We have mastered how to repeat code using while, do-while, and for loops (including nested ones).
But what if we need to stop a loop prematurely or skip an iteration based on a condition?
Next Lecture: Jump Statements (`break`, `continue`, and `goto`).

Jump Statements in C: break, continue, and goto

Understanding how to alter the normal flow of loops using jump statements.

Agenda

Introduction to Jump Statements
The `break` Statement
Examples of `break`
The `continue` Statement
Examples of `continue`
Comparison: `break` vs `continue`
The `goto` Statement and Labels
Best Practices and Summary

What are Jump Statements?

By default, loops execute until their test condition becomes false.
Sometimes, we need to alter this normal flow based on a specific event or condition inside the loop.
C provides **jump statements** to perform an unconditional transfer of control:

- `break`: Exits the loop entirely.
- `continue`: Skips the rest of the current iteration and goes to the next.
- `goto`: Jumps to a specific labeled line of code.

The `break` Statement

The `break` statement immediately terminates the loop or `switch` statement in which it is placed.
When `break` is encountered, the control passes to the statement immediately following the loop.
It is almost always used with an `if` statement to exit the loop early when a specific condition occurs.
In nested loops, `break` only exits the innermost loop containing it.

Example: `break` in a Loop

Searching for a specific number and exiting early:

The `continue` Statement

The `continue` statement skips the remaining code inside the loop for the **current iteration only**.
When `continue` is encountered, control immediately jumps to the loop's update/evaluation step:

- In a `for` loop, it jumps to the update expression (e.g., `i++`), then checks the condition.
- In a `while` or `do-while` loop, it jumps directly to the test condition.

It does **not** terminate the loop completely, unlike `break`.

Example: `continue` in a Loop

Printing only odd numbers by skipping even numbers:

Comparing `break` and `continue`

`break`:

- Terminates the loop entirely.
- Control goes to the statement following the loop.
- Used to exit early when a goal is achieved or an error occurs.

`continue`:

- Terminates only the current iteration.
- Control goes to the loop's next iteration (update/test).
- Used to skip processing for specific unwanted cases.

The goto Statement

The goto statement performs an absolute, unconditional jump to another part of the program.
It requires a **label** to know where to jump.
A label is an identifier followed by a colon (:).
Syntax:

```
goto label_name;
...
label_name:
 // code to execute
```

Example: Using goto

Jumping over code using a label:

The Problem with `goto` (Spaghetti Code)

While `goto` is valid C, its use is heavily discouraged by modern programming standards.
Why?

- It disrupts the structured flow of the program.
- Code with many `goto` statements becomes difficult to trace, read, and maintain.
- This unreadable, tangled code is often called **'Spaghetti Code'**.

Any logic written with `goto` can be rewritten using `if`, `while`, `for`, `break`, and `continue`.

When is `goto` acceptable? (Rare Cases)

There is one scenario where C programmers sometimes still use `goto`:
Breaking out of deeply nested loops.
Since `break` only exits the innermost loop, exiting 3 or 4 nested loops requires complex boolean flags.
A single `goto` can jump completely out of the nested structure cleanly.
Even then, it is often better to extract the nested loops into a separate function and use `return`.

Summary

`break` exits the current loop or `switch` entirely.
`continue` skips the rest of the current iteration and forces the next iteration of the loop.
`goto` jumps to a specific labeled line of code within the function.
`break` and `continue` are essential tools for fine-tuning loop execution.
`goto` should be avoided to prevent unreadable 'spaghetti code'.

Preview of Next Lecture

Introduction to Arrays
Need for Arrays
Declaring and Initializing 1D Arrays
Accessing Array Elements

Unit 4: Array and Pointer

Lecture 19: Introduction to an Array & 1D Array

Agenda for Today

- The Need for Arrays
- What is an Array?
- Array Terminology and Memory Layout
- Advantages and Limitations of Arrays
- Declaring 1D Arrays
- Initializing 1D Arrays
- Accessing and Modifying Array Elements
- C Code Examples: Reading, Printing, and Processing Arrays

The Need for Arrays: A Motivating Example

Scenario: Suppose we need to store the marks of 5 students.
Without arrays, we must declare 5 separate variables:

```
int marks1 = 45;
int marks2 = 50;
int marks3 = 38;
int marks4 = 42;
int marks5 = 48;
```

The Problem:

- What if we have 60 students? Or 1000 students?
- Managing hundreds of unique variable names is impractical.
- We cannot easily write a loop to process all marks (e.g., finding the average) because each variable has a different name.

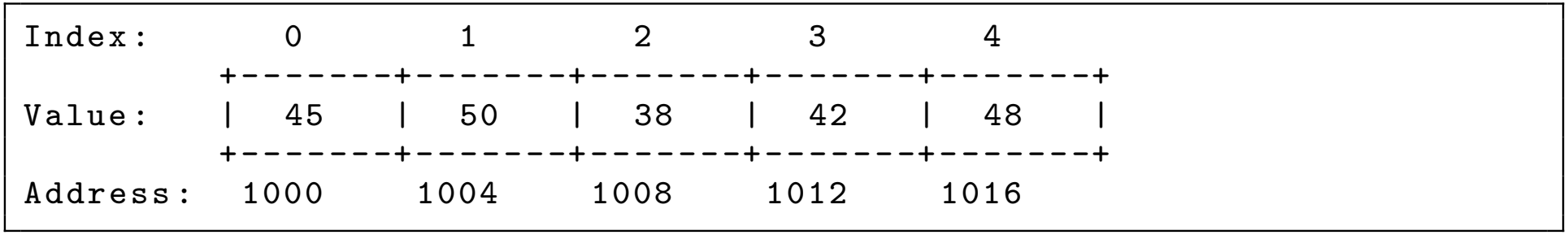
What is an Array?

Definition:
An array is a collection of elements of the **same data type**, stored in **contiguous (adjacent) memory locations**, and sharing a **single common name**.
Key Characteristics:

- **Homogeneous**: All elements must be of the same type (e.g., all `int` or all `float`).
- **Contiguous Allocation**: Memory is allocated in a single unbroken block.
- **Fixed Size**: Once an array is declared, its size cannot be changed (static memory allocation).
- **Indexed Access**: Elements are accessed using their position or 'index'.

Array Terminology & Memory Representation

Consider an integer array `marks` of size 5:



(Assuming a 32-bit architecture where `int` takes 4 bytes)

- **Element**: Individual data items in the array (e.g., 45, 50).
- **Index (Subscript)**: The position of an element. In C, **indexing always starts from 0**.
- **Base Address**: The memory address of the first element (index 0). Here, 1000.

Advantages and Limitations of Arrays

Advantages:
1. **Code Optimization**: Less code is needed to handle large amounts of data.
2. **Random Access**: Any element can be accessed directly using its index (e.g., `marks[3]`) in O(1) time.
3. **Easy Traversal**: Easy to traverse using loops.

Limitations:
1. **Fixed Size**: Array size is fixed at compile-time. We cannot shrink or expand it during runtime (leads to memory wastage or shortage).
2. **Homogeneous**: Cannot store different data types in the same array (e.g., cannot mix `int` and `float`).
3. **Insertion/Deletion overhead**: Inserting or removing elements in the middle requires shifting elements.

One-Dimensional (1D) Arrays: Declaration

To use an array, it must be declared so the compiler can allocate memory.

Syntax:

`data_type array_name[array_size];`

- `data_type`: Type of elements (e.g., `int`, `float`, `char`).
- `array_name`: Valid C identifier.
- `array_size`: An integer constant indicating the number of elements.

