Department of Computer Science and Engineering

Session – 2024-2025

Semester – 2

Subject – Programming in C

Prepared by :
Manish Dongre
Lecturer (CSE)

Unit 1.0: Introduction to 'C' Programming

1.1. Program Logic Development using Algorithms and Flowcharts

Introduction:

Before writing any C program, it's crucial to plan and develop the program's logic. This involves understanding the problem, devising a step-by-step solution, and representing it visually. Two primary tools for this are algorithms and flowcharts.

1. Algorithms

- **Definition:** An algorithm is a well-defined, step-by-step procedure or a set of rules to solve a specific problem. It's a sequence of unambiguous instructions designed to produce a particular output from given inputs.
- **Characteristics of a Good Algorithm:**
  - **Unambiguous:** Each step should be clear and have only one interpretation.
  - **Input:** An algorithm should have zero or more well-defined inputs.
  - **Output:** An algorithm should produce one or more well-defined outputs.
  - **Finiteness:** An algorithm must terminate after a finite number of steps.
  - **Effectiveness:** Each step must be basic enough that it can be carried out in a finite amount of time.
  - **Generality:** The algorithm should be applicable to a range of inputs.
- **Representing Algorithms:**
  - Algorithms are usually written in plain English or in a pseudo-code, which resembles programming code but isn't meant for execution.
- **Example: Algorithm to find the sum of two numbers:**
  1. Start
  2. Read the first number (a).
  3. Read the second number (b).
  4. Calculate the sum (sum = a + b).
  5. Display the sum.
  6. End.
- **Advantages of Algorithms:**
  - Easy to understand and develop.
  - Language-independent.
  - Helps in debugging.
- **Disadvantages of Algorithms:**
  - Can be time-consuming to write for complex problems.
  - Lack of visual representation.

2. Flowcharts

- **Definition:** A flowchart is a graphical representation of an algorithm. It uses various symbols to depict the steps and flow of the program.
- **Flowchart Symbols:**
  - **Oval (Start/End):** Represents the beginning or end of a program.
  - **Rectangle (Process):** Represents a processing step, such as calculations or data manipulation.
  - **Parallelogram (Input/Output):** Represents input or output operations.
  - **Diamond (Decision):** Represents a decision point, where the flow depends on a condition.
  - **Arrow (Flow Line):** Indicates the direction of flow.
  - **Circle (Connector):** Connects different parts of the flowchart.
- **Example: Flowchart to find the sum of two numbers:**
  - (Oval) Start
  - (Parallelogram) Read a, b
  - (Rectangle) sum = a + b

    - (Parallelogram) Display sum
    - (Oval) End.
- **Advantages of Flowcharts:**
    - Provides a visual representation of the program logic.
    - Facilitates communication.
    - Helps in analyzing and designing programs.
- **Disadvantages of Flowcharts:**
    - Can be complex for large programs.
    - Difficult to modify.
    - Time-consuming to draw.

3. Relationship between Algorithms and Flowcharts

- Algorithms and flowcharts are complementary tools.
- An algorithm provides the logical steps, and a flowchart visualizes those steps.
- It's often helpful to write an algorithm first and then create a flowchart based on it.

4. Importance of Program Logic Development

- **Problem Solving:** Helps in breaking down complex problems into manageable steps.
- **Clarity:** Provides a clear understanding of the program's logic.
- **Efficiency:** Enables the development of efficient and optimized programs.
- **Maintainability:** Makes it easier to modify and maintain programs.
- **Debugging:** Simplifies the process of finding and fixing errors.

5. Steps in Program Logic Development

1. **Problem Definition:** Clearly understand the problem to be solved.
2. **Algorithm Design:** Develop a step-by-step solution.
3. **Flowchart Creation:** Visualize the algorithm using a flowchart.
4. **Code Implementation:** Translate the algorithm or flowchart into C code.
5. **Testing and Debugging:** Test the program and fix any errors.
6. **Documentation:** Document the program's logic and code.

6. Example of Algorithm and Flowchart Combination.

- **Problem:** Write a program to find the largest of three numbers.
- **Algorithm:**
    1. Start
    2. Read three numbers (a, b, c).
    3. If a > b and a > c, then display "a is the largest".
    4. Else if b > a and b > c, then display "b is the largest".
    5. Else display "c is the largest".
    6. End.
- **Flowchart:**
    - (Oval) Start
    - (Parallelogram) Read a, b, c
    - (Diamond) a > b and a > c ?
        - (Yes) (Parallelogram) Display "a is the largest"
        - (No) (Diamond) b > a and b > c ?
            - (Yes) (Parallelogram) Display "b is the largest"
            - (No) (Parallelogram) Display "c is the largest"
    - (Oval) End

1.2: Algorithm Development and Writing Algorithms Using Pseudo-Codes

1. Introduction to Algorithms

- **Definition:** An algorithm is a finite sequence of well-defined, computer-implementable instructions, typically used to solve a class of specific problems or to perform a computation.
- **Purpose:**
  - To provide a clear, step-by-step solution to a problem.
  - To serve as a blueprint for coding.
  - To facilitate communication of the solution between developers.
- **Essential Properties:**
  - **Input:** Zero or more inputs are supplied.
  - **Output:** At least one output is produced.
  - **Definiteness:** Each instruction is clear and unambiguous.
  - **Finiteness:** The algorithm terminates after a finite number of steps.
  - **Effectiveness:** Each instruction is basic enough to be carried out, in principle, by a person using pencil and paper.
  - **Generality:** The algorithm applies to a set of inputs.

2. Algorithm Development Process

1. **Problem Definition:**
   - Clearly understand the problem.
   - Identify inputs, desired outputs, and constraints.
2. **Algorithm Design:**
   - Break the problem into smaller, manageable sub-problems.
   - Develop a logical sequence of steps to solve each sub-problem.
   - Consider different approaches and choose the most efficient one.
3. **Algorithm Representation:**
   - Express the algorithm using a suitable method (e.g., pseudo-code, flowcharts).
4. **Algorithm Analysis:**
   - Evaluate the algorithm's efficiency (time and space complexity).
   - Check for correctness and completeness.
5. **Algorithm Refinement:**
   - Optimize the algorithm for better performance.
   - Make necessary adjustments based on analysis.
6. **Implementation (Coding):**
   - Translate the algorithm into a specific programming language (e.g., C).
7. **Testing and Debugging:**
   - Verify the correctness of the implemented algorithm.
   - Identify and fix any errors.

3. Pseudo-Code

- **Definition:** Pseudo-code is a human-readable description of an algorithm that resembles programming code but does not adhere to any specific programming language syntax.
- **Purpose:**
  - To express algorithms in a clear and concise manner.
  - To bridge the gap between natural language and programming code.
  - To facilitate algorithm design and communication.
- **Characteristics:**
  - Uses simple English-like statements.
  - Avoids strict syntax rules.
  - Focuses on the logical flow of the algorithm.
  - Can be easily translated into actual code.
- **Common Pseudo-Code Conventions:**

- **Keywords:** Use keywords like `READ`, `WRITE`, `IF`, `ELSE`, `WHILE`, `FOR`, `REPEAT`, `UNTIL`, `FUNCTION`, `RETURN`.
- **Indentation:** Use indentation to indicate blocks of code and control structures.
- **Variables:** Use descriptive variable names.
- **Arithmetic Operations:** Use standard arithmetic operators (+, -, *, /, %).
- **Comparison Operators:** Use comparison operators (>, <, ==, !=, >=, <=).
- **Assignment:** Use an arrow or equal sign for assignment (e.g., `variable ← value` or `variable = value`).
- **Comments:** Use comments to explain parts of the algorithm.

4. Developing and Writing Algorithms Using Pseudo-Codes

Example 1: Algorithm to find the largest of two numbers

Code snippet
```
ALGORITHM FindLargest
INPUT: num1, num2 (two numbers)
OUTPUT: largest (the larger number)

IF num1 > num2 THEN
    largest ← num1
ELSE
    largest ← num2
ENDIF

WRITE largest
END
```

Example 2: Algorithm to calculate the factorial of a number

Code snippet
```
ALGORITHM Factorial
INPUT: n (a non-negative integer)
OUTPUT: factorial (the factorial of n)

IF n < 0 THEN
    WRITE "Factorial is not defined for negative numbers"
ELSE
    factorial ← 1
    FOR i ← 1 TO n DO
        factorial ← factorial * i
    ENDFOR
    WRITE factorial
ENDIF
END
```

Example 3: Algorithm to find the sum of elements in an array

Code snippet
```
ALGORITHM ArraySum
INPUT: array (an array of numbers), size (the size of the array)
OUTPUT: sum (the sum of the array elements)

sum ← 0
FOR i ← 0 TO size - 1 DO
    sum ← sum + array[i]
ENDFOR
WRITE sum
END
```

5. Tips for Writing Effective Pseudo-Code

- **Keep it simple:** Use clear and concise language.

- **Be consistent:** Follow a consistent style and use keywords appropriately.
- **Focus on logic:** Emphasize the flow of the algorithm rather than syntax details.
- **Use indentation:** Improve readability by indenting blocks of code.
- **Add comments:** Explain complex parts of the algorithm.
- **Test with examples:** Walk through the algorithm with sample inputs to ensure correctness.

6. Advantages of Using Pseudo-Code

- **Readability:** Easy to understand and follow.
- **Language Independence:** Not tied to any specific programming language.
- **Flexibility:** Allows for easy modification and refinement.
- **Communication:** Facilitates communication between developers and non-programmers.
- **Planning:** Helps in planning and structuring the program logic.

1.3: Flowcharts - Definition, Importance, Symbols, and Limitations

1. Definition of Flowchart

- A flowchart is a graphical representation of an algorithm or process. It uses various symbols and connecting lines to illustrate the sequence of steps involved in solving a problem or completing a task.
- Flowcharts provide a visual overview, making it easier to understand the logic and flow of a program or process.

2. Importance of Flowcharts

- **Visual Communication:** Flowcharts provide a clear and concise way to communicate the logic of a program or process to others, including non-programmers.
- **Problem Analysis:** They help in breaking down complex problems into smaller, more manageable steps, making it easier to analyze and solve them.
- **Program Planning:** Flowcharts serve as a blueprint for coding, ensuring that the program logic is well-defined before implementation.
- **Documentation:** They provide valuable documentation for programs and processes, making it easier to maintain and modify them in the future.
- **Debugging:** Flowcharts can aid in identifying errors and logical flaws in a program or process.
- **Efficiency:** Flowcharts help in designing efficient algorithms by visualizing the flow and identifying potential bottlenecks.

3. Symbols of Flowchart

- **Terminals (Oval):**
  - Represent the start or end of a flowchart.
  - Indicate the beginning and termination points of a process.
- **Input/Output (Parallelogram):**
  - Represent input or output operations.
  - Indicate where data is received from or sent to external sources.
- **Processing (Rectangle):**
  - Represent processing steps, such as calculations, data manipulations, or assignments.
  - Indicate any operation that changes the data.
- **Decision (Diamond):**
  - Represent decision points, where the flow of the program depends on a condition.
  - Typically have two or more exit paths, corresponding to different outcomes.
- **Flow Lines (Arrows):**
  - Indicate the direction of flow in the flowchart.

    - Connect different symbols and show the sequence of steps.
- **Connection (Circle):**
    - Used to connect parts of a flowchart when the flow lines become too long or complex.
    - Help to maintain clarity and readability.
- **Off-Page Connectors (Pentagon/Special Shape):**
    - Used to connect parts of a flowchart that span multiple pages.
    - Help to maintain continuity and avoid clutter in large flowcharts.

4. Detailed Symbol Descriptions

- **Terminals (Oval):**
    - Example: "Start," "End," "Begin," "Stop."
    - Usage: Marks the beginning and end of the entire process or program.
- **Input/Output (Parallelogram):**
    - Example: "Read num1, num2," "Display result," "Print report."
    - Usage: Shows data entering (input) or data leaving (output) the program or process.
- **Processing (Rectangle):**
    - Example: "sum = num1 + num2," "calculate average," "update counter."
    - Usage: Represents any computation or data transformation.
- **Decision (Diamond):**
    - Example: "Is num1 > num2?," "Is counter < 10?," "Is file found?"
    - Usage: Shows a point where the flow branches based on a condition (usually yes/no or true/false).
- **Flow Lines (Arrows):**
    - Usage: Shows the sequence of steps, connecting the symbols in the order they are executed.
- **Connection (Circle):**
    - Usage: Used to connect parts of a flowchart on the same page, avoiding long flow lines.
- **Off-Page Connectors (Pentagon/Special Shape):**
    - Usage: Used to connect parts of a flowchart that are on different pages, maintaining continuity.

5. Limitations of Flowcharts

- **Complexity for Large Programs:** Flowcharts can become very complex and difficult to manage for large and intricate programs.
- **Time-Consuming:** Drawing and maintaining flowcharts can be time-consuming, especially for frequently changing programs.
- **Lack of Flexibility:** Flowcharts can be difficult to modify or update, particularly when significant changes are made to the program logic.
- **Limited Representation of Data Structures:** Flowcharts are not well-suited for representing complex data structures or algorithms involving intricate data manipulations.
- **Abstraction Level:** Flowcharts may not provide sufficient detail for certain programming tasks, requiring additional documentation or specifications.
- **Difficulty in Automated Generation:** Converting flowcharts directly into code can be challenging, as it requires manual interpretation and translation.
- **Maintenance Issues:** When programs are changed, flowcharts must also be updated, this can be easily overlooked.
- **Subjectivity:** The level of detail and style of flowcharts can vary, leading to inconsistencies and potential misunderstandings.

