

## **Diploma Engineering - Programming in C: Class Notes**

### **UNIT I: Basics of C Programming**

#### **History of C Language**

The C programming language has a rich history, evolving from earlier languages and becoming one of the most influential programming languages ever created.

- Early Beginnings (ALGOL, BCPL, B):
  - The roots of C can be traced back to ALGOL (Algorithmic Language), developed in the late 1950s, which introduced the concept of structured programming.
  - BCPL (Basic Combined Programming Language), developed by Martin Richards in 1967, was a typeless language intended for writing compilers and operating systems.
  - B language, developed by Ken Thompson at Bell Labs in 1969, was a simplified version of BCPL. It was used to create the first version of the UNIX operating system.
- Birth of C (1972):
  - Dennis Ritchie, also at Bell Labs, began developing C in 1972. He aimed to add data types to B and make it more powerful for system programming.
  - The primary motivation was to rewrite the UNIX operating system, which was initially written in assembly language. Writing an OS in a high-level language was revolutionary at the time.
- Evolution and Standardization:
  - K&R C (1978): Brian Kernighan and Dennis Ritchie published "The C Programming Language," which became the de-facto standard for C for many years. This version is often referred to as "K&R C."
- Legacy and Influence:
  - C became immensely popular for system programming (operating systems, embedded systems, compilers, databases) due to its efficiency and close-to-hardware access.
  - It profoundly influenced the development of many other modern programming languages, most notably C++, Java, JavaScript, C#, Python, and Go, which borrow heavily from C's syntax and concepts.

#### **Features of C Language**

C is a powerful and versatile language known for its efficiency, low-level access, and portability. Here are its key features:

### 1. Mid-level Language:

- C is often called a "middle-level" programming language.
- It combines the features of both high-level languages (like structured programming, readability) and low-level languages (like direct memory manipulation, bitwise operations).
- This makes it suitable for both system programming (close to hardware) and application programming.

### 2. Structured Programming Language:

- C supports structured programming concepts, allowing programs to be broken down into functions, loops, and conditional statements.
- This improves code readability, maintainability, and reusability, reducing the need for goto statements.

### 3. Rich Set of Library Functions:

- C comes with a rich collection of built-in functions (standard library) that perform common tasks such as input/output, string manipulation, mathematical operations, memory management, etc.
- These functions are organized into header files (e.g., `stdio.h`, `math.h`, `string.h`).

### 4. Portability:

- C programs are highly portable. Code written on one machine (with a C compiler) can often be compiled and run on different machines with minimal or no changes.
- This "write once, compile anywhere" philosophy (though not as absolute as Java) was a significant advantage.

### 5. Memory Management (Pointers):

- C provides direct access to memory through the use of pointers.
- This allows programmers to efficiently manage memory (allocate and deallocate dynamically) and interact directly with hardware, which is crucial for system-level programming.

### 6. Fast Execution Speed:

- C compilers generate highly optimized machine code.
- Its low-level capabilities and direct memory access contribute to its high execution speed, making it ideal for performance-critical applications.

### 7. Extensibility:

- C is an extensible language, meaning new features and user-defined functions can be easily added and integrated.
- You can create your own libraries and reuse them across different projects.

## 8. Case-Sensitive:

- C is a case-sensitive language.<sup>25</sup> `int` is different from `Int`, and `variable` is different from `Variable`.

## 9. Recursion:

- C supports recursion, where a function can call itself.<sup>26</sup> This is useful for solving problems that can be broken down into smaller, similar sub-problems (e.g., factorial, Fibonacci series).

## 10. Modularity:

- Programs can be divided into smaller, independent modules (functions).<sup>27</sup> These modules can be developed and tested separately and then combined to form a complete program.<sup>28</sup>

## 11. Absence of Garbage Collection:

- Unlike some modern languages (e.g., Java, Python), C does not have automatic garbage collection.<sup>29</sup> Programmers are responsible for manually allocating and deallocating memory, which gives fine-grained control but also places a burden on the developer to prevent memory leaks.<sup>30</sup>
- 

## 1.1 Steps in Development of a Program, Flowcharts, and Algorithm Development

### Steps in Development of a Program

Developing a computer program typically involves a series of structured steps. Following these steps helps ensure a well-designed, functional, and maintainable program.