Examples:

```
int marks[5];          // Array of 5 integers
float salary[10];      // Array of 10 floats
char name[20];         // Array of 20 characters
```

1D Arrays: Compile-Time Initialization (Part 1)

Arrays can be initialized at the time of declaration using an initializer list (comma-separated values enclosed in curly braces {}).

Syntax:

`data_type array_name[size] = {val1, val2, ..., valN};`

Example: Full Initialization

```
int numbers[5] = {10, 20, 30, 40, 50};
```

Here, the size is 5, and exactly 5 values are provided.

- `numbers[0]` becomes 10
- `numbers[4]` becomes 50

1D Arrays: Compile-Time Initialization (Part 2)

Partial Initialization:
If fewer values are provided than the array size, the remaining elements are automatically initialized to zero.

```
int values[5] = {10, 20};
```

Result: `values` contains {10, 20, 0, 0, 0}.
Note: To initialize all elements to 0, use `int arr[10] = {0};`
Initialization without Size:
If an initializer list is provided, the array size can be omitted. The compiler determines the size automatically based on the number of elements.

```
int ages[] = {18, 19, 20, 21};
```

Result: Compiler allocates an array of size 4.

Accessing and Modifying Array Elements

Individual elements are accessed using the array name and the index inside square brackets.

Accessing values:

```c
int marks[3] = {80, 90, 85};
printf("First student: %d\n", marks[0]); // Prints 80
```

Modifying values:

```c
marks[1] = 95;  // Updates the second element from 90 to 95
```

Warning: Array Bounds

C **does not** perform bound checking. Accessing `marks[5]` in an array of size 3 will access out-of-bounds memory, leading to unpredictable behavior (garbage values or program crash/segmentation fault).

Reading and Printing a 1D Array (Using Loops)

Since array indices form a continuous arithmetic sequence (0, 1, 2, ...), we can use loops (like `for`) to process arrays efficiently.

Scanning (Reading) an array from User:

```c
int a[5];
int i;
printf("Enter 5 numbers:\n");
for(i = 0; i < 5; i++) {
    scanf("%d", &a[i]); // Note the & symbol
}
```

Printing an array:

```c
printf("Array elements are:\n");
for(i = 0; i < 5; i++) {
    printf("%d ", a[i]); // No & symbol
}
```

Example 1: Sum and Average of Array Elements

```c
#include <stdio.h>
int main() {
    int marks[5], i, sum = 0;
    float average;
    printf("Enter marks of 5 subjects:\n");
    for(i = 0; i < 5; i++) {
        scanf("%d", &marks[i]);
        sum = sum + marks[i];  // Calculate sum on the fly
    }
    average = (float)sum / 5;  // Typecast to float for precision
    printf("Total Marks = %d\n", sum);
    printf("Average = %.2f\n", average);
    return 0;
}
```

Example 2: Finding Maximum Element in an Array

```c
#include <stdio.h>
int main() {
    int arr[5] = {23, 14, 56, 12, 38};
    int i, max;
    max = arr[0]; // Assume first element is max
    for(i = 1; i < 5; i++) {
        if(arr[i] > max) {
            max = arr[i]; // Update max if a larger element is found
        }
    }
    printf("The maximum element is: %d\n", max);
    return 0;
}
```

Lecture Summary

- **Arrays** are collections of variables of the same data type stored in contiguous memory.
- They solve the problem of handling large volumes of similar data.
- **Indexing** starts at `0` and ends at `size - 1`.
- **Initialization** can happen at compile-time (using {}) or run-time (using `scanf` in loops).
- Partial initialization sets unassigned elements to zero.
- **Loops** (especially `for` loops) are used extensively to traverse, read, and write arrays.
- **C does no bounds checking**; programmers must ensure they do not access indices outside `0` to `size - 1`.

1D Array Memory and Display

Welcome to Lecture 20! Today we will explore how elements of a one-dimensional array are organized in memory and how we can traverse and display them effectively in C.

Agenda

What we will cover today:

1. Recap of 1D Arrays
2. Storing Array Elements in Memory
3. Memory Addresses and Contiguous Allocation
4. Calculating Memory Address of Elements
5. Traversing a 1D Array
6. Displaying Array Elements
7. Common Pitfalls during Array Display

Recap: What is a 1D Array?

Before discussing memory, let's recall:

- An array is a collection of elements of the **same data type**.
- Elements are accessed using an **index** (starting from 0).
- Syntax: `data_type array_name[size];`

Storing Array Elements in Memory

How does C store an array?

- **Contiguous Allocation**: All elements of an array are stored in consecutive memory locations.
- If the first element is at address `X`, the next element is at `X + sizeof(data_type)`.
- This contiguous nature allows fast, random access using indices.

Visualizing Memory Layout

Let's visualize `int marks[5] = {10, 20, 30, 40, 50};`
Assume base address (address of `marks[0]`) is 2000.
Also, assume an `int` takes 4 bytes of memory.

— Index — Element — Memory Address —

— 0 — marks[0] (10) — 2000 - 2003 —
— 1 — marks[1] (20) — 2004 - 2007 —
— 2 — marks[2] (30) — 2008 - 2011 —
— 3 — marks[3] (40) — 2012 - 2015 —
— 4 — marks[4] (50) — 2016 - 2019 —

Calculating Element Addresses

The compiler calculates the address of any element `array[i]` automatically using a formula:

Address of array[i] = Base_Address + (i * size_of_data_type)

Example for `marks[3]`:

Address = 2000 + (3 * 4)

Address = 2000 + 12 = 2012

Printing Memory Addresses in C

We can verify contiguous memory allocation in C by printing the addresses using the %p format specifier and the & (address-of) operator.

Traversing a 1D Array

What is Traversal?

- **Traversal** means visiting each element of the array exactly once.
- We usually traverse an array to read its values, print them, or perform calculations (like finding a sum or maximum).
- A `for` loop is the most natural construct for array traversal since we know the exact number of elements.

Displaying Array Elements

To display all elements of a 1D array:

1. Determine the size of the array (N).
2. Write a loop starting from index 0 up to N-1.
3. Inside the loop, use `printf` to print `array[index]`.

Example: Displaying Elements

A complete program to initialize and display a 1D array of floats.

Taking Input and Displaying

Often, we don't hardcode values. Let's see how to take input and then display it.

Common Pitfalls: Out of Bounds

What happens if we access an invalid index?

- **Array out-of-bounds** occurs when you try to access an index `< 0` or `>= size`.
- C does **not** perform bounds checking!
- Accessing `arr[5]` in an array of size 5 will read or corrupt random memory, leading to unpredictable behavior or a Segmentation Fault.

Summary

To wrap up today's lecture:

- 1D Array elements are stored in **contiguous** memory locations.
- The address of an element is calculated as: `Base Address + (Index * Size)`.
- We use loops (mostly `for` loops) to traverse and **display** array elements.
- Be extremely careful about array bounds (indices `0` to `N-1`)!

Next Lecture Preview

In our next lecture, we will explore:

- Operations on 1D Arrays.
- Finding minimum, maximum, and average of array elements.
- Introduction to Two-Dimensional (2D) Arrays.

Programming Exercises on 1D Arrays

Mastering One-Dimensional Arrays through Practical Programming

Agenda

- Review of 1D Array Basics
- Exercise 1: Finding Sum and Average of Elements
- Exercise 2: Finding Maximum and Minimum
- Exercise 3: Searching an Element (Linear Search)
- Exercise 4: Reversing an Array
- Exercise 5: Counting Even and Odd Elements
- Summary and Practice Problems

Quick Review: 1D Arrays

A one-dimensional array is a collection of elements of the same data type stored in contiguous memory locations.

```c
#include <stdio.h>
int main() {
    // Declaration and Initialization
    int marks[5] = {85, 90, 78, 92, 88};
    // Accessing elements using an index (0 to size-1)
    printf("First element: %d\n", marks[0]);
    return 0;
}
```

- Indexing always starts at 0.
- Size of array `marks` is 5, valid indices are 0 to 4.