1.4: Basic Structure of a 'C' Program

Introduction:

A C program, regardless of its complexity, follows a specific basic structure. Understanding this structure is essential for writing well-organized and maintainable code.

Basic Structure of a C Program:

A typical C program consists of several sections, each serving a specific purpose. These sections, when arranged correctly, allow the C compiler to understand and execute the code.

1. Documentation Section (Comments):

- **Purpose:** This section is used to add comments to the code. Comments are non-executable statements that explain what the code does, who wrote it, and other relevant information.
- **Types of Comments:**
  - **Single-line comments:** Start with `//` and continue to the end of the line.
  - **Multi-line comments:** Enclosed within `/*` and `*/`.
- **Importance:**
  - Enhances code readability.
  - Aids in program maintenance.
  - Helps in debugging.
- **Example:**

C
```
// This is a single-line comment
/*
This is a multi-line comment.
It can span multiple lines.
*/
```

2. Preprocessor Directives (Link Section):

- **Purpose:** Preprocessor directives are instructions to the C preprocessor, which modifies the source code before compilation.
- **Common Directives:**
  - `#include`: Includes header files (e.g., `stdio.h`, `stdlib.h`).
  - `#define`: Defines macros (symbolic constants).
- **Importance:**
  - Includes necessary libraries for standard functions.
  - Defines constants for easy modification.
- **Example:**

C
```
#include <stdio.h>  // Includes standard input/output library
#include <stdlib.h> // Includes standard library
#define PI 3.14159   // Defines PI as a constant
```

3. Definition Section:

- **Purpose:** This section can be used to define symbolic constants or macros using the `#define` preprocessor directive.
- **Importance:**
  - Allows for easy replacement of values throughout the code.
  - Improves code readability.
- **Example:**

C
```
#define MAX_SIZE 100
```

4. Global Declaration Section:

- **Purpose:** This section declares global variables and user-defined functions that can be accessed from any part of the program.
- **Importance:**
  - Provides variables accessible to all functions.
  - Declares functions before their definition.
- **Example:**

C
```
int globalVariable; // Global variable declaration
void myFunction();    // Function declaration
```

5. Main Function Section:

- **Purpose:** The `main()` function is the entry point of every C program. Execution starts here.
- **Structure:**
  - `int main() { ... }` or `void main() { ... }`
  - Contains declaration and executable parts.
- **Declaration Part:** Declares variables used within the `main()` function.
- **Executable Part:** Contains the program's logic and statements.
- **Importance:**
  - Essential for program execution.
  - Organizes the program's primary logic.
- **Example:**

C
```
int main() {
    int localVariable; // Local variable declaration
    localVariable = 10; // Executable statement
    printf("Value: %d\n", localVariable); // Executable statement
    return 0; // Returns an integer value
}
```

6. Subprogram Section (User-Defined Functions):

- **Purpose:** This section defines user-defined functions, which are blocks of code that perform specific tasks.
- **Structure:**
  - `returnType functionName(parameters) { ... }`
- **Importance:**
  - Breaks down complex programs into smaller, manageable parts.
  - Promotes code reusability.
- **Example:**

C
```
void myFunction() {
    printf("This is a user-defined function.\n");
}
```

Complete Example:

C
```
#include <stdio.h>

#define MAX_VALUE 100
```

```
int globalVariable = 50;

void myFunction() {
    printf("Global variable: %d\n", globalVariable);
}

int main() {
    int localVariable = 20;
    printf("Local variable: %d\n", localVariable);
    myFunction();
    return 0;
}
```

Explanation of the example:

1. `#include <stdio.h>`: Includes the standard input/output library.
2. `#define MAX_VALUE 100`: Defines a constant `MAX_VALUE`.
3. `int globalVariable = 50;`: Declares a global variable.
4. `void myFunction()`: Defines a user-defined function.
5. `int main()`: The main function where the program execution starts.
6. `int localVariable = 20;`: Declares a local variable in the main function.
7. `printf()`: Prints values to the console.
8. `myFunction()`: Calls the user-defined function.
9. `return 0;`: Returns 0 to indicate successful execution.

Key Takeaways:

- A well-structured C program is easier to read, understand, and maintain.
- Comments are crucial for documenting code.
- Preprocessor directives are used for including libraries and defining constants.
- Global and local variables have different scopes.
- `main()` is the entry point, and user-defined functions enhance code modularity.

1.5: Data Concepts - Character Set, C Tokens, Keywords and Identifiers, Constants, Variables and its Declaration

1. Character Set

- **Definition:** The character set in C refers to the set of valid characters that can be used to write C programs.
- **Categories:**
  - **Letters:**
    - Uppercase: A-Z
    - Lowercase: a-z
  - **Digits:** 0-9
  - **Special Characters:**
    - Arithmetic operators: +, -, *, /, %
    - Relational operators: <, >, ==, !=, >=, <=
    - Logical operators: &&, ||, !
    - Assignment operators: =, +=, -=, *=, /=, %=
    - Punctuation marks: ;, :, ?, ,, ., ', "
    - Special symbols: #, @, $, &, _, ~, |, ^, !, \
  - **White Spaces:**
    - Space, tab, newline, vertical tab, form feed
- **Importance:**
  - Forms the basic building blocks of C programs.

- Ensures that the compiler can correctly interpret the code.

2. C Tokens

- **Definition:** A token is the smallest individual unit in a C program. The compiler recognizes tokens as the fundamental units of source code.
- **Categories:**
  - **Keywords:** Reserved words with special meanings.
  - **Identifiers:** Names given to variables, functions, etc.
  - **Constants:** Fixed values that do not change during program execution.
  - **Strings:** Sequences of characters enclosed in double quotes.
  - **Operators:** Symbols that perform operations.
  - **Special Characters:** Punctuation marks and other symbols.
- **Importance:**
  - Essential for the compiler to parse and understand the code.
  - Forms the structure of the C language.

3. Keywords and Identifiers

- **Keywords:**
  - **Definition:** Reserved words that have predefined meanings in the C language.
  - **Examples:** `int`, `float`, `char`, `if`, `else`, `while`, `for`, `return`, `void`, `static`, `const`.
  - **Rules:**
    - Cannot be used as variable names or identifiers.
    - Are always written in lowercase.
- **Identifiers:**
  - **Definition:** Names given to variables, functions, arrays, structures, etc., by the programmer.
  - **Rules:**
    - Must start with a letter or underscore (_).
    - Can contain letters, digits, and underscores.
    - Case-sensitive (e.g., `variable` and `Variable` are different).
    - Cannot be keywords.
    - Should not contain white spaces.
    - Length should be less than or equal to 31 characters.
  - **Examples:** `myVariable`, `calculateSum`, `_count`, `studentName`.
- **Importance:**
  - Keywords provide the structure and syntax of the language.
  - Identifiers allow programmers to name and reference program elements.

4. Constants

- **Definition:** Fixed values that do not change during program execution.
- **Types:**
  - **Integer Constants:** Whole numbers without decimal points (e.g., 10, -5, 0).
  - **Floating-Point Constants:** Numbers with decimal points (e.g., 3.14, -2.5, 0.0).
  - **Character Constants:** Single characters enclosed in single quotes (e.g., 'A', 'b', '9').
  - **String Constants:** Sequences of characters enclosed in double quotes (e.g., "Hello", "C Programming").
  - **Enumeration Constants:** Named integer constants.
  - **Symbolic Constants:** Defined using `#define` preprocessor directive or `const` keyword.
- **Importance:**
  - Provide fixed values for calculations and comparisons.
  - Improve code readability and maintainability.

5. Variables and its Declaration

- **Variables:**

    - **Definition:** Named memory locations used to store data that can change during program execution.
    - **Declaration:** The process of specifying the data type and name of a variable before it is used.
    - **Syntax:** `data_type variable_name;`
    - **Initialization:** Assigning an initial value to a variable at the time of declaration.
    - **Syntax:** `data_type variable_name = value;`
    - **Examples:**
        - `int age;`
        - `float salary = 2500.50;`
        - `char grade = 'A';`
- **Data Types:**
    - **Basic Data Types:**
        - `int`: Integer (e.g., -10, 0, 25).
        - `float`: Floating-point number (e.g., 3.14, -2.5).
        - `double`: Double-precision floating-point number.
        - `char`: Character (e.g., 'A', 'b', '9').
        - `void`: Represents the absence of a value.
    - **Derived Data Types:**
        - Arrays, pointers, structures, unions, enumerations.
- **Importance:**
    - Variables allow programs to store and manipulate data.
    - Data types ensure that data is stored and processed correctly.
    - Declaration is necessary for the compiler to allocate memory and perform type checking.

Example Code Snippet:

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

#define PI 3.14159

int main() {
    int age = 30;
    float radius = 5.5;
    char grade = 'A';
    char name[] = "John Doe";

    printf("Age: %d\n", age);
    printf("Radius: %f\n", radius);
    printf("Grade: %c\n", grade);
    printf("Name: %s\n", name);
    printf("PI: %f\n", PI);

    return 0;
}
```

Key Takeaways:

- Understanding the character set, tokens, keywords, identifiers, constants, and variables is fundamental to C programming.
- Proper declaration and use of variables and constants ensures that the program behaves as expected.
- Data types are important for memory management and correct data handling.

1.6: Data Types and Data Type Conversion

1. Data Types in C

- **Definition:** Data types specify the type of data that a variable can hold. They determine the size of memory allocated to the variable and the operations that can be performed on it.
- **Basic (Primitive) Data Types:**

    - `int`: Integer (whole numbers, e.g., -10, 0, 25).
    - `float`: Floating-point number (single-precision, e.g., 3.14, -2.5).
    - `double`: Floating-point number (double-precision, e.g., 3.14159265359).
    - `char`: Character (single byte, e.g., 'A', 'b', '9').
    - `void`: Represents the absence of a type.
- **Derived Data Types:**
    - Arrays: Collections of elements of the same data type.
    - Pointers: Variables that store memory addresses.
    - Structures: User-defined data types that group related variables.
    - Unions: Similar to structures, but members share the same memory location.
    - Enumerations (enums): User-defined data types that assign names to integer constants.
- **Modifiers:**
    - `signed`, `unsigned`: Modify integer types to allow or disallow negative values.
    - `short`, `long`: Modify integer types to change their size.
    - `long double`: Modifies double to increase its precision.

2. Data Type Conversion

- **Definition:** Data type conversion (or type casting) is the process of changing the data type of a value or variable.
- **Types of Data Type Conversion:**
    - **Implicit Conversion (Automatic Conversion):**
        - Performed automatically by the compiler.
        - Occurs when operands of different data types are used in an expression.
        - Generally, the lower data type is promoted to the higher data type.
        - Example: `int` to `float`, `float` to `double`.
    - **Explicit Conversion (Type Casting):**
        - Performed explicitly by the programmer using the cast operator.
        - Syntax: `(data_type) expression;`
        - Allows for more control over the conversion.
        - Example: `(int) 3.14`, `(float) 10`.

3. Implicit Conversion (Automatic Conversion) Detailed

- **Rules:**
    - When an arithmetic operation is performed between an integer and a floating-point number, the integer is automatically converted to a floating-point number.
    - When assigning a value of one data type to a variable of another data type, the value is automatically converted to the variable's data type, if possible.
    - The compiler generally converts from a lower data type to a higher data type to prevent loss of data.
- **Example:**

C
```
#include <stdio.h>

int main() {
    int integerValue = 10;
    float floatValue = 5.5;
    float result;

    result = integerValue + floatValue; // integerValue is implicitly converted to
float

    printf("Result: %f\n", result); // Output: Result: 15.500000

    return 0;
}
```

4. Explicit Conversion (Type Casting) Detailed

- **Purpose:**
  - To force a conversion when the compiler does not perform automatic conversion.
  - To prevent loss of data or precision.
  - To perform specific conversions for certain operations.
- **Example:**

C
```
#include <stdio.h>

int main() {
    float floatValue = 3.14;
    int integerValue;

    integerValue = (int) floatValue; // Explicitly cast float to int

    printf("IntegerValue: %d\n", integerValue); // Output: IntegerValue: 3

    return 0;
}
```

- **Example 2:**

C
```
#include <stdio.h>

int main(){
    int a = 10;
    int b = 3;
    float result;

    result = (float) a / b; // a is casted to float, so the division is performed in
float.