#### 1. Problem Definition/Analysis:

- Clearly understand what the program needs to achieve.
- Identify the inputs, outputs, and processing requirements.
- What data will the program need? What results should it produce?

#### 2. Algorithm Development:

- An *algorithm* is a step-by-step procedure for solving a problem.
- It's a logical sequence of instructions that, when followed, will lead to the desired output.
- Algorithms can be expressed using natural language, pseudocode, or flowcharts.

#### 3. Flowcharting (Optional but Recommended):

- A *flowchart* is a graphical representation of an algorithm.

- It uses standard symbols to depict the flow of control and operations.
- Helps visualize the program logic and identify potential errors before coding.

#### 4. **Coding (Implementation):**

- Translate the algorithm (and/or flowchart) into a specific programming language (in this case, C).
- Write the source code using the syntax and rules of the chosen language.

#### 5. **Compilation/Interpretation:**

- For compiled languages like C, the source code is translated into machine-readable code (executable file) by a *compiler*.
- The compiler checks for syntax errors.

#### 6. **Debugging:**

- The process of finding and fixing errors (bugs) in the program.
- Errors can be syntax errors (caught by the compiler), logical errors (program doesn't do what it's supposed to), or runtime errors (errors that occur during execution).

#### 7. **Testing:**

- Running the program with various inputs to ensure it produces the correct outputs for all expected scenarios, including edge cases.
- Verify that the program meets all the initial requirements.

#### 8. **Documentation:**

- Writing comments within the code to explain its purpose and logic.
- Creating external documentation like user manuals, design specifications, and technical guides. This helps others understand and maintain the code.


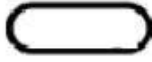





#### 9. **Maintenance:**

- Ongoing process of updating and improving the program after deployment.
- Includes fixing new bugs, adding new features, and adapting to changes in the environment.

### **Flowcharts**

A flowchart is a visual representation of the sequence of steps and decisions needed to perform a process. It uses standard symbols, each representing a specific type of operation or step in the process.

### Common Flowchart Symbols:

Symbol	Symbol Name	Description
	Flow Lines	Used to connect symbols
	Terminal	Used to start, pause or halt in the program logic
	Input/output	Represents the information entering or leaving the system
	Processing	Represents arithmetic and logical instructions
	Decision	Represents a decision to be made
	Connector	Used to Join different flow lines
	Sub function	used to call function

### Example Flowchart: To calculate the sum of two numbers.

A[Start] --> B(Input num1, num2)

B --> C{Sum = num1 + num2}

C --> D[Print Sum]

D --> E[End]

### Algorithm Development

An algorithm is a finite set of unambiguous instructions that, when performed, accomplishes a specific task. It's language-independent and focuses on the logic.

### Characteristics of a Good Algorithm:

- **Finiteness:** It must terminate after a finite number of steps.
- **Definiteness:** Each step must be precisely and unambiguously defined.
- **Input:** It must have zero or more well-defined inputs.
- **Output:** It must have one or more well-defined outputs.
- **Effectiveness:** Each step must be sufficiently basic that it can be carried out, at least in principle, by a person using pencil and paper.