Exercise 1: Sum and Average

Problem Statement:
Write a C program to read `N` elements into an integer array, then calculate the sum and average of those elements.

Logic:

1. Declare an array of sufficient size.
2. Read the number of elements `N`.
3. Use a loop to input `N` elements.
4. Iterate through the array, adding each element to a `sum` variable.
5. Calculate average as `(float)sum / N`.

Code: Sum and Average

```c
#include <stdio.h>
int main() {
    int arr[100], n, i, sum = 0;
    float average;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d integers:\n", n);
    for(i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        sum += arr[i]; // Accumulate sum during input
    }
    average = (float)sum / n;
    printf("Sum = %d\n", sum);
    printf("Average = %.2f\n", average);
    return 0;
}
```

Exercise 2: Maximum and Minimum

Problem Statement:
Write a C program to find the largest and smallest elements in a given 1D array.
Logic:
1. Assume the first element is both the maximum (`max`) and minimum (`min`).
2. Traverse the array from the second element (index 1).
3. If current element ¿ `max`, update `max`.
4. If current element ¡ `min`, update `min`.

Code: Maximum and Minimum

```c
#include <stdio.h>
int main() {
    int arr[5] = {45, 12, 89, 34, 67};
    int i, max, min;
    max = arr[0];
    min = arr[0];
    for(i = 1; i < 5; i++) {
        if(arr[i] > max) {
            max = arr[i];
        }
        if(arr[i] < min) {
            min = arr[i];
        }
    }
    printf("Maximum element = %d\n", max);
    printf("Minimum element = %d\n", min);
    return 0;
}
```

Exercise 3: Linear Search

Problem Statement:
Write a C program to search for a specific element (key) in an array and print its position if found.
Logic:
1. Read the array elements and the `key` to search.
2. Iterate through the array.
3. Compare each element with the `key`.
4. If a match is found, record the index, set a flag, and break out of the loop.
5. Check the flag after the loop to determine if the element was found.

Code: Linear Search

```c
#include <stdio.h>
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int i, key, found = 0, n = 5;
    printf("Enter element to search: ");
    scanf("%d", &key);
    for(i = 0; i < n; i++) {
        if(arr[i] == key) {
            printf("Element %d found at index %d.\n", key, i);
            found = 1;
            break;
        }
    }
    if(found == 0) {
        printf("Element %d not found in the array.\n", key);
    }
    return 0;
}
```

Exercise 4: Reversing an Array

Problem Statement:
Write a program to reverse the elements of an array *in-place* (without using a second array).
Logic:
1. Use two pointers or indices: `start` at 0 and `end` at N-1.
2. Swap the elements at `start` and `end`.
3. Increment `start` and decrement `end`.
4. Repeat until `start` ¿= `end`.

Code: Reversing an Array

```c
#include <stdio.h>
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int i, temp, start = 0, end = 4;
    while (start < end) {
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
    printf("Reversed array: ");
    for(i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Exercise 5: Counting Even and Odd

Problem Statement:
Write a program to count how many even and odd numbers exist in an array.

Logic:

1. Initialize `even_count = 0` and `odd_count = 0`.
2. Iterate through each element of the array.
3. Use the modulo operator (`% 2`):
   - If `arr[i] % 2 == 0`, increment `even_count`.
   - Else, increment `odd_count`.

Code: Count Even and Odd

```c
#include <stdio.h>
int main() {
    int arr[6] = {11, 22, 33, 44, 55, 66};
    int i, evenCount = 0, oddCount = 0;
    for(i = 0; i < 6; i++) {
        if(arr[i] % 2 == 0) {
            evenCount++;
        } else {
            oddCount++;
        }
    }
    printf("Total Even numbers: %d\n", evenCount);
    printf("Total Odd numbers: %d\n", oddCount);
    return 0;
}
```

Summary

What we covered today:

- We applied 1D array concepts to solve 5 practical problems.
- Learned how to aggregate data (Sum, Average).
- Learned how to find extremes (Max, Min).
- Performed a sequential lookup (Linear Search).
- Manipulated data structure directly (In-place reversal).
- Filtered array contents (Even/Odd counting).

Next Lecture Preview

Multi-Dimensional Arrays in C

- What if we need to store data in a grid or table format?
- Introduction to 2D arrays.
- Memory mapping of matrices.
- Matrix addition and traversal.

Unit 4: Array and Pointer

Focus: Declaring and initializing two-dimensional arrays of integers in C.

Agenda for Today

1. What is a Two-Dimensional Array?
2. Memory Layout of 2D Arrays
3. Syntax for Declaration
4. Various Methods of Initialization
5. Partial Initialization & Default Values
6. Summary and Best Practices

What is a Two-Dimensional Array?

- A **two-dimensional (2D) array** is an array of arrays.
- It organizes data in a tabular format, consisting of **rows** and **columns**.
- It is commonly used to represent grids, boards (like a chessboard), and matrices in mathematics.
- To identify any specific element, you need two indices: one for the row and one for the column.

Memory Layout of 2D Arrays

- Although we visualize a 2D array as a grid, computer memory is strictly linear.
- C uses **Row-Major Order** to store 2D arrays in memory.
- This means all elements of the first row are stored contiguously in memory, followed immediately by all elements of the second row, and so on.
- Example for a 2x3 array: Elements are stored in the sequence `(0,0)`, `(0,1)`, `(0,2)`, `(1,0)`, `(1,1)`, `(1,2)`.

Declaring a Two-Dimensional Array

The syntax for declaring a 2D array is similar to a 1D array, but it requires two sets of square brackets.

Syntax:

```
data_type array_name[row_size][column_size];
```

- `data_type`: The type of data to be stored (e.g., `int`, `float`).
- `array_name`: The identifier for the array.
- `row_size`: Total number of rows (must be a positive integer constant).
- `column_size`: Total number of columns (must be a positive integer constant).

Examples of 2D Array Declaration

```
// Declares an integer array with 3 rows and 4 columns
int matrix[3][4];
// Declares a float array for student marks (5 students, 3 subjects)
float marks[5][3];
// Declares a character array (often used for strings, but here as a
    grid)
char board[3][3];
```

- In `matrix[3][4]`, there are 3 * 4 = 12 `int` elements in total.
- If an `int` takes 4 bytes, `matrix` takes 12 * 4 = 48 bytes of memory.

Initialization of 2D Arrays (Method 1)

Fully Bracketed Initialization (Recommended)

- You can initialize a 2D array at the time of declaration using nested braces {}.
- Each inner set of braces represents a single row.

```
int matrix[2][3] = {
    {10, 20, 30}, // Row 0
    {40, 50, 60}  // Row 1
};
```

- This method is highly readable and directly mimics the grid structure.

Initialization of 2D Arrays (Method 2)

Flat Initialization

- You can also provide a single, flat list of values.
- Because C uses row-major order, it automatically fills the first row, then the second, and so on.

```
int matrix[2][3] = {10, 20, 30, 40, 50, 60};
```

- `10, 20, 30` go to row 0.
- `40, 50, 60` go to row 1.
- While valid, this is less readable than using nested braces.

Omitting Row Size during Initialization

- When initializing a 2D array at declaration, you **can omit the row size**, but you **must specify the column size**.
- The compiler automatically determines the number of rows based on the number of elements provided.

```
// Compiler determines row size is 2 based on 6 elements / 3 columns
int matrix[][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

- `int matrix[][] = { ... };` is **INVALID**. The column size is mandatory to know how long a row is in memory.