    printf("Result: %f\n", result); // result = 3.333333
    return 0;
}
```

5. Potential Issues with Type Conversion

- **Loss of Data:** Converting from a higher data type to a lower data type can result in loss of data (e.g., converting `float` to `int` truncates the decimal part).
- **Loss of Precision:** Converting from `double` to `float` can result in loss of precision.
- **Overflow:** Converting a large value to a smaller data type can result in overflow, where the value wraps around.
- **Undefined Behavior:** Some conversions can lead to undefined behavior, which can cause unpredictable results.

6. Best Practices

- Use explicit type casting when necessary to control the conversion.
- Be aware of potential data loss or precision loss when performing conversions.
- Avoid unnecessary conversions.
- Use appropriate data types for the data being stored and manipulated.

Key Takeaways:

- Data types define the kind of data a variable can hold.
- Implicit conversion occurs automatically, while explicit conversion requires the cast operator.
- Be mindful of potential issues like data loss and overflow when performing type conversions.

- Understanding data types and their conversions is crucial for writing correct and efficient C programs.

1.7: Operators and its Types

1. Introduction to Operators

- **Definition:** Operators are symbols that tell the compiler to perform specific mathematical or logical manipulations.
- **Purpose:**
  - To perform operations on variables and constants.
  - To control the flow of a program.
  - To manipulate data at the bit level.

2. Types of Operators

2.1. Arithmetic Operators

- **Purpose:** Perform basic mathematical operations.
- **Operators:**
  - `+` (Addition)
  - `-` (Subtraction)
  - `*` (Multiplication)
  - `/` (Division)
  - `%` (Modulus - remainder of integer division)
- **Example:**

C

```
int a = 10, b = 5, result;
result = a + b; // result = 15
result = a - b; // result = 5
result = a * b; // result = 50
result = a / b; // result = 2
result = a % b; // result = 0
```

2.2. Relational Operators

- **Purpose:** Compare two operands and return a Boolean value (true or false).
- **Operators:**
  - `==` (Equal to)
  - `!=` (Not equal to)
  - `>` (Greater than)
  - `<` (Less than)
  - `>=` (Greater than or equal to)
  - `<=` (Less than or equal to)
- **Example:**

C

```
int a = 10, b = 5;
int result;
result = (a == b); // result = 0 (false)
result = (a != b); // result = 1 (true)
result = (a > b);  // result = 1 (true)
result = (a < b);  // result = 0 (false)
```

2.3. Logical Operators

- **Purpose:** Combine or negate Boolean expressions.
- **Operators:**
  - `&&` (Logical AND)
  - `||` (Logical OR)
  - `!` (Logical NOT)
- **Example:**

C
```
int a = 10, b = 5;
int result;
result = (a > 5 && b < 10); // result = 1 (true)
result = (a > 15 || b < 10); // result = 1 (true)
result = !(a == b);         // result = 1 (true)
```

2.4. Assignment Operators

- **Purpose:** Assign values to variables.
- **Operators:**
  - `=` (Simple assignment)
  - `+=` (Add and assign)
  - `-=` (Subtract and assign)
  - `*=` (Multiply and assign)
  - `/=` (Divide and assign)
  - `%=` (Modulus and assign)
- **Example:**

C
```
int a = 10;
a += 5; // a = 15 (a = a + 5)
a -= 3; // a = 12 (a = a - 3)
```

2.5. Increment and Decrement Operators

- **Purpose:** Increase or decrease the value of a variable by 1.
- **Operators:**
  - `++` (Increment)
  - `--` (Decrement)
- **Types:**
  - **Prefix:** `++a`, `--a` (increment/decrement before using the value)
  - **Postfix:** `a++`, `a--` (increment/decrement after using the value)
- **Example:**

C
```
int a = 10, b;
b = ++a; // a = 11, b = 11 (prefix)
b = a++; // b = 10, a = 11 (postfix)
```

2.6. Conditional (Ternary) Operator

- **Purpose:** A shorthand for `if-else` statements.
- **Operator:** `?:`
- **Syntax:** `condition ? expression1 : expression2;`
- **Example:**

C
```
int a = 10, b = 5, max;
max = (a > b) ? a : b; // max = 10
```

2.7. Bitwise Operators

- **Purpose:** Perform operations on individual bits of integers.
- **Operators:**
  - `&` (Bitwise AND)
  - `|` (Bitwise OR)
  - `^` (Bitwise XOR)
  - `~` (Bitwise NOT)
  - `<<` (Left shift)
  - `>>` (Right shift)
- **Example:**

C

```
int a = 60;  // 0011 1100
int b = 13;  // 0000 1101
int result;
result = a & b;  // 0000 1100 (12)
result = a | b;  // 0011 1101 (61)
result = a ^ b;  // 0011 0001 (49)
result = ~a;     // 1100 0011 (-61)
result = a << 2; // 1111 0000 (240)
result = a >> 2; // 0000 1111 (15)
```

2.8. Special Operators

- **Comma Operator (`,`):**
  - Evaluates multiple expressions and returns the value of the last expression.
  - **Example:** `int a, b; a = (b = 5, b + 10);` (a becomes 15)
- **Sizeof Operator (`sizeof`):**
  - Returns the size of a variable or data type in bytes.
  - **Example:** `sizeof(int), sizeof(a)`
- **Pointer Operators (`*`, `&`):**
  - `&` (Address-of operator): Returns the memory address of a variable.
  - `*` (Dereference operator): Accesses the value stored at a memory address.
  - **Example:** `int a = 10; int *ptr = &a;`
- **Member Selection Operators (`.`, `->`):**
  - `.` (Dot operator): Accesses members of a structure or union.
  - `->` (Arrow operator): Accesses members of a structure or union through a pointer.
  - Example: structure and pointer to structure.

Key Takeaways:

- Operators are essential for performing operations in C.
- Understanding the different types of operators is crucial for writing effective C programs.
- Bitwise operators allow for low-level manipulation of data.
- Special operators provide additional functionality for specific tasks.

1.8: Input/Output Functions - printf(), scanf(), getch(), putch(), getchar()

1. Introduction to Input/Output Functions

- Input/output (I/O) functions are essential for interacting with the user and external devices.
- They allow programs to receive data (input) and display results (output).
- The standard I/O library (`stdio.h`) provides many useful I/O functions.

2. printf() Function

- **Purpose:** Formatted output to the standard output (console).

- **Syntax:** `int printf(const char *format, ...);`
- **Format Specifiers:**
  - `%d` or `%i`: Integer
  - `%f`: Floating-point number (float)
  - `%lf`: Floating-point number (double)
  - `%c`: Character
  - `%s`: String
  - `%p`: Pointer address
  - `%%`: Prints a literal `%`
- **Escape Sequences:**
  - `\n`: Newline
  - `\t`: Tab
  - `\\`: Backslash
  - `\"`: Double quote
  - `\'`: Single quote
- **Example:**

C

```c
#include <stdio.h>

int main() {
    int age = 30;
    float salary = 2500.50;
    char grade = 'A';
    char name[] = "John Doe";

    printf("Name: %s\n", name);
    printf("Age: %d, Salary: %.2f, Grade: %c\n", age, salary, grade);
    return 0;
}
```

3. scanf() Function

- **Purpose:** Formatted input from the standard input (keyboard).
- **Syntax:** `int scanf(const char *format, ...);`
- **Format Specifiers:** Similar to `printf()`.
  - Note: when scanning strings, `scanf()` will read until the first whitespace.
- **Address-of Operator (`&`):** Used to provide the memory address of the variable where the input should be stored.
- **Example:**

C

```c
#include <stdio.h>

int main() {
    int age;
    float salary;
    char name[50];

    printf("Enter your name: ");
    scanf("%s", name);

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

    printf("Enter your salary: ");
    scanf("%f", &salary);

    printf("Name: %s, Age: %d, Salary: %.2f\n", name, age, salary);
    return 0;
}
```

4. getch() Function

- **Purpose:** Reads a single character from the keyboard without echoing it to the console.
- **Syntax:** `int getch(void);`
- **Header File:** `conio.h` (non-standard, often used in Windows)
- **Usage:** Commonly used for reading passwords or single-character input without displaying it.
- **Example:**

C

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

int main() {
    char ch;

    printf("Enter a character: ");
    ch = getch();

    printf("\nYou entered: %c\n", ch);
    return 0;
}
```

5. putch() Function

- **Purpose:** Writes a single character to the standard output (console).
- **Syntax:** `int putch(int character);`
- **Header File:** `conio.h` (non-standard, often used in Windows)
- **Usage:** Used to display single characters.
- **Example:**

C

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

int main() {
    char ch = 'A';

    putch(ch);
    putch('\n'); // Newline

    return 0;
}
```

6. getchar() Function

- **Purpose:** Reads a single character from the standard input (keyboard) and echoes it to the console.
- **Syntax:** `int getchar(void);`
- **Header File:** `stdio.h`
- **Usage:** Used to read characters, including whitespace characters.
- **Example:**

C

```
#include <stdio.h>

int main() {
    char ch;

    printf("Enter a character: ");
    ch = getchar();

    printf("You entered: %c\n", ch);
    return 0;
}
```

Key Differences and Considerations:

- **`scanf()` vs. `getchar()`:**
  - `scanf()` is used for formatted input (multiple values, specific data types).
  - `getchar()` is used for reading single characters, including whitespace.
- **`getch()` vs. `getchar()`:**
  - `getch()` does not echo the character to the console and is often platform-dependent.
  - `getchar()` echoes the character to the console and is part of the standard C library.
- **`putch()` vs. `printf()`:**
  - `putch()` is used for displaying single characters.
  - `printf()` is used for formatted output (multiple values, strings).
- **Portability:** `conio.h` functions (`getch()`, `putch()`) are not part of the standard C library and may not be available on all systems (especially non-Windows systems).

Example of combining multiple functions:

C

```
#include <stdio.h>

int main() {
    char name[50];
    int age;

    printf("Enter your name: ");
    scanf("%s", name);

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

    printf("Name: ");
    for(int i = 0; name[i] != '\0'; i++){
        putch(name[i]);
    }
    putch('\n');
    printf("Age: %d\n", age);

    return 0;
}
```

These functions are fundamental for creating interactive C programs. Understanding their usage and differences is essential for effective programming.

Unit 2.0 - Decision Making and Branching Statements

2.1 Introduction of Decision Making Statements in 'C'

Decision making statements in C enable the program to evaluate conditions and alter the flow of execution based on the outcome of these evaluations. Essentially, they allow the program to "choose" which path to take.

Why are decision making statements necessary?

- **Conditional Execution:** To execute specific blocks of code only when certain conditions are met.
- **Program Logic:** To implement complex logic where different actions need to be performed under different circumstances.
- **User Input Handling:** To respond differently based on the input provided by the user.
- **Error Handling:** To execute specific code to handle errors or unexpected situations.

C provides several keywords to implement decision making:

- `if`
- `else`
- `else if`
- `switch`

2.2 Decision Making with IF Statement

The `if` statement is the most basic decision-making statement in C. It allows a block of code to be executed only if a specified condition evaluates to true.

Syntax of the Simple IF Statement:

```
if (condition) {
    statement(s); // Block of code to be executed if the condition is true
}
// Statements after the if block
```

Explanation:

1. The `if` keyword marks the beginning of the decision-making statement.
2. The `condition` inside the parentheses is an expression that is evaluated. This expression typically involves relational operators (==, !=, >, <, >=, <=) and/or logical operators (`&&` (AND), `||` (OR), `!` (NOT)).
3. If the `condition` evaluates to a **non-zero value (considered true)**, the `statement(s)` inside the curly braces `{}` (the `if` block) are executed.
4. If the `condition` evaluates to **zero (considered false)**, the `statement(s)` inside the `if` block are skipped, and the program execution continues with the statements after the `if` block.
5. The curly braces `{}` are optional if the `if` block contains only a single statement. However, it is **highly recommended** to always use curly braces for better readability and to easily add more statements to the block later without introducing errors.

Example 1: Checking if a number is positive

```
#include <stdio.h>

int main() {
    int num;

    printf("Enter an integer: ");
    scanf("%d", &num);
```

```
    if (num > 0) {
        printf("%d is a positive number.\n", num);
    }

    printf("This statement is always executed.\n");

    return 0;
}
```

Output (if user enters 5):

```
Enter an integer: 5
5 is a positive number.
This statement is always executed.
```

Output (if user enters -3):

```
Enter an integer: -3
This statement is always executed.
```

Example 2: Checking if a character is a vowel (lowercase)

```
#include <stdio.h>

int main() {
    char ch;

    printf("Enter a lowercase character: ");
    scanf("%c", &ch);

    if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
        printf("%c is a lowercase vowel.\n", ch);
    }

    printf("End of program.\n");

    return 0;
}
```

Output (if user enters 'e'):

```
Enter a lowercase character: e
e is a lowercase vowel.
End of program.
```

Output (if user enters 'b'):

```
Enter a lowercase character: b
End of program.
```

The IF...ELSE Statement

The `if...else` statement provides a way to execute one block of code if a condition is true and another block of code if the condition is false.

Syntax of the IF...ELSE Statement:

```
if (condition) {
    statement(s)-1; // Block of code to be executed if the condition is true
} else {
    statement(s)-2; // Block of code to be executed if the condition is false
}
// Statements after the if...else block
```

Explanation:

1. The `if` part is the same as the simple `if` statement. The `condition` is evaluated.
2. If the `condition` evaluates to **true (non-zero)**, the `statement(s)-1` inside the `if` block are executed, and the `statement(s)-2` inside the `else` block are skipped.
3. If the `condition` evaluates to **false (zero)**, the `statement(s)-1` inside the `if` block are skipped, and the `statement(s)-2` inside the `else` block are executed.
4. After either the `if` block or the `else` block is executed, the program control flows to the statements after the `if...else` block.

Example 3: Checking if a number is even or odd