### Example Algorithm: To calculate the sum of two numbers.

1. **START**

2. Declare two variables, num1 and num2, to store the numbers.
3. Declare a variable sum to store the result.
4. Read the value of num1 from the user.
5. Read the value of num2 from the user.
6. Calculate  $\text{sum} = \text{num1} + \text{num2}$ .
7. Display the value of sum.
8. **END**

## **1.2 Structure and Syntax of C Programme, Basic Preprocessor Directives and Library Functions, Program Debugging, I/O Statements, Constants, Variables and Data Types**

### **Structure and Syntax of a C Program**

Every C program follows a basic structure. Understanding this structure is crucial for writing correct C code.

```
#include <stdio.h> // Preprocessor Directive

// Global variable declaration (optional)
int globalVar = 10;

// Function declaration (optional)
void myFunction();

int main() { // main function: Entry point of the program

    // Local variable declaration
    int a = 5;
    char myChar = 'A';

    // Executable statements
    printf("Hello, World!\n"); // Output statement
    myFunction(); // Function call
    return 0; // Indicates successful execution
}
```

```
// Function definition

void myFunction() {

    printf("This is inside myFunction.\n");

}
```

### Explanation of Elements:

#### 1. Preprocessor Directives (#include, #define, etc.):

- These are instructions to the C preprocessor, which is a program that processes the source code *before* it is passed to the compiler.
- They typically start with a # symbol.
- #include <stdio.h>: This directive tells the preprocessor to include the contents of the stdio.h (Standard Input/Output) header file into the program. This file contains declarations for standard input/output functions like printf() and scanf().

#### 2. Global Variable Declaration (Optional):

- Variables declared outside any function are called global variables.
- They can be accessed from any function in the program.

#### 3. Function Declaration (Prototype - Optional for main and if defined before use):

- A function declaration tells the compiler about a function's name, return type, and parameters *before* it's actually defined. This is necessary if a function is called before its definition in the code.

#### 4. main() Function:

- Every C program **must** have a main() function.
- It is the entry point of the program. Execution always begins from main().
- int main(): Indicates that the main function returns an integer value.
- The code within the curly braces {} of main() is executed sequentially.

#### 5. Local Variable Declaration:

- Variables declared inside a function (like a and myChar in main) are called local variables.
- They are only accessible within the function in which they are declared.

#### 6. Executable Statements:

- These are the instructions that perform operations.
- Each statement in C typically ends with a semicolon (;).

## 7. **return 0;;**

- In the main function, return 0; indicates that the program executed successfully. A non-zero value typically indicates an error.

## 8. **Comments:**

- Used to explain the code. They are ignored by the compiler.
- Single-line comments: `// This is a single-line comment.`
- Multi-line comments: `/* This is a multi-line comment. It can span across multiple lines. */`

## **Basic Preprocessor Directives**

Besides `#include`, another common preprocessor directive is `#define`.

- **#define:** Used to define macros. Macros are essentially text substitutions that happen before compilation.

C

```
#define PI 3.14159 // Defines a constant PI
```

```
#define MAX(a, b) ((a) > (b) ? (a) : (b)) // Defines a macro for finding the maximum of two numbers
```

```
int main() {  
    float circumference = 2 * PI * 5;  
    int result = MAX(10, 20);  
    printf("Circumference: %f\n", circumference);  
    printf("Max: %d\n", result);  
    return 0;  
}
```

## **Library Functions**

C provides a rich set of built-in functions, known as *library functions*, grouped into various header files. These functions perform common tasks, saving you from writing them from scratch.

- **stdio.h (Standard Input/Output):**
  - `printf()`: For formatted output to the console.
  - `scanf()`: For formatted input from the console.
  - `getchar()`, `putchar()`, `gets()`, `puts()`: For character and string I/O.

- **math.h (Mathematical Functions):**
  - sqrt(): Calculates square root.
  - pow(): Calculates power.
  - sin(), cos(), tan(): Trigonometric functions.
- **stdlib.h (Standard Library):**
  - malloc(), free(): For dynamic memory allocation.
  - exit(): To terminate the program.
  - rand(), srand(): For random number generation.
- **string.h (String Manipulation):**
  - strlen(): Calculates string length.
  - strcpy(): Copies a string.
  - strcat(): Concatenates strings.
  - strcmp(): Compares strings.