Partial Initialization

- If you provide fewer values than the total capacity, the remaining elements are automatically initialized to 0.

```
// 3x3 matrix, partially initialized
int grid[3][3] = {
    {1, 2},      // Row 0 gets 1, 2, 0
    {4}          // Row 1 gets 4, 0, 0
                 // Row 2 gets 0, 0, 0
};
```

Quick Zero Initialization:

```
// Initializes ALL elements to zero
int zeros[5][5] = {0};
```

Accessing Elements

- Elements in a 2D array are accessed using two indices: `array_name[row_index][column_index]`.
- **Important:** Both indices start at `0`.
- For an array declared as `int arr[R][C];`:
- Valid row indices: `0` to `R-1`
- Valid column indices: `0` to `C-1`

```
int matrix[2][2] = {
    {5, 10},
    {15, 20}
};
// matrix[0][0] is 5
// matrix[1][1] is 20
```

Code Example: Basic Declaration & Access

```c
#include <stdio.h>
int main() {
    // Declaration and Initialization
    int grid[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };
    // Accessing and printing specific elements
    printf("Element at (0,0): %d\n", grid[0][0]); // Output: 1
    printf("Element at (1,2): %d\n", grid[1][2]); // Output: 6
    // Modifying an element
    grid[0][1] = 99;
    printf("Modified element at (0,1): %d\n", grid[0][1]); // Output:
        99
    return 0;
}
```

Key Rules for 2D Arrays

1. **Two Dimensions:** Require two indices (row and column) for accessing elements.
2. **Contiguous Memory:** Stored sequentially in memory using row-major order.
3. **Sizes:** Both dimensions must be positive integers at declaration.
4. **Mandatory Column Size:** During initialization without explicit row size, column size must still be provided.
5. **Zero-Indexing:** Row and column indices always start at 0.

Common Mistakes to Avoid

- **Out of bounds access:** Accessing `arr[2][3]` when array is declared as `arr[2][3]` (valid indices are `0-1` and `0-2`).
- **Comma inside brackets:** Using `arr[1, 2]` instead of `arr[1][2]`. The former uses the comma operator and evaluates to `arr[2]`, which is incorrect.
- **Omitting column size:** Declaring `int arr[][] = {{1,2}, {3,4}};` will cause a compilation error.
- **Forgetting nested braces:** While flat initialization works, it easily leads to confusion about which element belongs where.

Next Lecture Preview

- **Processing Two-Dimensional Arrays**
- How to use nested `for` loops to read input into a 2D array.
- How to print a 2D array in a matrix format.
- Basic matrix operations (e.g., Matrix Addition).

Preparation: Review nested `for` loops, as they are essential for working with 2D arrays.

Two-Dimensional Arrays: Memory & Display

Understanding how 2D arrays are stored in memory and how to properly iterate over them to display their contents.

Agenda for Today

1. Quick Recap of 2D Arrays
2. Linear Memory Concept
3. Row-Major vs. Column-Major Order
4. Memory Layout of 2D Arrays in C
5. Memory Address Calculation
6. Displaying 2D Arrays (Nested Loops)
7. Formatting Output as a Matrix
8. Code Examples and Common Mistakes

Recap: Two-Dimensional Arrays

A 2D array is essentially an array of arrays.

```
// Declaration and Initialization
int matrix[2][3] = {
    {10, 20, 30},
    {40, 50, 60}
};
```

- `matrix` has 2 rows and 3 columns.
- Total elements = $2 \times 3 = 6$.
- Accessed using `matrix[row][col]`.

The Reality of Computer Memory

Although we visualize a 2D array as a grid or a table, **computer memory is linear** (1-dimensional).

- RAM is a continuous sequence of memory addresses.
- Therefore, a 2D grid must be flattened into a 1D sequence to be stored in RAM.
- How does the compiler map a 2D structure `(row, col)` to a 1D address?

Memory Mapping Strategies

There are two primary ways to flatten a 2D array:

1. **Row-Major Order**:
   - Elements are stored row by row.
   - The entire first row is stored in memory, followed by the entire second row, and so on.
2. **Column-Major Order**:
   - Elements are stored column by column.
   - Used in some other languages like Fortran or MATLAB.

Storing 2D Arrays in C (Row-Major)

The C language stores 2D arrays in Row-Major Order.

For `int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};`

Memory Layout:

`arr[0][0]` (1)
`arr[0][1]` (2)
`arr[0][2]` (3)
`arr[1][0]` (4)
`arr[1][1]` (5)
`arr[1][2]` (6)

Visualizing Memory Addresses

Assume a 32-bit system where `int` takes 4 bytes. Let base address be 1000.

```
int arr[2][3];
```

Element	Value	Address (Decimal)
`arr[0][0]`	10	1000
`arr[0][1]`	20	1004
`arr[0][2]`	30	1008
`arr[1][0]`	40	1012
`arr[1][1]`	50	1016
`arr[1][2]`	60	1020

Address Calculation Formula

How does the compiler find `arr[i][j]`?
For an array declared as `Type arr[ROWS][COLS];` with Base Address `B` and element size `S`:
Address of `arr[i][j]` = $B + (i \times COLS + j) \times S$
Example: `arr[1][2]` (Base 1000, COLS=3, S=4)

- Address = $1000 + (1 \times 3 + 2) \times 4$
- Address = $1000 + (5) \times 4 = 1020$

Displaying Array Elements

To print or process a 2D array, we must visit every element.
Since a 2D array has rows and columns, we use **nested loops**.

- The **outer loop** typically controls the rows.
- The **inner loop** typically controls the columns.

This perfectly matches C's row-major memory layout, ensuring efficient memory access.

Basic Loop Structure

```c
#include <stdio.h>
int main() {
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int i, j;
    for(i = 0; i < 2; i++) {        // Outer loop for rows
        for(j = 0; j < 3; j++) {    // Inner loop for columns
            printf("%d ", a[i][j]);
        }
    }
    return 0;
}
```

Output:

1 2 3 4 5 6

Formatting Output as a Matrix

To make it look like a 2D grid, print a newline (\n) after every row finishes.

```
    for(i = 0; i < 2; i++) {          // For each row...
        for(j = 0; j < 3; j++) {      // Print all columns in that row
            printf("%d ", a[i][j]);
        }
        printf("\n");                 // Move to the next line
    }
```

Output:

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

Alignment with Field Width

If numbers have different digit counts, the matrix will look misaligned. Use a format specifier with width (e.g., %4d).

```
int a[2][3] = {{1, 200, 3}, {40, 5, 6000}};
for(int i = 0; i < 2; i++) {
    for(int j = 0; j < 3; j++) {
        printf("%4d ", a[i][j]); // 4 spaces allocated
    }
    printf("\n");
}
```

Output:

```
   1  200    3
  40    5 6000
```

Common Mistakes

1. **Swapping bounds**: Using column count for the outer loop and row count for the inner loop, but keeping `arr[i][j]`.
   - Causes Out-of-Bounds memory access.
2. **Missing Newline**: Forgetting the `\n` after the inner loop makes the output hard to read.
3. **Redeclaring Array Sizes**: Do not redeclare array bounds in loop indices, e.g., using <= instead of <.
   - E.g., `for(i=0; i<=2; i++)` for a 2-row array is WRONG. Legal indices are 0 and 1.

Summary

- **Memory Layout**: C stores 2D arrays in contiguous, linear memory using Row-Major Order.
- **Address Calculation**: The compiler calculates memory offsets based on the base address, row index, column index, and total columns.
- **Traversal**: Nested loops are required to visit every element of a 2D array.
- **Formatting**: Output can be formatted as a neat grid using newlines and fixed-width format specifiers (%4d).