```
#include <stdio.h>

int main() {
    int num;

    printf("Enter an integer: ");
    scanf("%d", &num);

    if (num % 2 == 0) {
        printf("%d is an even number.\n", num);
    } else {
        printf("%d is an odd number.\n", num);
    }

    printf("Program finished.\n");

    return 0;
}
```

Output (if user enters 10):

```
Enter an integer: 10
10 is an even number.
Program finished.
```

Output (if user enters 7):

```
Enter an integer: 7
7 is an odd number.
Program finished.
```

Example 4: Checking if a character is a vowel or a consonant (lowercase)

```
#include <stdio.h>

int main() {
    char ch;

    printf("Enter a lowercase character: ");
    scanf("%c", &ch);

    if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
        printf("%c is a lowercase vowel.\n", ch);
    } else {
        printf("%c is a lowercase consonant.\n", ch);
    }

    printf("End of program.\n");

    return 0;
}
```

Output (if user enters 'i'):

```
Enter a lowercase character: i
i is a lowercase vowel.
End of program.
```

Output (if user enters 'z'):

```
Enter a lowercase character: z
z is a lowercase consonant.
End of program.
```

Nesting of IF...ELSE Statement

Nesting of `if...else` statements means placing one `if` statement (or `if...else` statement) inside the block of another `if` or `else` statement. This allows for more complex decision-making structures where multiple conditions need to be evaluated in a hierarchical manner.

Syntax of Nested IF...ELSE Statement:

```
if (condition1) {
    // Statements executed if condition1 is true
    if (condition2) {
        // Statements executed if condition1 is true AND condition2 is true
    } else {
        // Statements executed if condition1 is true AND condition2 is false
    }
} else {
    // Statements executed if condition1 is false
    if (condition3) {
        // Statements executed if condition1 is false AND condition3 is true
    } else {
        // Statements executed if condition1 is false AND condition3 is false
    }
}
// Statements after the nested if...else block
```

The level of nesting can go deeper, but it's generally recommended to avoid excessive nesting as it can make the code difficult to read and understand.

Example 5: Finding the largest of three numbers

```
#include <stdio.h>

int main() {
    int num1, num2, num3;

    printf("Enter three integers: ");
    scanf("%d %d %d", &num1, &num2, &num3);

    if (num1 >= num2) {
        if (num1 >= num3) {
            printf("%d is the largest number.\n", num1);
        } else {
            printf("%d is the largest number.\n", num3);
        }
    } else { // num1 < num2
        if (num2 >= num3) {
            printf("%d is the largest number.\n", num2);
        } else {
            printf("%d is the largest number.\n", num3);
        }
    }

    return 0;
}
```

Output (if user enters 10 5 8):

```
Enter three integers: 10 5 8
10 is the largest number.
```

Output (if user enters 3 15 7):

```
Enter three integers: 3 15 7
15 is the largest number.
```

Output (if user enters 2 4 20):

```
Enter three integers: 2 4 20
20 is the largest number.
```

Important Note on Dangling Else: In nested `if` statements without explicit curly braces, an `else` clause is always associated with the nearest preceding `if` that doesn't already have an `else`. This can lead to logical errors if not careful. **Always use curly braces to clearly define the blocks.**

The ELSE IF Ladder

The `else if` ladder (also known as the `if-else-if` ladder or `if-elif-else` structure in other languages) is used to test multiple conditions in a sequential manner. It provides a more organized way to handle multiple mutually exclusive choices compared to deeply nested `if...else` statements.

Syntax of the ELSE IF Ladder:

```
if (condition1) {
    statement(s)-1; // Executed if condition1 is true
} else if (condition2) {
    statement(s)-2; // Executed if condition1 is false AND condition2 is true
} else if (condition3) {
    statement(s)-3; // Executed if condition1 and condition2 are false AND condition3 is
true
}
... // More else if conditions can be added
else {
    statement(s)-n; // Executed if all the above conditions are false (optional)
}
// Statements after the else if ladder
```

Explanation:

1. The `conditions` are evaluated sequentially from top to bottom.
2. As soon as a `condition` evaluates to **true**, the corresponding block of `statement(s)` is executed, and the rest of the ladder is skipped.
3. If none of the `conditions` evaluate to **true**, the `statement(s)` inside the final `else` block (if present) are executed.
4. The final `else` block is optional. If it's omitted and none of the conditions are true, then no block within the ladder is executed.

Example 6: Determining the grade based on marks

```
#include <stdio.h>

int main() {
    int marks;

    printf("Enter your marks (0-100): ");
    scanf("%d", &marks);
```

```
    if (marks >= 90 && marks <= 100) {
        printf("Grade: A\n");
    } else if (marks >= 80 && marks < 90) {
        printf("Grade: B\n");
    } else if (marks >= 70 && marks < 80) {
        printf("Grade: C\n");
    } else if (marks >= 60 && marks < 70) {
        printf("Grade: D\n");
    } else if (marks >= 0 && marks < 60) {
        printf("Grade: F\n");
    } else {
        printf("Invalid marks entered.\n");
    }

    return 0;
}
```

Output (if user enters 85):

```
Enter your marks (0-100): 85
Grade: B
```

Output (if user enters 45):

```
Enter your marks (0-100): 45
Grade: F
```

Output (if user enters 110):

```
Enter your marks (0-100): 110
Invalid marks entered.
```

Example 7: Checking the sign of a number using else if ladder

```
#include <stdio.h>

int main() {
    int num;

    printf("Enter an integer: ");
    scanf("%d", &num);

    if (num > 0) {
        printf("%d is a positive number.\n", num);
    } else if (num < 0) {
        printf("%d is a negative number.\n", num);
    } else {
        printf("The number is zero.\n");
    }

    return 0;
}
```

Output (if user enters -7):

```
Enter an integer: -7
-7 is a negative number.
```

Output (if user enters 0):

```
Enter an integer: 0
The number is zero.
```

2.3 The Switch Statement

The `switch` statement is another powerful decision-making statement in C that allows you to select one block of code to execute from multiple possible blocks based on the value of a single expression. It provides a more structured and often more readable alternative to a long `else if` ladder when you are comparing a variable against a set of constant values.

Syntax of the Switch Statement:

```
switch (expression) {
    case constant-expression1:
        statement(s)-1;
        break;
    case constant-expression2:
        statement(s)-2;
        break;
    case constant-expression3:
        statement(s)-3;
        break;
    // ... more case labels
    default:
        statement(s)-default;
        break; // Optional but recommended
}
// Statements after the switch statement
```

Explanation:

1. The `switch (expression)` part:
    - The `switch` keyword marks the beginning of the statement.
    - The `expression` inside the parentheses is evaluated. This expression must result in an integer type (char, int, enum) or an expression that can be implicitly converted to an integer type. Floating-point types and strings are **not allowed** in the `switch` expression.
2. The `case constant-expression:` labels:
    - The `case` keyword is followed by a `constant-expression`. This must be a constant value (literal or a constant variable declared with `const`).
    - The value of the `switch expression` is compared against the value of each `constant-expression`.
    - If a match is found (i.e., `expression == constant-expression`), the program execution jumps to the statements following that `case` label.
3. The `statement(s);` within each `case`:
    - These are the blocks of code that will be executed if the corresponding `case` matches the `switch expression`.
    - Multiple statements can be included in a `case` block without needing curly braces `{}` (unlike `if` statements).
4. The `break;` statement:
    - The `break` statement is crucial within each `case` block. When a `break` statement is encountered, it immediately exits the `switch` statement, and the program control flows to the statements after the `switch` block.
    - **If `break` is omitted, the execution will "fall through" to the next `case` label**, even if its `constant-expression` does not match the `switch expression`. This "fall-through" behavior can sometimes be useful for executing the same code for multiple cases, but it is more often a source of bugs if not intentional.
5. The `default:` label (optional):
    - The `default` label is optional. If present, the statements following the `default` label are executed if the value of the `switch expression` does not match any of the `constant-expression` values in the `case` labels.
    - If no `default` label is present and none of the `case` values match the `switch expression`, then no block of code within the `switch` statement is executed, and the program control flows to the statements after the `switch` block.

- It is generally considered good practice to include a `default` case to handle unexpected or out-of-range values. The `default` case is typically placed at the end of the `switch` statement, but its position doesn't affect its functionality. A `break` statement is also recommended in the `default` case, although it's not strictly necessary since it's usually the last part.

Flow of Execution in a Switch Statement:

1. The `switch` `expression` is evaluated once.
2. The result of the `expression` is compared sequentially with the `constant-expression` values of each `case` label.
3. If a match is found, the code block associated with that `case` is executed.
4. If a `break` statement is encountered, the `switch` statement is exited.
5. If no match is found and a `default` label exists, the code block associated with the `default` label is executed.
6. If no match is found and no `default` label exists, no code block within the `switch` statement is executed.
7. Execution continues with the statements following the `switch` statement.

Example 1: Displaying the day of the week based on a number

```
#include <stdio.h>

int main() {
    int day;

    printf("Enter a number (1-7) for the day of the week: ");
    scanf("%d", &day);

    switch (day) {
        case 1:
            printf("Sunday\n");
            break;
        case 2:
            printf("Monday\n");
            break;
        case 3:
            printf("Tuesday\n");
            break;
        case 4:
            printf("Wednesday\n");
            break;
        case 5:
            printf("Thursday\n");
            break;
        case 6:
            printf("Friday\n");
            break;
        case 7:
            printf("Saturday\n");
            break;
        default:
            printf("Invalid day number.\n");
            break;
    }

    printf("End of program.\n");

    return 0;
}
```

Output (if user enters 3):

```
Enter a number (1-7) for the day of the week: 3
Tuesday
End of program.
```

Output (if user enters 8):

```
Enter a number (1-7) for the day of the week: 8
Invalid day number.
End of program.
```

Example 2: Performing arithmetic operations based on an operator

```
#include <stdio.h>

int main() {
    char operator;
    int num1, num2;
    int result;

    printf("Enter two integers and an operator (+, -, *, /): ");
    scanf("%d %d %c", &num1, &num2, &operator);

    switch (operator) {
        case '+':
            result = num1 + num2;
            printf("%d + %d = %d\n", num1, num2, result);
            break;
        case '-':
            result = num1 - num2;
            printf("%d - %d = %d\n", num1, num2, result);
            break;
        case '*':
            result = num1 * num2;
            printf("%d * %d = %d\n", num1, num2, result);
            break;
        case '/':
            if (num2 != 0) {
                result = num1 / num2;
                printf("%d / %d = %d\n", num1, num2, result);
            } else {
                printf("Error: Division by zero!\n");
            }
            break;
        default:
            printf("Invalid operator.\n");
            break;
    }

    return 0;
}
```

Output (if user enters 10 5 +):

```
Enter two integers and an operator (+, -, *, /): 10 5 +
10 + 5 = 15
```

Output (if user enters 8 2 /):

```
Enter two integers and an operator (+, -, *, /): 8 2 /
8 / 2 = 4
```

Output (if user enters 6 0 /):

```
Enter two integers and an operator (+, -, *, /): 6 0 /
Error: Division by zero!
```

Output (if user enters 7 3 %):

```
Enter two integers and an operator (+, -, *, /): 7 3 %
Invalid operator.
```

Example 3: Demonstrating "fall-through" behavior

```
#include <stdio.h>

int main() {
    char grade = 'B';

    switch (grade) {
        case 'A':
            printf("Excellent!\n");
            break;
        case 'B':
            printf("Good job!\n");
            // No break statement here - intentional fall-through
        case 'C':
            printf("Keep it up!\n");
            break;
        case 'D':
            printf("Passed.\n");
            break;
        case 'F':
            printf("Failed.\n");
            break;
        default:
            printf("Invalid grade.\n");
    }

    return 0;
}
```

Output:

```
Good job!
Keep it up!
```

In this example, because there is no `break` statement after the `case 'B':` block, the execution falls through to the next `case 'C':` block, and its `printf` statement is also executed.

Advantages of the Switch Statement:

- **Readability:** For comparing a single variable against multiple constant values, `switch` can be more readable than a long `else if` ladder.
- **Efficiency (Potentially):** In some cases, compilers can optimize `switch` statements more effectively than a series of `if-else if` conditions, especially when there are many cases.

Disadvantages of the Switch Statement:

- **Limited Expression Type:** The `switch` expression must evaluate to an integer type (or something implicitly convertible to it). Floating-point numbers and strings cannot be directly used.
- **Constant Case Values:** The `case` labels must be constant expressions. You cannot use variables or ranges directly in `case` labels (without additional logic).
- **Need for `break`:** For most common use cases, you need to remember to include `break` statements at the end of each `case` block to prevent unintended fall-through.

When to Use Switch vs. If-Else If:

- Use `switch` when you need to compare a single variable against a set of specific constant values.
- Use `if-else if` when you need to evaluate more complex conditions, conditions involving ranges, or conditions involving different variables or non-integer types.

In conclusion, the `switch` statement is a valuable tool in C for implementing multi-way branching based on the value of an integer expression. Understanding its syntax, the role of `case` and `break`, and the concept of fall-through is essential for using it effectively in your C programs.

2.4 The Ternary Operator (?:)

The ternary operator, often called the conditional operator, is a concise way to express simple `if-else` conditions in a single line of code. It's a shorthand for making a decision based on a condition and returning one of two values.

Syntax of the Ternary Operator:

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

Explanation:

1. **`condition`:** This is an expression that is evaluated. It can be any expression that results in a boolean value (true or false), although in C, any non-zero value is considered true, and zero is considered false.
2. **`?` (Question Mark):** This symbol follows the `condition` and signifies the start of the ternary operator.
3. **`expression1`:** This expression is evaluated and returned if the `condition` is true (non-zero).
4. **`:` (Colon):** This symbol separates `expression1` from `expression2`.
5. **`expression2`:** This expression is evaluated and returned if the `condition` is false (zero).

How it Works:

The entire ternary operation evaluates to the result of either `expression1` or `expression2`, depending on the truthiness of the `condition`. It essentially says: "If the `condition` is true, then the result is `expression1`; otherwise, the result is `expression2`."

Equivalence to IF-ELSE:

The ternary operator can often replace a simple `if-else` statement of the following form:

```
if (condition) {
    variable = value1;
} else {
    variable = value2;
}
```

This can be rewritten using the ternary operator as:

```
variable = condition ? value1 : value2;
```

Important Considerations:

- **Readability:** While the ternary operator can make code more concise, it can also make complex conditions harder to read if overused or nested too deeply. For more intricate logic, a standard `if-else` statement might be clearer.
- **Return Value:** The ternary operator always returns a value. This value can then be assigned to a variable, used in another expression, or passed as an argument to a function.
- **Type Consistency:** Ideally, `expression1` and `expression2` should have compatible data types. If they have different types, the compiler will attempt to perform implicit type conversion, which might lead to unexpected results if you're not careful.

Examples:

Example 1: Finding the maximum of two numbers

```
#include <stdio.h>

int main() {
    int num1, num2, max;
```

```
    printf("Enter two integers: ");
    scanf("%d %d", &num1, &num2);

    max = (num1 > num2) ? num1 : num2;

    printf("The maximum of %d and %d is %d\n", num1, num2, max);

    return 0;
}
```

Output (if user enters 15 8):

```
Enter two integers: 15 8
The maximum of 15 and 8 is 15
```

Output (if user enters 4 12):

```
Enter two integers: 4 12
The maximum of 4 and 12 is 12
```

Example 2: Checking if a number is even or odd (returning a string)

```
#include <stdio.h>

int main() {
    int num;
    const char *parity; // Pointer to a constant character string

    printf("Enter an integer: ");
    scanf("%d", &num);

    parity = (num % 2 == 0) ? "Even" : "Odd";

    printf("%d is %s\n", num, parity);

    return 0;
}
```

Output (if user enters 7):

```
Enter an integer: 7
7 is Odd
```

Output (if user enters 20):

```
Enter an integer: 20
20 is Even
```

Example 3: Determining the sign of a number (returning 1, -1, or 0)

```
#include <stdio.h>

int main() {
    int num;
    int sign;

    printf("Enter an integer: ");
    scanf("%d", &num);

    sign = (num > 0) ? 1 : (num < 0) ? -1 : 0;

    printf("The sign of %d is %d (1 for positive, -1 for negative, 0 for zero)\n", num,
sign);

    return 0;
}
```

Output (if user enters -5):

```
Enter an integer: -5
The sign of -5 is -1 (1 for positive, -1 for negative, 0 for zero)
```

Output (if user enters 0):

```
Enter an integer: 0
The sign of 0 is 0 (1 for positive, -1 for negative, 0 for zero)
```

Output (if user enters 10):

```
Enter an integer: 10
The sign of 10 is 1 (1 for positive, -1 for negative, 0 for zero)
```

Nesting Ternary Operators:

As seen in Example 3, you can nest ternary operators. However, this can quickly become difficult to read and understand. It's generally advisable to use nesting sparingly and only for very simple nested conditions. For more complex nested logic, stick to `if-else if-else` statements for better clarity.

Example of Poor Readability with Deep Nesting:

```
// Avoid this kind of deep nesting if possible
int result = (a > b) ? (a > c ? a : c) : (b > c ? b : c); // Finding the maximum of
three numbers (less readable)
```

While the above code finds the maximum of three numbers using nested ternary operators, the version in Example 5 of the `if-else` section is generally considered more readable.

When to Use the Ternary Operator:

- For simple conditional assignments.
- In short expressions where choosing between two values based on a condition makes the code more concise and still readable.
- Sometimes within function arguments where a conditional value is needed.

When to Avoid the Ternary Operator (or use with caution):

- For complex conditions involving multiple logical operators.
- When the expressions to be evaluated are long or have side effects.
- For deeply nested conditions, as it can significantly reduce readability.

2.5 The GOTO Statement

The `goto` statement in C provides an unconditional jump to a labeled statement within the same function. It allows you to transfer the program's control flow directly to a specific point in the code.

Syntax of the GOTO Statement:

```
goto label_name;

// ... some code ...

label_name:
    statement(s);
```

Explanation:

1. **`goto label_name;`**: This statement causes the program's execution to immediately jump to the statement labeled with `label_name`.
2. **`label_name:`**: This is an identifier followed by a colon (`:`) that marks a specific location within the function. The `label_name` must follow the same naming rules as other identifiers in C.
3. The label and the `goto` statement that refers to it must be within the same function. You cannot jump out of one function into another using `goto`.

How it Works:

When the compiler encounters a `goto` statement, it searches for the corresponding label within the current function. Once the label is found, the program's execution resumes from the statement immediately following the label.

Examples:

Example 1: Simple Unconditional Jump

```
#include <stdio.h>

int main() {
    printf("Start of program.\n");
    goto end;
    printf("This line will be skipped.\n");
end:
    printf("End of program.\n");
    return 0;
}
```

Output:

```
Start of program.
End of program.
```

In this example, the `goto end;` statement causes the program to jump directly to the label `end:`, skipping the `printf("This line will be skipped.\n");` statement.

Example 2: Jumping Forward in a Loop (Less Common)

```
#include <stdio.h>

int main() {
    for (int i = 0; i < 5; i++) {
        if (i == 2) {
            goto skip_print;
```

```
        }
        printf("Value of i: %d\n", i);
    skip_print:; // Label for jumping to
    }
    printf("Loop finished.\n");
    return 0;
}
```

Output:

```
Value of i: 0
Value of i: 1
Loop finished.
```

Here, when `i` becomes 2, the `goto skip_print;` statement jumps to the `skip_print:` label, effectively skipping the `printf` statement for `i = 2`. The loop then continues. Note the empty statement `;` after the label, which is syntactically required if you want the label to precede another statement.

Example 3: Jumping Backward (Creating a Loop - Generally Discouraged)

```
#include <stdio.h>

int main() {
    int i = 0;
start:
    if (i < 5) {
        printf("Value of i: %d\n", i);
        i++;
        goto start;
    }
    printf("Loop finished (created with goto).\n");
    return 0;
}
```

Output:

```
Value of i: 0
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Loop finished (created with goto).
```

This example demonstrates how `goto` can be used to create a loop-like structure by jumping backward. However, this approach is generally considered bad programming practice as it makes the code harder to follow and maintain compared to using standard loop constructs like `for` and `while`.

The Controversy and Disadvantages of GOTO:

The `goto` statement is one of the most debated features in programming. It is often associated with "spaghetti code" – code with a complex and tangled control flow that is difficult to read, understand, and debug.

Here are the main disadvantages of using `goto` excessively:

- **Reduced Readability:** Uncontrolled jumps can make the program's flow very hard to follow logically. It breaks the structured top-to-bottom execution that programmers typically expect.
- **Increased Complexity:** Code with many `goto` statements can become very complex and difficult to reason about. This increases the likelihood of introducing bugs and makes debugging much harder.
- **Maintenance Difficulties:** Modifying or extending code that heavily uses `goto` can be challenging and error-prone because the impact of changes can be hard to predict due to the jumps in control flow.
- **Violation of Structured Programming Principles:** Structured programming emphasizes using well-defined control flow constructs like `if-else`, `for`, `while`, and functions to create modular and understandable code. `goto` disrupts this structure.

Legitimate (Rare) Uses of GOTO:

Despite its general discouragement, there are a few rare situations where `goto` might be considered acceptable or even beneficial:

- **Breaking out of deeply nested loops:** In situations with multiple levels of nested loops, using `break` only exits the innermost loop. `goto` can provide a cleaner way to jump out of all the nested loops at once upon a specific condition.

```
#include <stdio.h>

int main() {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            if (i * j > 10) {
                goto exit_loops;
            }
            printf("i = %d, j = %d\n", i, j);
        }
    }
exit_loops:
    printf("Exiting nested loops.\n");
    return 0;
}
```

- **Centralized error handling in a function:** In some functions, especially those involving resource allocation, `goto` can be used to jump to a common cleanup section when an error occurs at different points in the function. This can avoid code duplication for releasing resources.

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

int process_data() {
    int *data1 = NULL;
    int *data2 = NULL;
    int result = 0;

    data1 = (int *)malloc(10 * sizeof(int));
    if (data1 == NULL) goto error;

    data2 = (int *)malloc(20 * sizeof(int));
    if (data2 == NULL) goto error;

    // ... process data ...
    result = 1;

cleanup:
    if (data1 != NULL) free(data1);
    if (data2 != NULL) free(data2);
    return result;

error:
    printf("Memory allocation failed.\n");
    result = -1;
    goto cleanup;
}

int main() {
    int status = process_data();
    printf("Process status: %d\n", status);
    return 0;
}
```

Best Practices:

- **Avoid `goto` whenever possible.** Use structured control flow statements like `if-else`, `for`, `while`, and `switch` to create clear and maintainable code.
- If you find yourself tempted to use `goto`, think carefully if there is a more structured way to achieve the same result.
- If you must use `goto` (in those rare legitimate cases), use it sparingly and make the jumps as short and forward as possible to minimize confusion. Always document the reason for using `goto`.
- Avoid jumping backward as it can easily lead to infinite loops or very convoluted logic.

Unit 3.0 - Structured Loop Control Statements

3.1 Introduction

Loop control statements in C are fundamental constructs that allow you to execute a block of code repeatedly as long as a certain condition remains true. Loops are essential for automating repetitive tasks and processing collections of data efficiently. C provides three primary structured loop control statements:

- `while` statement
- `do...while` statement
- `for` statement

The WHILE Statement

The `while` statement in C is an **entry-controlled loop**. This means that the condition is evaluated *before* the loop body is executed. If the condition is initially false, the loop body will not be executed at all.

Syntax of the WHILE Statement:

```
while (condition) {
    // Body of the loop - one or more statements
}
// Statements following the while loop
```

Explanation:

1. **`while (condition)`:**
    - The `while` keyword marks the beginning of the loop.
    - The `condition` inside the parentheses `()` is an expression that is evaluated at the beginning of each iteration.
    - The `condition` can be any expression that evaluates to a Boolean value (true or false). In C, any non-zero value is considered true, and zero is considered false.
2. **`{ // Body of the loop }`:**
    - The block of code enclosed within the curly braces `{}` constitutes the body of the loop.
    - If the `condition` evaluates to true, all the statements inside the loop body are executed sequentially.
3. **Flow of Execution:**
    1. The `condition` is evaluated first.
    2. If the `condition` is false, the loop terminates immediately, and the program control moves to the first statement after the `while` loop.
    3. If the `condition` is true, the statements inside the loop body are executed.
    4. After all the statements in the body have been executed, the control returns to the beginning of the `while` statement, and the `condition` is evaluated again.
    5. This process repeats as long as the `condition` remains true.

Important Points:

- It is crucial to ensure that something within the loop body eventually makes the `condition` false; otherwise, the loop will run indefinitely (an **infinite loop**). Typically, one or more variables used in the `condition` are modified within the loop's body.
- If the loop body consists of only a single statement, the curly braces `{}` are optional, but it is good practice to always use them for clarity and maintainability.

Examples:

Example 1: Printing numbers from 1 to 5

```
#include <stdio.h>

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

    while (i <= 5) { // Condition
        printf("%d ", i);
        i++; // Update (increment i)
    }
    printf("\nLoop finished.\n");

    return 0;
}
```

Output:

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

Example 2: Taking input from the user until they enter 0 and calculating the sum

```
#include <stdio.h>

int main() {
    int number;
    int sum = 0;

    printf("Enter numbers (enter 0 to stop): ");
    scanf("%d", &number);

    while (number != 0) {
        sum += number;
        printf("Enter another number (enter 0 to stop): ");
        scanf("%d", &number);
    }

    printf("Sum of entered numbers: %d\n", sum);

    return 0;
}
```

Output (for a possible input sequence):

```
Enter numbers (enter 0 to stop): 10
Enter another number (enter 0 to stop): 5
Enter another number (enter 0 to stop): -2
Enter another number (enter 0 to stop): 0
Sum of entered numbers: 13
```

Example 3: Implementing a simple counter with a limit

```
#include <stdio.h>

int main() {
    int count = 0;
    int limit = 3;

    while (count < limit) {
        printf("Count is: %d\n", count);
        count++;
    }
    printf("Counter reached the limit.\n");
```

```
    return 0;
}
```

Output:

```
Count is: 0
Count is: 1
Count is: 2
Counter reached the limit.
```

The DO…WHILE Statement

The `do...while` statement in C is an **exit-controlled loop**. This means that the loop body is executed *at least once* before the condition is evaluated. The condition is checked *after* each iteration of the loop. If the condition is true, the loop continues; otherwise, it terminates.