Concept of Pointer

Understanding memory addresses and pointer fundamentals in C.

Agenda

- Introduction to Memory and Addresses
- What is a Pointer?
- The Address-of Operator (&)
- Declaration of Pointers
- Initialization of Pointers
- The Dereference Operator (*)
- NULL Pointers

Previous Lecture Recap

- Passed individual array elements and entire arrays to functions.
- Modified array contents inside a function (pass by reference behavior).
- Passed 2D arrays to functions using specific syntax.

Memory Basics: Variables and Addresses

Every variable in C has two fundamental properties:
1. **Value**: The actual data stored in the variable.
2. **Address**: The physical location (memory address) where the data is stored in the computer's RAM.

Think of memory as a huge array of bytes, where each byte has a unique address (like a house number).

The Address-of Operator (&)

- To find out where a variable is stored, C provides the **Address-of operator** `&`.
- Placing `&` before a variable name gives its memory address.
- Addresses are typically displayed in hexadecimal format.
- We use the `%p` format specifier in `printf` to print an address.

Example: Printing Memory Addresses

```c
#include <stdio.h>
int main() {
    int num = 42;
    char ch = 'A';
    printf("Value of num: %d\n", num);
    /* Use %p for pointers/addresses, cast to void* for safety */
    printf("Address of num: %p\n", (void*)&num);
    printf("Value of ch: %c\n", ch);
    printf("Address of ch: %p\n", (void*)&ch);
    return 0;
}
```

What is a Pointer?

- A **pointer** is a special type of variable in C.
- Instead of storing a normal data value (like an integer or character), a pointer stores a **memory address**.
- Because it stores the address of another variable, we say it 'points to' that variable.
- Pointers provide indirect access to memory.

Declaration of Pointers

Syntax to declare a pointer:
`data_type *pointer_name;`

- `data_type`: The type of data the pointer will point to (e.g., `int`, `char`, `float`).
- `*` (asterisk): Indicates that the variable is a pointer.
- `pointer_name`: The name of the pointer variable.

Examples:

```
int *p; // Pointer to an integer
char *c; // Pointer to a character
float *f; // Pointer to a float
```

Initialization of Pointers

- Declaring a pointer does not automatically point it anywhere valid (it contains a garbage address).
- You must **initialize** it by assigning it a valid memory address, usually using the & operator.

```
int var = 100;
int *ptr;       /* Declaration */
ptr = &var;     /* Initialization */
```

- Now `ptr` points to `var`.

Example: Pointer Initialization

```c
#include <stdio.h>
int main() {
    int score = 85;
    int *ptr = &score; /* Declare and initialize */
    printf("Address of score: %p\n", (void*)&score);
    printf("Value stored in ptr: %p\n", (void*)ptr);
    return 0;
}
```

The Dereference Operator (*)

- Once a pointer holds an address, you can access or modify the value at that address using the **Dereference (or Indirection) operator** $*$.
- Placing $*$ before a pointer variable gets the value stored at the memory address it points to.

```
int var = 10;
int *p = &var;
printf("%d", *p); /* Prints 10 */
```

Example: Dereferencing a Pointer

```c
#include <stdio.h>
int main() {
    int age = 20;
    int *pAge = &age;
    printf("Direct access: %d\n", age);
    printf("Indirect access via pointer: %d\n", *pAge);
    /* Modifying the value using the pointer */
    *pAge = 21;
    printf("New age directly: %d\n", age);
    return 0;
}
```

NULL Pointers

- What if you declare a pointer but don't have an address to assign it yet?
- Assign it the macro `NULL` (defined in `<stdio.h>` and others).
- A **NULL pointer** points to nothing (address 0).

```
int *ptr = NULL;
if (ptr == NULL) {
    printf("Pointer is not pointing to anything yet.\n");
}
```

- Safe practice: Always initialize to `NULL` if not immediately assigning an address. Never dereference a NULL pointer!

Summary

- **Memory Addresses:** Variables are stored in memory; `&var` gives the address.
- **Pointer Concept:** A pointer is a variable that stores a memory address.
- **Declaration:** `int *p;` declares a pointer to an integer.
- **Initialization:** `p = &var;` assigns the address of `var` to `p`.
- **Dereferencing:** `*p` accesses or modifies the value at the address `p` holds.
- **NULL:** Use `NULL` for a pointer that points nowhere safely.

Next Lecture Preview

- Pointer arithmetic (adding and subtracting from pointers).
- Arrays of pointers.
- The deep relationship between arrays and pointers in C.
- How arrays decay into pointers.

Lecture 25: Introduction to String

Unit 5: Introduction to String, Structures, Function and File Operations
Topics Covered:

- Declaration of Strings
- Initialization of Strings
- Displaying Strings

Agenda for Today

- What is a String in C?
- The concept of the Null Character (\0)
- How to Declare a String
- Various methods to Initialize a String
- Memory representation of Strings
- How to display (print) a String
- How to read (input) a String
- Comprehensive Code Example

What is a String in C?

- In C, a **string** is not a built-in primitive data type (unlike `int`, `float`, or `char`).
- Instead, a string is defined as a **one-dimensional array of characters**.
- It represents a sequence of characters treated as a single data item.
- Example of strings: `"Hello"`, `"GTU"`, `"C Programming"`.

The Null Terminator (\0)

- The most important characteristic of a string in C is how it ends.
- Every valid C string is terminated by a special character called the **null character** (`'\0'`).
- The ASCII value of the null character is `0`.
- It tells the compiler and functions (like `printf`) where the string officially ends.
- Because of the null character, the size of the character array must be at least one greater than the number of characters in the string.

Declaration of a String

Since a string is an array of characters, we declare it just like an array of type `char`.

Syntax:

```
char string_name[size];
```

- `char`: Data type specifying the array holds characters.
- `string_name`: Name of the variable.
- `size`: The maximum number of characters the string can hold (including the null character).

Initialization of a String: Method 1

Character-by-Character Initialization

We can initialize a string by explicitly listing each character in a comma-separated list enclosed in curly braces.

Important: When doing this, we *must* manually add the null character `'\0'` at the end.

Initialization of a String: Method 2

String Literal Initialization

The most common and convenient way to initialize a string is by assigning a string literal (text enclosed in double quotes).

When using double quotes, the C compiler **automatically appends** the null character '\0' at the end.

Memory Layout of a String

Let's visualize the memory allocation for `char str[6] = "Hello";`

— Index — 0 — 1 — 2 — 3 — 4 — 5 —

———–———————————

— Value — 'H' — 'e' — 'l' — 'l' — 'o' — '\0' —

— Address— 1000 — 1001 — 1002 — 1003 — 1004 — 1005 —

- Each character takes 1 byte of memory.
- The string ends exactly where the \0 is placed.

Displaying a String using printf

To print a string using the standard `printf` function, we use the format specifier `%s`.

- `%s` tells `printf` to expect a string (a pointer to a character array).
- `printf` prints characters one by one until it encounters the `\0` character.

Displaying a String using puts

The `<stdio.h>` library provides a dedicated function for displaying strings: `puts()`.

- `puts(string_name)` outputs the string to the console.
- **Advantage:** It automatically appends a newline character (`\n`) after printing the string, unlike `printf`.

Reading a String using scanf

We can read a string from the user using `scanf` with the `%s` format specifier.

Note on `&`: Unlike other variables, we **do not** use the `&` (address-of) operator with string names in `scanf` because an array name is already a pointer to its first element.

Limitation: `scanf` stops reading as soon as it encounters a whitespace (space, tab, or newline).

Reading a String with Spaces

To read a string containing spaces, we should use `fgets()` instead of `scanf()`.

- `fgets(array_name, size, stdin)` reads an entire line of text until a newline or EOF is reached.
- It is safe because it limits the number of characters read, preventing buffer overflow.
- *(Historically, `gets()` was used, but it is dangerous and removed from modern C standards.)*

Complete Example Program

Let's combine declaration, reading, and displaying a string into a single program.

Summary

- A string is a 1D array of characters terminated by `\0`.
- The size of the array must be at least `length_of_string + 1`.
- Strings can be initialized character-by-character or using double quotes.
- `%s` is the format specifier for strings in `printf` and `scanf`.
- `scanf("%s")` cannot read strings with spaces.
- Use `fgets()` to safely read strings containing spaces.
- `puts()` provides a simple way to print a string with an automatic newline.

Standard Library String Functions

Exploring <string.h>: strlen(), strcpy(), strcat(), and strcmp().

Today's Agenda

- Introduction to `<string.h>` library
- Finding string length using `strlen()`
- Copying strings with `strcpy()`
- Concatenating strings with `strcat()`
- Comparing strings using `strcmp()`
- Common pitfalls and best practices

The <string.h> Library

- The C standard library provides built-in functions to manipulate strings.
- To use these functions, you must include the <string.h> header file.
- These functions generally operate on null-terminated character arrays ('\0').
- They provide efficient and standard ways to perform common string operations, saving programming time.

String Length: `strlen()`

Purpose: Calculates the length of a string.
Syntax: `size_t strlen(const char *str);`

- It counts the number of characters in a string up to, **but not including**, the null character (`'\0'`).
- It returns an unsigned integer type (`size_t`), which can be treated as a standard integer for most basic applications.

Example: Using strlen()

```c
#include <stdio.h>
#include <string.h>
int main() {
    char greeting[50] = "Hello, GTU!";
    int length;
    length = strlen(greeting);
    printf("The string is: %s\n", greeting);
    printf("Length of the string is: %d\n", length);
    return 0;
}
```

String Copy: `strcpy()`

Purpose: Copies a string from a source to a destination.
Syntax: `char strcpy(char dest, const char *src);`

- It copies the entire string pointed to by `src` into the array pointed to by `dest`.
- It also copies the terminating null character (`'\0'`).
- **Warning:** You must ensure the destination array is large enough to hold the copied string, including the null terminator.

Example: Using strcpy()

```c
#include <stdio.h>
#include <string.h>
int main() {
    char source[] = "Diploma Engineering";
    char destination[30];
    // Copying source to destination
    strcpy(destination, source);
    printf("Source string: %s\n", source);
    printf("Destination string: %s\n", destination);
    return 0;
}
```

String Concatenation: `strcat()`

Purpose: Appends (joins) one string to the end of another.
Syntax: `char strcat(char dest, const char *src);`

- It appends a copy of the `src` string to the end of the `dest` string.
- The initial character of `src` overwrites the null character at the end of `dest`.
- A new null character is appended at the end of the newly formed string.
- **Warning:** `dest` must have enough remaining space to hold the combined string.

Example: Using strcat()

```c
#include <stdio.h>
#include <string.h>
int main() {
    char firstName[20] = "John ";
    char lastName[] = "Doe";
    // Concatenating lastName to firstName
    strcat(firstName, lastName);
    printf("Full Name: %s\n", firstName);
    return 0;
}
```

String Comparison: `strcmp()`

Purpose: Compares two strings character by character.
Syntax: `int strcmp(const char str1, const char str2);`

- It compares the ASCII values of characters in `str1` and `str2`.
- The comparison is case-sensitive (e.g., 'A' is not equal to 'a').
- It returns an integer based on the comparison result.

Understanding `strcmp()` Return Values

- `0`: If `str1` and `str2` are exactly identical.
- `> 0` **(Positive)**: If the first non-matching character in `str1` has a **greater** ASCII value than the corresponding character in `str2`.
- `< 0` **(Negative)**: If the first non-matching character in `str1` has a **lesser** ASCII value than the corresponding character in `str2`.

Example: Using strcmp()

```c
#include <stdio.h>
#include <string.h>
int main() {
    char s1[] = "Hello";
    char s2[] = "Hello";
    char s3[] = "World";
    if (strcmp(s1, s2) == 0) {
        printf("s1 and s2 are equal.\n");
    }
    if (strcmp(s1, s3) != 0) {
        printf("s1 and s3 are not equal.\n");
    }
    return 0;
}
```

Important Safety Considerations

- **Buffer Overflows:** `strcpy` and `strcat` assume the destination array is large enough. If it's not, they overwrite adjacent memory, leading to crashes or security vulnerabilities.
- **Null Termination:** All these functions rely on strings ending with `\0`. If a string loses its null terminator, these functions will read/write out of bounds.
- **Safer Alternatives:** Modern C programming often uses `strncpy()` and `strncat()`, which allow specifying a maximum number of characters to copy/concatenate.

Summary of String Functions

- `#include <string.h>` is required for string functions.
- `strlen(s)`: Returns the number of characters in string `s`.
- `strcpy(d, s)`: Copies string `s` into string `d`.
- `strcat(d, s)`: Appends string `s` to the end of string `d`.
- `strcmp(s1, s2)`: Compares `s1` and `s2` lexicographically (returns 0 if equal).

Next Lecture Preview

- Introduction to User-Defined Functions
- Why we need functions
- Function declaration, definition, and calling
- Pass by value concept

Introduction to Structures in C

Unit 5: Introduction to String, Structures, Function and File Operations
Topic 5.2: Introduction to structures: Declaring, Initialization, Accessing
Understanding user-defined data types for complex data management.

Agenda

Need for Structures
What is a Structure?
Defining a Structure (Syntax & Examples)
Declaring Structure Variables
Memory Allocation for Structures
Initializing a Structure
Accessing Structure Members
Modifying Member Values
Complete Program Example
Summary

The Need for Structures

Arrays can only store elements of the **same data type** (homogeneous).
In real-world applications, we often need to group related data of **different types**.
Example: To represent a Student, we need:

- `int roll_no` (Integer)
- `char name[50]` (String)
- `float marks` (Floating-point)

Managing multiple independent variables or separate arrays for each attribute is difficult and error-prone.
Structures solve this by bundling them together.

What is a Structure?

A structure is a **user-defined data type** in C.
It allows combining data items of different kinds (int, float, char, arrays) under a single name.
The individual data items within a structure are called **members** or **fields**.
The `struct` keyword is used to define a structure.
Unlike arrays, structures represent a record of related information.

Defining a Structure: Syntax

A structure is defined using the `struct` keyword followed by a tag name.

```
struct structure_name {
    data_type member1;
    data_type member2;
    // ... more members
};
```

The definition is enclosed in curly braces {} and **must end with a semicolon** ;.
This definition usually goes outside and above the `main()` function so it can be accessed globally.

Example: Defining a Structure

Let's define a structure to represent a Student.

```
struct Student {
    int roll_no;
    char name[50];
    float marks;
};
```

- `struct` is the keyword.
- `Student` is the tag name.
- `roll_no`, `name`, and `marks` are structure members.

No memory is allocated at this stage. It only defines the 'shape' of the data.

Declaring Structure Variables

To use the structure, we must declare variables of its type.

Method 1: Along with definition

```
struct Student {
    int roll_no;
    float marks;
} s1, s2;
```

Method 2: Separately (Recommended)

```
struct Student {
    int roll_no;
    float marks;
};

int main() {
    struct Student s1, s2;
    // ...
}
```

Variables s1 and s2 now physically exist in memory.