Syntax of the DO…WHILE Statement:

```
do {
    // Body of the loop - one or more statements
} while (condition);
// Statements following the do...while loop
```

Explanation:

1. **`do { // Body of the loop }`**:
    - The `do` keyword marks the beginning of the loop.
    - The block of code enclosed within the curly braces `{}` is the body of the loop and is executed in the first iteration without checking the `condition`.
2. **`while (condition);`**:
    - The `while` keyword followed by the `condition` in parentheses `()` and a semicolon `;` marks the end of the `do...while` loop.
    - The `condition` is evaluated *after* each execution of the loop body.
    - If the `condition` is true (non-zero), the control goes back to the `do` statement, and the loop body is executed again.
    - If the `condition` is false (zero), the loop terminates, and the program control moves to the statement following the `while (condition);`.

Important Points:

- The `do...while` loop is useful when you need to execute a block of code at least once, regardless of the initial state of the condition. For example, when prompting a user for input and you want to ensure they are asked at least once.
- Remember to include the semicolon `;` after the `while (condition)` in a `do...while` loop, as it is part of the syntax.
- Similar to the `while` loop, you must ensure that the `condition` eventually becomes false to avoid an infinite loop.

Examples:

Example 1: Printing numbers from 1 to 5 using do...while

```
#include <stdio.h>

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

    do {
        printf("%d ", i);
        i++; // Update
```

```
    } while (i <= 5); // Condition
    printf("\nLoop finished.\n");

    return 0;
}
```

Output:

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

Example 2: Validating user input (ensuring a positive number is entered)

```
#include <stdio.h>

int main() {
    int number;

    do {
        printf("Enter a positive integer: ");
        scanf("%d", &number);
    } while (number <= 0); // Condition: continue if number is not positive

    printf("You entered: %d\n", number);

    return 0;
}
```

Output (for a possible input sequence):

```
Enter a positive integer: -5
Enter a positive integer: 0
Enter a positive integer: 12
You entered: 12
```

In this example, the user is prompted to enter a number at least once. The loop continues to ask for input until a positive number is entered.

Example 3: Simple menu-driven program

```
#include <stdio.h>

int main() {
    int choice;

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

        switch (choice) {
            case 1:
                printf("You selected Option A.\n");
                break;
            case 2:
                printf("You selected Option B.\n");
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3); // Continue until the user chooses to exit
```

```
    return 0;
}
```

This example shows how `do...while` can be used to present a menu to the user at least once and continue to display it until a specific exit condition is met.

Key Differences Between WHILE and DO…WHILE:

Feature	`while` statement	`do...while` statement
Condition Check	Before the loop body (entry-controlled)	After the loop body (exit-controlled)
Body Execution	May not execute at all if the initial condition is false	Executes at least once
Syntax	`while (condition) { ... }`	`do { ... } while (condition);`
Use Cases	When the loop body might not need to execute	When the loop body needs to execute at least once

Export to Sheets

Understanding the characteristics of `while` and `do...while` loops allows you to choose the appropriate loop structure for different programming tasks in C, leading to more efficient and logical code.

3.2 The FOR Statement

The `for` statement in C provides a concise way to structure a loop that iterates a specific number of times or until a certain condition is met. It encapsulates the initialization, condition checking, and update of a loop counter (or control variable) within its header. It is an **entry-controlled loop**, similar to the `while` statement.

Syntax of the FOR Statement:

```
for (initialization; condition; update) {
    // Body of the loop - one or more statements
}
// Statements following the for loop
```

Explanation:

1. **`for (...)`**: The `for` keyword marks the beginning of the loop. The parentheses contain three expressions separated by semicolons:
    - **`initialization`**: This expression is executed *only once* at the beginning of the loop. It is typically used to initialize loop counter variables. You can declare and initialize variables here.
    - **`condition`**: This expression is evaluated *before* each iteration of the loop (including the first). If the `condition` is true (non-zero), the loop body is executed. If it is false (zero), the loop terminates.
    - **`update`**: This expression is executed *at the end* of each iteration, after the loop body has been executed. It is typically used to increment or decrement the loop counter variable.
2. **`{ // Body of the loop }`**: The block of code enclosed within the curly braces `{}` constitutes the body of the loop and is executed in each iteration as long as the `condition` is true.
3. **Flow of Execution:**
    1. The `initialization` expression is executed.
    2. The `condition` is evaluated.
    3. If the `condition` is true, the loop body is executed.
    4. After the loop body is executed, the `update` expression is executed.

5. Steps 2-4 are repeated until the `condition` becomes false.
6. Once the `condition` is false, the loop terminates, and the program control moves to the first statement after the `for` loop.

Important Points:

- Any or all of the three expressions (`initialization`, `condition`, `update`) in the `for` loop header can be omitted, but the semicolons must remain.
  - If the `condition` is omitted, it is assumed to be always true, potentially leading to an infinite loop (which you would then need to control with `break` statements inside the loop).
  - If the `initialization` or `update` is omitted, you would typically handle these operations elsewhere in your code.
- The scope of variables declared in the `initialization` part of the `for` loop is usually limited to the loop itself (in more modern C standards like C99 and later).

Examples:

Example 1: Printing numbers from 1 to 5 using for loop

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) { // Initialization: i = 1; Condition: i <= 5; Update:
i++
        printf("%d ", i);
    }
    printf("\nLoop finished.\n");

    return 0;
}
```

Output:

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

Example 2: Calculating the sum of the first 10 natural numbers

```
#include <stdio.h>

int main() {
    int sum = 0;

    for (int i = 1; i <= 10; i++) {
        sum += i;
    }

    printf("Sum of first 10 natural numbers: %d\n", sum);

    return 0;
}
```

Output:

```
Sum of first 10 natural numbers: 55
```

Example 3: Iterating through an array

```
#include <stdio.h>

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    for (int i = 0; i < size; i++) {
        printf("Element at index %d: %d\n", i, numbers[i]);
    }

    return 0;
}
```

Output:

```
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50
```

Example 4: For loop with a step of 2

```
#include <stdio.h>

int main() {
    for (int i = 0; i <= 10; i += 2) {
        printf("%d ", i);
    }
    printf("\n");

    return 0;
}
```

Output:

```
0 2 4 6 8 10
```

Example 5: Decrementing loop counter

```
#include <stdio.h>

int main() {
    for (int i = 10; i >= 1; i--) {
        printf("%d ", i);
    }
    printf("\n");

    return 0;
}
```

Output:

```
10 9 8 7 6 5 4 3 2 1
```

The BREAK Statement

The `break` statement is used to immediately terminate the innermost enclosing loop (either `while`, `do...while`, or `for`) or the `switch` statement. When a `break` statement is encountered, the program's control flow jumps to the statement immediately following the terminated loop or `switch` block.

Syntax of the BREAK Statement:

```
break;
```

Usage within Loops:

The `break` statement is often used within a loop when a certain condition is met, and there is a need to exit the loop prematurely, before the loop's natural termination condition is reached.

Example 6: Using break to exit a loop based on a condition

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i > 5) {
            break; // Exit the loop when i becomes greater than 5
        }
        printf("%d ", i);
    }
    printf("\nLoop terminated early.\n");

    return 0;
}
```

Output:

```
1 2 3 4 5
Loop terminated early.
```

Example 7: Using break in a while loop

```
#include <stdio.h>

int main() {
    int i = 0;
    while (1) { // Infinite loop (will be terminated by break)
        printf("%d ", i);
        i++;
        if (i > 5) {
            break;
        }
    }
    printf("\nLoop terminated with break.\n");

    return 0;
}
```

Output:

```
0 1 2 3 4 5
Loop terminated with break.
```

The CONTINUE Statement

The `continue` statement is used to skip the rest of the current iteration of the innermost enclosing loop (`while`, `do...while`, or `for`) and proceed to the next iteration. For `while` and `do...while` loops, this means the control jumps to the condition evaluation. For `for` loops, the control jumps to the `update` expression.

Syntax of the CONTINUE Statement:

```
continue;
```

Usage within Loops:

The `continue` statement is useful when you want to skip processing for certain values within a loop but want the loop to continue with the next iteration.

Example 8: Using continue to skip even numbers

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            continue; // Skip even numbers
        }
        printf("%d ", i);
    }
    printf("\n");

    return 0;
}
```

Output:

```
1 3 5 7 9
```

Example 9: Using continue in a while loop

```
#include <stdio.h>

int main() {
    int i = 0;
    while (i < 5) {
        i++;
        if (i == 3) {
            continue; // Skip when i is 3
        }
        printf("Value of i: %d\n", i);
    }

    return 0;
}
```

Output:

```
Value of i: 1
Value of i: 2
Value of i: 4
Value of i: 5
```

Summary of BREAK and CONTINUE:

Statement	Purpose	Effect on Loop
`break`	Immediately terminates the innermost enclosing loop or `switch` statement.	Program control jumps to the statement immediately following the terminated loop or `switch` block.
`continue`	Skips the rest of the current iteration of the innermost enclosing loop.	Program control jumps to the condition evaluation (`while`, `do...while`) or the `update` expression (`for`).

Unit 4.0 - User-Defined Functions

4.1 Concept and Need of Functions

In C programming, a **function** is a self-contained block of code that performs a specific task. It is a fundamental building block of structured programming, allowing you to break down complex problems into smaller, more manageable, and reusable units.

Concept of Functions:

Think of a function as a mini-program within your main program. It has a name, may accept input values (called **arguments** or **parameters**), performs a set of operations, and may return a result. Once defined, a function can be **called** (or invoked) from different parts of your program as many times as needed, without having to rewrite the same code.

Key Aspects of a Function:

1. **Function Declaration (Prototype):** This declares the function's name, the types and number of arguments it expects, and the type of value it will return. It informs the compiler about the function's interface before it's actually defined or used.

   ```
   return_type function_name(type1 parameter1, type2 parameter2, ...);
   ```

2. **Function Definition:** This contains the actual code that the function executes. It includes the function header (similar to the declaration but without the semicolon) and the function body enclosed in curly braces `{}`.

   ```
   return_type function_name(type1 parameter1, type2 parameter2, ...) {
       // Statements that perform the function's task
       // ...
       return value; // Optional, depending on the return_type
   }
   ```

3. **Function Call (Invocation):** This is how you execute the code within a function from another part of your program. You use the function's name followed by parentheses `()` containing the actual values (arguments) you want to pass to the function.

   ```
   result = function_name(argument1, argument2, ...);
   ```

Need for Functions:

Functions are crucial for several reasons, contributing significantly to good programming practices and efficient software development:

1. **Modularity and Organization:** Functions allow you to divide a large and complex program into smaller, logical, and independent modules. Each function can be responsible for a specific part of the overall task. This makes the code more organized, easier to understand, and simplifies the development process.
2. **Code Reusability:** Once a function is defined to perform a specific task, it can be called from multiple locations within the same program or even from different programs (if organized into libraries). This eliminates the need to write the same code repeatedly, saving time and effort.
3. **Reduced Code Duplication:** Reusing functions naturally leads to less code duplication. This makes the codebase smaller, easier to maintain, and reduces the chances of inconsistencies or errors that can arise from having the same logic implemented in multiple places.
4. **Improved Readability and Understandability:** Breaking down a program into well-named functions makes the code more readable and easier to follow. Each function's name should clearly indicate its purpose, allowing programmers to understand the program's logic at a higher level without getting bogged down in the details of every single operation.
5. **Simplified Debugging and Testing:** When a program is organized into functions, it becomes easier to isolate and test individual units of code. If an error occurs, you can often pinpoint the problematic function more quickly. Testing each function independently (unit testing) can ensure their correctness and reduce the likelihood of bugs in the overall program.

6. **Abstraction:** Functions allow you to abstract away the underlying implementation details of a task. When you call a function, you only need to know what it does and what inputs it expects, not necessarily how it performs its operation. This simplifies the use of the function and allows for changes in the implementation without affecting the parts of the program that call it (as long as the function's interface remains the same).
7. **Collaboration in Large Projects:** In team-based software development, functions enable different team members to work on different parts of the program concurrently. Each member can be responsible for designing and implementing specific functions, making collaboration more efficient and manageable.

Examples Illustrating the Need for Functions:

Example 1: Calculating the area of a rectangle (without functions)

```
#include <stdio.h>

int main() {
    int length1 = 5, width1 = 10, area1;
    area1 = length1 * width1;
    printf("Area of rectangle 1: %d\n", area1);

    int length2 = 7, width2 = 3, area2;
    area2 = length2 * width2;
    printf("Area of rectangle 2: %d\n", area2);

    int length3 = 12, width3 = 6, area3;
    area3 = length3 * width3;
    printf("Area of rectangle 3: %d\n", area3);

    return 0;
}
```

In this example, the logic for calculating the area is repeated three times. If we needed to calculate the area of many rectangles, this would lead to a lot of redundant code.

Example 2: Calculating the area of a rectangle (using a function)

```
#include <stdio.h>

// Function declaration (prototype)
int calculate_area(int length, int width);

int main() {
    int length1 = 5, width1 = 10;
    int area1 = calculate_area(length1, width1);
    printf("Area of rectangle 1: %d\n", area1);

    int length2 = 7, width2 = 3;
    int area2 = calculate_area(length2, width2);
    printf("Area of rectangle 2: %d\n", area2);

    int length3 = 12, width3 = 6;
    int area3 = calculate_area(length3, width3);
    printf("Area of rectangle 3: %d\n", area3);

    return 0;
}

// Function definition
int calculate_area(int length, int width) {
    return length * width;
}
```

Here, the `calculate_area` function encapsulates the area calculation logic. It is called multiple times with different arguments, demonstrating code reusability and improved organization.

Example 3: Checking if a number is even (without functions)

```
#include <stdio.h>

int main() {
    int num1 = 10;
    if (num1 % 2 == 0) {
        printf("%d is even.\n", num1);
    } else {
        printf("%d is odd.\n", num1);
    }

    int num2 = 7;
    if (num2 % 2 == 0) {
        printf("%d is even.\n", num2);
    } else {
        printf("%d is odd.\n", num2);
    }

    return 0;
}
```

Example 4: Checking if a number is even (using a function)

```
#include <stdio.h>

// Function declaration
int is_even(int num);

int main() {
    int num1 = 10;
    if (is_even(num1)) {
        printf("%d is even.\n", num1);
    } else {
        printf("%d is odd.\n", num1);
    }

    int num2 = 7;
    if (is_even(num2)) {
        printf("%d is even.\n", num2);
    } else {
        printf("%d is odd.\n", num2);
    }

    return 0;
}

// Function definition
int is_even(int num) {
    return (num % 2 == 0); // Returns 1 (true) if even, 0 (false) if odd
}
```

This example shows how a function can encapsulate a logical test, making the `main` function cleaner and easier to understand.

4.2 Library Functions

C provides a rich set of pre-written functions organized into standard libraries. These **library functions** are designed to perform common and essential tasks, saving programmers from having to implement them from scratch. To use these functions, you typically need to include the corresponding header file in your C program using the `#include` preprocessor directive.

4.2.1 Math Functions (`math.h`)

The `math.h` header file declares various mathematical functions that perform operations like trigonometry, logarithms, exponentiation, and more. To use these functions, you must include `#include <math.h>` at the beginning of your program.

Commonly Used Math Functions:

- **`sqrt(x)`**: Returns the non-negative square root of `x`.

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

```
int main() {
     double num = 16.0;
     double result = sqrt(num);
     printf("Square root of %.2f is %.2f\n", num, result); // Output: Square root of
16.00 is 4.00
return 0;
}
```

- **`pow(x, y)`**: Returns `x` raised to the power of `y` (xy).

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

```
int main() {
    double base = 2.0;
    double exponent = 3.0;
    double result = pow(base, exponent);
    printf("%.2f raised to the power of %.2f is %.2f\n", base, exponent, result); //
Output: 2.00 raised to the power of 3.00 is 8.00
    return 0;
}
```

- **`sin(x)`, `cos(x)`, `tan(x)`**: Returns the sine, cosine, and tangent of `x` (where `x` is in radians).

```
#include <stdio.h>
#include <math.h>
#define PI 3.1415926535

int main() {
    double angle_degrees = 30.0;
    double angle_radians = angle_degrees * PI / 180.0;
    double sine_value = sin(angle_radians);
    printf("Sine of %.2f degrees is %.2f\n", angle_degrees, sine_value); // Output:
Sine of 30.00 degrees is 0.50
    return 0;
}
```

- **`log(x)`**: Returns the natural logarithm (base *e*) of `x`.

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

```
int main() {
    double num = 2.71828;
    double result = log(num);
    printf("Natural logarithm of %.5f is %.5f\n", num, result); // Output: Natural
logarithm of 2.71828 is 1.00000
    return 0;
}
```

- **`log10(x)`**: Returns the base-10 logarithm of `x$.

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

int main() {
    double num = 100.0;
    double result = log10(num);
    printf("Base-10 logarithm of %.2f is %.2f\n", num, result); // Output: Base-10
logarithm of 100.00 is 2.00
    return 0;
}
```

- **`fabs(x)`**: Returns the absolute value of a floating-point number `x`.

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

int main() {
    double num1 = -5.2;
    double abs_num1 = fabs(num1);
    printf("Absolute value of %.2f is %.2f\n", num1, abs_num1); // Output: Absolute
value of -5.20 is 5.20
    return 0;
}
```

- **`ceil(x)`**: Returns the smallest integer value greater than or equal to `x`.

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

int main() {
    double num = 4.3;
    double result = ceil(num);
    printf("Ceiling of %.2f is %.0f\n", num, result); // Output: Ceiling of 4.30 is
5
    return 0;
}
```

- **`floor(x)`**: Returns the largest integer value less than or equal to `x$.

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

int main() {
    double num = 4.7;
    double result = floor(num);
    printf("Floor of %.2f is %.0f\n", num, result); // Output: Floor of 4.70 is 4
    return 0;
}
```

4.2.2 String Handling Functions (`string.h`)

The `string.h` header file declares functions for manipulating C-style strings (arrays of characters terminated by a null character `\0`). You need to include `#include <string.h>` to use these functions.

Commonly Used String Handling Functions:

- **`strlen(s)`**: Returns the length of the string `s` (excluding the null terminator).

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

int main() {
    char str[] = "Hello";
    size_t length = strlen(str);
    printf("Length of '%s' is %zu\n", str, length); // Output: Length of 'Hello' is
5
    return 0;
}
```

- **`strcpy(dest, src)`**: Copies the string `src` (including the null terminator) to the string `dest`. You must ensure that `dest` has enough space to hold the contents of `src`.

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

int main() {
    char source[] = "World";
    char destination[20];
    strcpy(destination, source);
    printf("Source string: %s\n", source);       // Output: Source string: World
    printf("Destination string: %s\n", destination); // Output: Destination string:
World
    return 0;
}
```

- **`strncpy(dest, src, n)`**: Copies at most `n` characters from the string `src` to the string `dest`. If `src` has fewer than `n` characters, the remainder of `dest` will be padded with null characters. If `src` has `n` or more characters, `dest` will *not* be null-terminated.

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

int main() {
    char source[] = "Programming";
    char destination[10];
    strncpy(destination, source, 5);
    destination[5] = '\0'; // Manually null-terminate
    printf("Copied string: %s\n", destination); // Output: Copied string: Progr
    return 0;
}
```

- **`strcat(dest, src)`**: Appends the string `src` to the end of the string `dest`. You must ensure that `dest` has enough space to hold the combined string (including the null terminator).

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

int main() {
    char str1[20] = "Hello, ";
    char str2[] = "World!";
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1); // Output: Concatenated string:
Hello, World!
    return 0;
}
```

- **`strncat(dest, src, n)`**: Appends at most `n` characters from the string `src` to the end of the string `dest`. The resulting string in `dest` will always be null-terminated.

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

int main() {
    char str1[20] = "Hello";
    char str2[] = ", Universe!";
    strncat(str1, str2, 8);
    printf("Concatenated string: %s\n", str1); // Output: Concatenated string:
Hello, Univ
    return 0;
}
```

- **`strcmp(s1, s2)`**: Compares the string `s1` to the string `s2` lexicographically.
    - Returns 0 if `s1` and `s2` are equal.
    - Returns a negative value if `s1` comes before `s2` in lexicographical order.
    - Returns a positive value if `s1` comes after `s2` in lexicographical order.

<!-- end list -->

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

int main() {
    char str1[] = "apple";
    char str2[] = "banana";
    char str3[] = "apple";

    printf("strcmp(\"%s\", \"%s\") = %d\n", str1, str2, strcmp(str1, str2)); //
Output: strcmp("apple", "banana") = -1
    printf("strcmp(\"%s\", \"%s\") = %d\n", str1, str3, strcmp(str1, str3)); //
Output: strcmp("apple", "apple") = 0
    printf("strcmp(\"%s\", \"%s\") = %d\n", str2, str1, strcmp(str2, str1)); //
Output: strcmp("banana", "apple") = 1
    return 0;
}
```

- **`strncmp(s1, s2, n)`**: Compares at most the first `n` characters of the string `s1` to the string `s2`. The return value is similar to `strcmp`.

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

int main() {
    char str1[] = "compare1";
    char str2[] = "compare2";

    printf("strncmp(\"%s\", \"%s\", 7) = %d\n", str1, str2, strncmp(str1, str2, 7));
// Output: strncmp("compare1", "compare2", 7) = 0
    printf("strncmp(\"%s\", \"%s\", 8) = %d\n", str1, str2, strncmp(str1, str2, 8));
// Output: strncmp("compare1", "compare2", 8) = -1
    return 0;
}
```

4.2.3 Other Miscellaneous Functions

C provides various other useful library functions in different header files. Here are a few examples:

- **Standard Input/Output (`stdio.h`)**:
    - **`printf(format, ...)`**: Formatted output to the standard output (usually the console).
    - **`scanf(format, ...)`**: Formatted input from the standard input (usually the keyboard).
    - **`fopen(filename, mode)`**: Opens a file.
    - **`fclose(file)`**: Closes a file.
    - **`fprintf(file, format, ...)`**: Formatted output to a file.
    - **`fscanf(file, format, ...)`**: Formatted input from a file.

```c
#include <stdio.h>

int main() {
    int age;
    printf("Enter your age: ");
    scanf("%d", &age);
    printf("You are %d years old.\n", age);
    return 0;
}
```

- **Standard Library (`stdlib.h`)**:
  - **`atoi(str)`**: Converts a string `str` to an integer.
  - **`atof(str)`**: Converts a string `str` to a double.
  - **`atol(str)`**: Converts a string `str` to a long integer.
  - **`malloc(size)`**: Dynamically allocates memory.
  - **`calloc(num, size)`**: Dynamically allocates and initializes memory to zero.
  - **`free(ptr)`**: Deallocates dynamically allocated memory.
  - **`exit(status)`**: Terminates the program.
  - **`rand()`**: Generates a pseudo-random integer.
  - **`srand(seed)`**: Seeds the pseudo-random number generator.

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

int main() {
    char num_str[] = "123";
    int num = atoi(num_str);
    printf("String '%s' converted to integer: %d\n", num_str, num); // Output:
String '123' converted to integer: 123

    srand(time(NULL)); // Seed the random number generator with the current time
    int random_num = rand() % 100; // Generate a random number between 0 and 99
    printf("Random number: %d\n", random_num);
    return 0;
}
```

- **Character Handling (`ctype.h`)**:
  - **`isalpha(c)`**: Checks if `c` is an alphabetic character.
  - **`isdigit(c)`**: Checks if `c` is a digit.
  - **`isalnum(c)`**: Checks if `c` is an alphanumeric character.
  - **`islower(c)`**: Checks if `c` is a lowercase character.
  - **`isupper(c)`**: Checks if `c` is an uppercase character.
  - **`tolower(c)`**: Converts `c` to lowercase.
  - **`toupper(c)`**: Converts `c` to uppercase.

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

int main() {
    char ch = 'A';
    if (isupper(ch)) {
        printf("'%c' is an uppercase letter.\n", ch); // Output: 'A' is an uppercase
letter.
        printf("Lowercase of '%c' is '%c'.\n", ch, tolower(ch)); // Output:
Lowercase of 'A' is 'a'.
    }
    return 0;
}
```

Benefits of Using Library Functions:

- **Efficiency:** Library functions are often highly optimized for performance.
- **Reliability:** They are usually well-tested and less prone to errors than user-written code for the same tasks.
- **Portability:** Standard library functions are generally available across different C compilers and operating systems, making your code more portable.
- **Reduced Development Time:** They save you the effort of writing common functionalities from scratch, allowing you to focus on the specific logic of your application.

To effectively use library functions, it's essential to:

1. **Identify the appropriate header file** for the functions you need.
2. **Include the header file** using `#include <header_file.h>`.
3. **Understand the function's syntax**, including the types and number of arguments it expects and the type of value it returns.
4. **Refer to documentation** (e.g., man pages, online C documentation) for detailed information about each function's behavior and potential error conditions.

By leveraging the power of C's standard library, you can write more efficient, reliable, and maintainable programs.

Unit 5.0 - Single Dimensional Array in 'C'

5.1 Declaring and Initializing One-Dimensional Arrays

An **array** in C is a collection of elements of the same data type stored in contiguous memory locations. A **one-dimensional array** (also known as a linear array) is the simplest form of an array, representing a sequence of elements arranged in a single row or column. It's like a list of variables of the same type, accessed using a common name and an index (or subscript).

Declaring a One-Dimensional Array:

To declare a one-dimensional array in C, you use the following syntax:

```
data_type array_name[size];
```

Explanation:

- **`data_type`**: Specifies the data type of the elements that will be stored in the array (e.g., `int`, `float`, `char`, `double`). All elements in an array must be of the same data type.
- **`array_name`**: A valid identifier that you will use to refer to the array. Follow the standard naming conventions for variables.
- **`[size]`**: Specifies the number of elements the array can hold. The `size` must be a constant integer expression (a positive integer literal or a `const int` variable initialized with a positive value at declaration). The size is fixed at the time of declaration and cannot be changed during the program's execution for a statically allocated array.