Memory Allocation for Structures

When a structure variable is declared, memory is allocated.
The size of a structure is generally the sum of the sizes of its members.

```
struct Student {
    int roll_no;    // 4 bytes
    char name[20]; // 20 bytes
    float marks;    // 4 bytes
};
struct Student s1;
```

Total size for `s1` is approximately `4 + 20 + 4 = 28 bytes`.
(Note: Compilers may add 'padding' bytes for memory alignment, so the actual size might be slightly larger. Use `sizeof(struct Student)` to find the exact size.)

Initializing Structures

Structures can be initialized at the time of declaration.
Values are provided in a comma-separated list enclosed in curly braces {}.
The order of values must match the order of members in the structure definition.

Syntax & Example:

```
struct Student s1 = {101, "Ravi Kumar", 85.5};
```

- `roll_no` becomes `101`
- `name` becomes `"Ravi Kumar"`
- `marks` becomes `85.5`

Partial Initialization

If we provide fewer values than the number of members, the remaining members are automatically initialized to zero.

```
struct Student {
    int roll_no;
    char name[20];
    float marks;
};

// Partial initialization
struct Student s2 = {102, "Amit"};
```

Result:

- `s2.roll_no` is 102
- `s2.name` is `"Amit"`
- `s2.marks` is `0.000000` (automatically set to zero)

Accessing Structure Members

To read or modify a specific member of a structure, we use the **dot operator (.)**, also known as the member access operator.
Syntax: `structure_variable.member_name`
Example:

```
printf("Roll Number: %d\n", s1.roll_no);
printf("Marks: %.2f\n", s1.marks);
```

You cannot perform operations like `printf("%d", s1);` directly on the entire structure. You must access individual members.

Updating and Taking Input

You can assign new values or take input from the user for individual members.

Updating manually:

```
s1.marks = 92.5; // Changing marks
```

Taking input via scanf:

```
printf("Enter roll no: ");
scanf("%d", &s1.roll_no);

printf("Enter marks: ");
scanf("%f", &s1.marks);
```

Notice the & operator before `s1.roll_no`. We are passing the address of that specific member.

Handling Strings in Structures

Assigning values to character arrays (strings) in structures requires care.

Wrong way (after declaration):

```
s1.name = "Rahul"; // ERROR: Cannot assign to array directly
```

Correct way (using strcpy):

```
#include <string.h>
// ...
strcpy(s1.name, "Rahul");
```

Taking string input:

```
scanf("%s", s1.name); // No & needed for arrays
```

Complete Example Program

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

struct Book {
    int id;
    char title[50];
    float price;
};

int main() {
    struct Book b1;

    b1.id = 101;
    strcpy(b1.title, "C Programming");
    b1.marks = 250.50; // Oops, should be b1.price = 250.50;
    b1.price = 250.50;

    printf("Book ID: %d\n", b1.id);
    printf("Title: %s\n", b1.title);
    printf("Price: $%.2f\n", b1.price);

    return 0;
}
```

Summary

Structures group related, heterogeneous data into a single user-defined type.
The `struct` keyword is used to define the template (which ends with a ;).
Memory is only allocated when structure variables are declared.
Structures can be **initialized** at declaration using curly braces {}.
The **dot operator (.)** is essential for accessing, updating, or printing individual members.

Lecture 28: Introduction to Function

What is a Function?
Advantages of Using Functions
Library Functions
User-Defined Functions

Agenda

Concept and definition of a Function
Why do we need functions in C?
Broad classification: Types of Functions
Exploring Standard Library Functions
Introduction to User-Defined Functions (UDF)
Comparing Library vs. User-Defined Functions

What is a Function?

A function is a group of statements that together perform a specific task.
Every C program has at least one function, which is main().
You can divide up your code into separate functions based on logic.
Functions take inputs (parameters), process them, and return a result.

Why Use Functions?

Reusability: Write code once, use it multiple times.
Modularity: Breaks a large, complex problem into smaller, manageable chunks.
Readability: Makes the code cleaner and easier to understand.
Debugging: Easier to isolate and fix errors in individual functions.
Teamwork: Multiple programmers can work on different functions simultaneously.

Types of Functions in C

C functions are broadly categorized into two types:

1. Library Functions (Built-in functions)
   - Pre-defined in standard C libraries.
   - Ready to use.
2. User-Defined Functions
   - Created by the programmer to meet specific requirements.
   - Tailored to solve custom problems.

Library Functions

These are standard functions whose code is already written, compiled, and stored in C libraries.
To use them, you must include the corresponding header file using the #include directive.
Examples of header files: ¡stdio.h¿, ¡math.h¿, ¡string.h¿, ¡stdlib.h¿.
They perform common tasks like I/O operations, mathematical calculations, and string manipulation.

Common Library Functions

printf() and scanf() - Found in ¡stdio.h¿ for output and input.
strlen(), strcpy(), strcmp() - Found in ¡string.h¿ for string operations.
pow(), sqrt(), sin(), cos() - Found in ¡math.h¿ for mathematics.
malloc(), calloc(), exit() - Found in ¡stdlib.h¿ for memory management and utility.

Example: Using a Library Function

```c
#include <stdio.h>
#include <math.h>

int main() {
    double num = 16.0;
    // Using sqrt() library function from math.h
    double result = sqrt(num);
    printf("The square root of %.1f is %.1f\n", num, result);
    return 0;
}
```

User-Defined Functions (UDF)

Functions created by the programmer are called User-Defined Functions.
They allow you to abstract specific logic into a named block.
You define the function's name, return type, parameters, and the body (the actual code).
Requires three steps to use properly:

1. Function Declaration (Prototype)
2. Function Definition (Implementation)
3. Function Call (Execution)

The 3 Elements of a UDF

1. Function Declaration (Prototype): Tells the compiler about the function name, return type, and parameters. Usually placed above main().
Syntax: return_type function_name(type arg1, type arg2);
2. Function Call: Invoking the function to execute its code.
Syntax: function_name(val1, val2);
3. Function Definition: The actual block of code that performs the task.
Syntax: return_type function_name(type arg1, type arg2) { // logic }

Example: User-Defined Function

```c
#include <stdio.h>

// Function declaration
int addNumbers(int a, int b);

int main() {
    int sum = addNumbers(5, 7); // Function call
    printf("Sum: %d\n", sum);
    return 0;
}