Examples of Array Declarations:

```
int numbers[5];         // Declares an integer array named 'numbers' that can hold 5
                           integer elements.
float prices[10];       // Declares a floating-point array named 'prices' that can hold 10
                           float elements.
char letters[26];       // Declares a character array named 'letters' that can hold 26
                           character elements.
double measurements[100]; // Declares a double-precision floating-point array named
                           'measurements' that can hold 100 double elements.

const int ARRAY_SIZE = 7;
int scores[ARRAY_SIZE]; // Declares an integer array named 'scores' with a size defined by
                            a constant.
```

Accessing Array Elements:

Elements in a one-dimensional array are accessed using their index (or subscript), which starts from 0 for the first element and goes up to `size - 1` for the last element. The index is enclosed in square brackets `[]` after the array name.

```
array_name[index];
```

Examples of Accessing Array Elements:

```
int numbers[5];

numbers[0] = 10; // Assigns the value 10 to the first element (index 0).
numbers[1] = 25; // Assigns the value 25 to the second element (index 1).
numbers[4] = 100; // Assigns the value 100 to the fifth element (index 4).

printf("The first element is: %d\n", numbers[0]); // Output: The first element is: 10
printf("The third element is: %d\n", numbers[2]); // Output: The third element is:
(garbage value, as it's not initialized yet)
```

Important Note: Accessing an array element using an index that is out of bounds (less than 0 or greater than or equal to `size`) leads to **undefined behavior**. This can result in crashes, incorrect results, or security vulnerabilities, and the compiler usually doesn't provide any warnings or errors at compile time.

Initializing a One-Dimensional Array:

You can initialize the elements of a one-dimensional array at the time of its declaration using an initializer list enclosed in curly braces `{}`. The values in the list are assigned to the array elements in the order they appear.

Syntax for Initialization:

```
data_type array_name[size] = {value1, value2, ..., valueN};
```

Examples of Array Initialization:

```
int grades[5] = {90, 85, 78, 95, 80}; // Initializes an integer array with 5 values.
float ratios[3] = {1.5, 2.7, 0.9};   // Initializes a float array with 3 values.
char vowels[5] = {'a', 'e', 'i', 'o', 'u'}; // Initializes a character array with 5
characters.
```

Partial Initialization:

If the number of initializers in the list is less than the declared size of the array, the remaining elements are automatically initialized to 0 (for numeric types) or `\0` (for character types).

```
int scores[10] = {55, 60, 65}; // The first three elements are initialized, the remaining
seven are initialized to 0.
char message[20] = {'H', 'e', 'l', 'l', 'o'}; // The first five elements are initialized,
the remaining fifteen are initialized to '\0'.
```

Initialization without Specifying Size:

If you initialize an array at the time of declaration, you can omit the size. The compiler will automatically determine the size of the array based on the number of initializers in the list.

```
int values[] = {1, 2, 3, 4, 5}; // The array 'values' will have a size of 5.
char name[] = "John";          // The array 'name' will have a size of 5 (including the
null terminator '\0').
```

Initializing Character Arrays (Strings):

Character arrays can be initialized using a string literal enclosed in double quotes `""`. In this case, the compiler automatically adds a null terminator (`\0`) at the end of the string to mark the end of the character sequence.

```
char greeting[] = "Hello"; // Equivalent to char greeting[] = {'H', 'e', 'l', 'l', 'o',
'\0'};
printf("%s\n", greeting); // Output: Hello
```

Iterating Through Arrays:

Loops, especially `for` loops, are commonly used to process the elements of an array sequentially.

Example of Iterating and Printing Array Elements:

```
#include <stdio.h>

int main() {
    int numbers[5] = {10, 20, 30, 40, 50};
    int size = sizeof(numbers) / sizeof(numbers[0]); // Calculate the size of the array

    printf("Elements of the array are:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("numbers[%d] = %d\n", i, numbers[i]);
    }

    return 0;
}
```

Output:

```
Elements of the array are:
numbers[0] = 10
numbers[1] = 20
numbers[2] = 30
numbers[3] = 40
numbers[4] = 50
```

Example of Reading Values into an Array from User Input:

```
#include <stdio.h>

int main() {
    int grades[3];
    printf("Enter the grades of 3 students:\n");
    for (int i = 0; i < 3; i++) {
        printf("Grade for student %d: ", i + 1);
        scanf("%d", &grades[i]);
    }

    printf("\nEntered grades are:\n");
    for (int i = 0; i < 3; i++) {
        printf("Student %d: %d\n", i + 1, grades[i]);
    }

    return 0;
}
```

5.2 Array Operations

Once a one-dimensional array is declared and initialized, you can perform various operations on its elements. This section will cover common array operations such as insertion, searching, deletion, and basic string operations (as strings are essentially character arrays).

5.2.1 Insertion

Insertion in an array involves adding a new element at a specific position. However, due to the fixed size nature of statically declared arrays in C, true dynamic insertion (increasing the array's size) is not directly possible. Instead, we typically perform insertion within the existing allocated space, which might involve shifting existing elements to make room for the new one.

Steps for Insertion (within existing size):

1. **Check if there is space:** Ensure that the array is not already full if you are trying to insert. If it is, you cannot insert without allocating a new, larger array and copying the elements.
2. **Validate the position:** The insertion position should be within the valid bounds of the array (from 0 to the current number of elements). Inserting at the end is also a common case.
3. **Shift elements:** If the insertion position is not at the end, shift all the elements from the insertion position to the end of the array one position to the right to create space for the new element.
4. **Insert the element:** Place the new element at the desired position.
5. **Update the number of elements:** If you are tracking the number of elements in the array explicitly, increment this count.

Example of Insertion in an Array:

```
#include <stdio.h>

int main() {
    int arr[10] = {1, 2, 4, 5, 6}; // Array with some initial elements (size 10)
    int size = 5;                  // Current number of elements
    int element_to_insert = 3;
    int position = 2;              // Insert at index 2 (3rd position)

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

    if (size >= 10) {
        printf("Array is full. Cannot insert.\n");
    } else if (position < 0 || position > size) {
        printf("Invalid insertion position.\n");
    } else {
        // Shift elements to the right
        for (int i = size - 1; i >= position; i--) {
            arr[i + 1] = arr[i];
        }
        // Insert the new element
        arr[position] = element_to_insert;
        size++; // Increment the size

        printf("Array after insertion:\n");
        for (int i = 0; i < size; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
Original array:
1 2 4 5 6
Array after insertion:
1 2 3 4 5 6
```

5.2.2 Searching

Searching in an array involves finding the index of a specific element if it exists in the array. Two common searching techniques for one-dimensional arrays are linear search and binary search. Binary search is more efficient but requires the array to be sorted.

1. Linear Search:

- Iterate through each element of the array one by one.
- Compare each element with the target element.
- If a match is found, return the index of the element.
- If the end of the array is reached without finding the element, return a value indicating that the element is not present (e.g., -1).

Example of Linear Search:

```
#include <stdio.h>

int linear_search(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Element found at index i
        }
    }
    return -1; // Element not found
}

int main() {
    int arr[] = {5, 12, 8, 2, 9, 1};
    int size = sizeof(arr) / sizeof(arr[0]);
    int element_to_find = 8;
    int index = linear_search(arr, size, element_to_find);

    if (index != -1) {
        printf("%d found at index %d\n", element_to_find, index); // Output: 8 found at
index 2
    } else {
        printf("%d not found in the array\n", element_to_find);
    }

    element_to_find = 15;
    index = linear_search(arr, size, element_to_find);
    if (index != -1) {
        printf("%d found at index %d\n", element_to_find, index);
    } else {
        printf("%d not found in the array\n", element_to_find); // Output: 15 not found in
the array
    }

    return 0;
}
```

2. Binary Search (for sorted arrays):

- Requires the array to be sorted in ascending or descending order.
- Compare the target element with the middle element of the array.
- If the target is equal to the middle element, the search is successful.
- If the target is less than the middle element, search in the left half of the array.
- If the target is greater than the middle element, search in the right half of the array.
- Repeat the process until the target is found or the search space is empty.

Example of Binary Search (assuming the array is sorted):

```
#include <stdio.h>

int binary_search(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2; // Calculate middle index (prevents overflow)

        if (arr[mid] == target) {
            return mid; // Element found at index mid
        } else if (arr[mid] < target) {
            low = mid + 1; // Search in the right half
        } else {
            high = mid - 1; // Search in the left half
        }
    }
    return -1; // Element not found
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
```

```
    int size = sizeof(arr) / sizeof(arr[0]);
    int element_to_find = 23;
    int index = binary_search(arr, 0, size - 1, element_to_find);

    if (index != -1) {
        printf("%d found at index %d\n", element_to_find, index); // Output: 23 found at
index 5
    } else {
        printf("%d not found in the array\n", element_to_find);
    }

    element_to_find = 95;
    index = binary_search(arr, 0, size - 1, element_to_find);
    if (index != -1) {
        printf("%d found at index %d\n", element_to_find, index);
    } else {
        printf("%d not found in the array\n", element_to_find); // Output: 95 not found in
the array
    }

    return 0;
}
```

5.2.3 Deletion

Deletion in an array involves removing an element at a specific position. Similar to insertion, with statically allocated arrays, we don't actually free up the memory. Instead, we shift the subsequent elements to overwrite the element to be deleted, effectively removing it from the logical view of the array and typically decrementing the count of valid elements.

Steps for Deletion:

1. **Validate the position:** The deletion position should be within the valid bounds of the current elements in the array (from 0 to `size - 1`).
2. **Shift elements:** Shift all the elements from the position after the deletion position one position to the left to fill the gap created by the deleted element.
3. **Update the number of elements:** Decrement the count of valid elements in the array.

Example of Deletion in an Array:

```
#include <stdio.h>

int main() {
    int arr[10] = {1, 2, 3, 4, 5, 6}; // Array with initial elements (size 10)
    int size = 6;                     // Current number of elements
    int position = 2;                 // Delete element at index 2 (value 3)

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

    if (position < 0 || position >= size) {
        printf("Invalid deletion position.\n");
    } else {
        // Shift elements to the left
        for (int i = position; i < size - 1; i++) {
            arr[i] = arr[i + 1];
        }
        size--; // Decrement the size

        printf("Array after deletion:\n");
        for (int i = 0; i < size; i++) {
            printf("%d ", arr[i]);
```

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

    return 0;
}
```

Output:

```
Original array:
1 2 3 4 5 6
Array after deletion:
1 2 4 5 6
```

5.2.4 String Operations

In C, strings are treated as arrays of characters terminated by a null character (`\0`). Therefore, many string operations can be implemented by directly manipulating the elements of the character array. However, the `string.h` library provides optimized and convenient functions for common string operations.

Basic String Operations (using array manipulation):

- **Accessing characters:** You can access individual characters of a string using their index, just like any other array.
- **Modifying characters:** You can change individual characters of a string by assigning a new value to a specific index (ensure you don't overwrite the null terminator unless intended).
- **Finding the length:** You can iterate through the character array until you encounter the null terminator to find the length of the string.

Example of Basic String Operations:

```
#include <stdio.h>

int main() {
    char str[] = "Hello";

    printf("String: %s\n", str);        // Output: String: Hello
    printf("First character: %c\n", str[0]); // Output: First character: H
    printf("Third character: %c\n", str[2]);  // Output: Third character: l

    str[1] = 'a'; // Modify the second character
    printf("Modified string: %s\n", str); // Output: Modified string: Hallo

    int length = 0;
    while (str[length] != '\0') {
        length++;
    }
    printf("Length of the string: %d\n", length); // Output: Length of the string: 5

    return 0;
}
```

5.2.5 Concatenation of Two Strings

Concatenation means joining two strings together to form a single string. You can achieve this by manually copying the characters of the second string to the end of the first string (ensuring enough space is allocated for the result) or by using the `strcat()` function from `string.h`.

1. Manual Concatenation:

- Ensure the destination character array has enough space to hold both strings plus the null terminator.
- Iterate through the first string to find the null terminator (end of the string).
- Start copying characters from the second string to the position after the null terminator of the first string until the null terminator of the second string is reached.
- Manually add a null terminator at the end of the concatenated string.

Example of Manual String Concatenation:

```
#include <stdio.h>

int main() {
    char str1[20] = "Hello, ";
    char str2[] = "World!";
    int i = 0, j = 0;

    // Find the end of str1
    while (str1[i] != '\0') {
        i++;
    }

    // Copy characters from str2 to the end of str1
    while (str2[j] != '\0') {
        str1[i] = str2[j];
        i++;
        j++;
    }

    str1[i] = '\0'; // Add the null terminator to the concatenated string

    printf("Concatenated string: %s\n", str1); // Output: Concatenated string: Hello,
World!

    return 0;
}
```

2. Using `strcat()` function:

- The `strcat()` function from `string.h` provides a simpler way to concatenate strings.
- You need to include `<string.h>` in your program.
- Ensure that the destination string (`dest`) has enough allocated space to hold the concatenated string.

Example of String Concatenation using `strcat()`:

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

int main() {
    char str1[20] = "Hello, ";
    char str2[] = "World!";

    strcat(str1, str2);

    printf("Concatenated string: %s\n", str1); // Output: Concatenated string: Hello,
World!

    return 0;
}
```

Understanding these basic array operations is fundamental for effectively working with collections of data in C. For string manipulation, while manual operations are possible, the functions in `string.h` are generally preferred for their efficiency and ease of use.