// Function definition
int addNumbers(int a, int b) {
    return a + b;
}
```

Library vs. User-Defined Functions

Source: Library functions are provided by the compiler; UDFs are written by the programmer.
Header Files: Library functions require specific header files; UDFs generally do not (unless defined in a custom header).
Modification: You cannot modify the inner logic of a Library function. You have full control over a UDF.
Purpose: Library functions are for generic, common tasks. UDFs are for application-specific tasks.

Summary

Functions divide a program into smaller, reusable pieces.
main() is the essential function where execution begins.
Library functions (like printf, sqrt) are built-in tools accessed via header files.
User-Defined Functions (UDF) are created by programmers for custom logic.
A UDF requires a declaration, a call, and a definition.

Next Lecture Preview

Categories of User-Defined Functions based on arguments and return values.
Functions with no arguments and no return value.
Functions with arguments but no return value.
Functions with arguments and a return value.
The concept of 'Pass by Value'.

Lecture 29: Library Functions (Math & Character)

Unit 5: Introduction to String, Structures, Function and File Operations
Topics:

- 5.3.2.1 Math Functions (¡math.h¿)
- 5.3.2.2 Character Functions (¡ctype.h¿)

Agenda for Today

Introduction to C Standard Library
Mathematical Functions (¡math.h¿)
Common Math Functions (sqrt, pow, ceil, floor, abs)
Character Functions (¡ctype.h¿)
Testing Characters (isalpha, isdigit, etc.)
Converting Characters (toupper, tolower)

The C Standard Library

The C Standard Library is a collection of pre-written functions.
Instead of writing code from scratch, we include header files to use these functions.
Benefits:

- Saves development time.
- Highly optimized and bug-free.
- Standardized across all C compilers.

Mathematical Functions (¡math.h¿)

To use math functions, include the `<math.h>` header file.
Most math functions take `double` arguments and return `double`.
If you compile on Linux using gcc, you must link the math library using the `-lm` flag (e.g., `gcc program.c -lm`).

Common Math Functions: sqrt() and pow()

`sqrt(x)`: Returns the square root of x.
`pow(x, y)`: Returns x raised to the power of y (x^y).
Both x and y should be of type double.

Rounding Functions: ceil() and floor()

`ceil(x)`: Rounds x up to the nearest integer (returns double).
`floor(x)`: Rounds x down to the nearest integer (returns double).

Absolute Value Functions

`abs(x)`: Returns the absolute (positive) value of an integer. (Requires `<stdlib.h>`)
`fabs(x)`: Returns the absolute value of a floating-point number. (Requires `<math.h>`)

Other Useful Math Functions

`sin(x)`, `cos(x)`, `tan(x)`: Trigonometric functions (x must be in radians).
`log(x)`: Natural logarithm (base e).
`log10(x)`: Common logarithm (base 10).
`exp(x)`: Exponential function (e^x).

Character Functions (¡ctype.h¿)

The `<ctype.h>` library provides functions to test and convert single characters.
Characters in C are internally represented as integers (ASCII values).
These functions take an `int` (or `char`) as an argument and return an `int`.

Character Testing Functions

These functions return non-zero (true) if the condition is met, and 0 (false) otherwise.
`isalpha(c)`: Checks if `c` is an alphabet (a-z or A-Z).
`isdigit(c)`: Checks if `c` is a digit (0-9).
`isalnum(c)`: Checks if `c` is alphanumeric (alphabet or digit).

More Character Testing Functions

`islower(c)`: Checks if `c` is lowercase (a-z).
`isupper(c)`: Checks if `c` is uppercase (A-Z).
`isspace(c)`: Checks if `c` is a whitespace character (space, tab, newline).
`ispunct(c)`: Checks if `c` is a punctuation mark.

Character Conversion Functions

These functions return the converted character. If conversion isn't possible, they return the original character.
`toupper(c)`: Converts lowercase to uppercase.
`tolower(c)`: Converts uppercase to lowercase.

Practical Example: Toggling Case

Let's write a small loop that reads a string and flips the case of every letter.

Summary

C provides powerful standard libraries to save time.
<math.h> handles complex calculations like `sqrt`, `pow`, `ceil`, and `floor`.
<stdlib.h> provides `abs()` for integers.
<ctype.h> is used for testing (`isalpha`, `isdigit`) and converting (`toupper`, `tolower`) characters.
Always include the proper header file to prevent compiler warnings and errors.

Next Lecture Preview

Next time, we will begin exploring Structures.
We will learn how to group different data types together.
We will define our own composite data types using `struct`.

Introduction to File Operations

Unit 5: Introduction to String, Structures, Function and File Operations
Topic 5.4: Introduction to File Operations, fopen() in r/w modes, and fclose()
Semester 3 - Diploma Engineering

Agenda

What is a File in C?
The Concept of a FILE Pointer
Opening Files with `fopen()`
The 'w' (Write) Mode
The 'r' (Read) Mode
Closing Files with `fclose()`
Summary of File Operations
Full Course Recap

What is a File?

Volatile Memory (RAM):
So far, all variables and arrays we created were stored in RAM. When the program terminates, this data is lost.
Non-Volatile Memory (Disk):
A file is a container in the computer's storage devices used for storing data permanently.
File Handling in C:
C provides a set of library functions (in `<stdio.h>`) to read from and write to files, enabling persistent data storage.

The FILE Pointer

To perform file operations, C uses a special data type called `FILE`.
A `FILE` pointer is a pointer variable that keeps track of the file we are currently working with.

Declaration Syntax:

```
FILE *pointer_name;
```

Example:

```
FILE *fp;
```

The file pointer acts as the communication link between the C program and the file on the disk.

Opening a File: fopen()

Before reading from or writing to a file, it must be opened.
The `fopen()` function is used to open a file.
Syntax:
`FILE *fopen(const char filename, const char *mode);`
Arguments:
1. `filename`: A string representing the name of the file (e.g., `"data.txt"`).
2. `mode`: A string specifying the purpose (e.g., `"r"` for read, `"w"` for write).
Return Value:
Returns a valid `FILE *` if successful, or `NULL` if the file could not be opened.

The 'w' (Write) Mode

The `"w"` mode opens a file for writing.

Key behaviors of 'w' mode:

- **If the file does not exist**: A new empty file is automatically created.
- **If the file already exists**: Its previous contents are completely erased (overwritten) before new writing begins.
- The file pointer is positioned at the beginning of the file.

Use `"w"` carefully, as it destroys existing data in the file.

Example: Opening in 'w' Mode

```c
#include <stdio.h>

int main() {
    FILE *fp;
    fp = fopen("output.txt", "w");

    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    printf("File created and opened successfully in write mode.\n");

    /* File operations would go here */

    return 0;
}
```

The 'r' (Read) Mode

The `"r"` mode opens a file for reading only.

Key behaviors of 'r' mode:

- **If the file does not exist**: `fopen()` returns `NULL`.
- **If the file exists**: It opens the file and positions the file pointer at the beginning.

You cannot modify the contents of a file opened in `"r"` mode.

Example: Opening in 'r' Mode

```c
#include <stdio.h>

int main() {
    FILE *fp;
    fp = fopen("input.txt", "r");

    if (fp == NULL) {
        printf("Failed to open file. It may not exist.\n");
        return 1;
    }

    printf("File opened successfully for reading.\n");

    /* Reading operations would go here */

    return 0;
}
```

Closing a File: fclose()

After finishing file operations, the file must be closed using `fclose()`.

Syntax:

```
int fclose(FILE *stream);
```

Example:

```
fclose(fp);
```

Why close a file?

1. Flushes any remaining data from memory buffers to the disk.
2. Releases system resources tied to the file.
3. Prevents data corruption.

Complete Example: Open and Close

```c
#include <stdio.h>

int main() {
    FILE *file_ptr;

    /* Open file */
    file_ptr = fopen("data.txt", "w");
    if (file_ptr == NULL) {
        printf("Error!\n");
        return 1;
    }

    printf("File opened successfully.\n");

    /* Close file */
    fclose(file_ptr);
    printf("File closed successfully.\n");

    return 0;
}
```

Summary of File Operations

- **FILE pointer**: A structure pointer used to handle files.
- **fopen()**: Opens a file and returns a `FILE *`.
- **"w" mode**: Opens for writing (creates new or overwrites existing).
- **"r" mode**: Opens for reading (file must exist).
- **NULL check**: Always verify that `fopen()` succeeded.
- **fclose()**: Safely closes the file and saves data to disk.

Course Recap - Programming in C

Unit 1: Introduction to Computer & C (Logic, Flowcharts, basic syntax, compilation).
Unit 2: Data Types, Operators, & I/O (Variables, `printf`, `scanf`, arithmetic).
Unit 3: Control Flow (`if`, `switch`, `for`, `while`, `do-while`).
Unit 4: Arrays (1D, 2D arrays, basic sorting/searching).
Unit 5: Strings, Structures, Functions, and Files.
Congratulations on completing *Programming in C*! You now have the foundational skills to build software.