## Program Debugging

Debugging is the process of identifying and fixing errors (bugs) in a program.

### Types of Errors:

1. **Syntax Errors:**
  - Violations of the grammar or rules of the C language.
  - Caught by the compiler during the compilation phase.
  - Examples: Missing semicolon, misspelled keywords, unmatched parentheses.
  - The compiler provides error messages that help pinpoint the location and type of error.
2. **Logical Errors:**
  - The program compiles and runs without errors, but produces incorrect or unexpected results.
  - The logic of the program is flawed.
  - Examples: Incorrect formula, wrong condition in an if statement, infinite loop.
  - Debugging logical errors often involves tracing the program's execution, printing intermediate values, or using a debugger.

### 3. Runtime Errors:

- Errors that occur during the execution of the program.
- Often lead to program crashes or abnormal termination.
- Examples: Division by zero, accessing an array out of bounds, memory leaks, stack overflow.
- These errors might not be caught by the compiler and only manifest during execution.

### Debugging Techniques:

- **Read Compiler Error Messages:** They are your first clue for syntax errors.
- **printf() Debugging:** Insert printf() statements at various points in your code to print the values of variables and trace the program's flow. This is a simple but effective technique.
- **Using a Debugger:** Integrated Development Environments (IDEs) like Code::Blocks, Visual Studio Code (with C/C++ extensions), or specialized debuggers (like GDB) allow you to:
  - Set **breakpoints**: Pause execution at specific lines.
  - **Step through code**: Execute the program line by line.
  - **Inspect variables**: View the current values of variables.
  - **Watch expressions**: Monitor the value of an expression as the program runs.
- **Code Review:** Have another programmer review your code to spot errors.
- **Test Cases:** Create specific input scenarios (test cases) that are designed to reveal errors.

### I/O Statements (Input/Output)

I/O statements are used to communicate with the user (or files).

- **printf() (Formatted Output):**
  - Used to display data on the console.
  - Syntax: printf("format string", arg1, arg2, ...);
  - **Format Specifiers:**
    - %d or %i: integer
    - %f: float
    - %lf: double
    - %c: character
    - %s: string
    - %p: pointer address

- %x or %X: hexadecimal
- %o: octal
- **Escape Sequences:**
  - \n: Newline
  - \t: Tab
  - \\: Backslash
  - \": Double quote
  - \': Single quote
  - \b: Backspace

Example:

```
int age = 25;
```

```
float height = 1.75;
```

```
char initial = 'J';
```

```
char name[] = "John"; // C-style string
```

```
printf("My name is %s, my initial is %c.\n", name, initial);
```

```
printf("I am %d years old and %.2f meters tall.\n", age, height); // .2f for 2 decimal places
```

- **scanf() (Formatted Input):**

- Used to read data from the console.
- Syntax: `scanf("format string", &variable1, &variable2, ...);`
- **Important:** Use the & (address-of) operator before variable names to pass their memory addresses, so scanf can store the input value directly into them.

Example:

```
int num1, num2;
```

```
float price;
```

```
printf("Enter two integers: ");
```

```
scanf("%d %d", &num1, &num2); // Read two integers
```

```
printf("Enter the price: ");
```

```
scanf("%f", &price); // Read a float
```

```
printf("You entered: %d, %d, %.2f\n", num1, num2, price);
```

## Constants, Variables, and Data Types

### Constants

Constants are fixed values that do not change during the execution of a program.

1. **Integer Constants:** Whole numbers (e.g., 10, -5, 0, 12345).
2. **Floating-point Constants:** Numbers with a decimal point or an exponent (e.g., 3.14, -0.001, 1.2e-5).
3. **Character Constants:** Single characters enclosed in single quotes (e.g., 'A', 'z', '5', '\n').
4. **String Literals:** A sequence of characters enclosed in double quotes (e.g., "Hello World", "C Programming"). String literals are actually arrays of characters terminated by a null character \0.
5. **const Keyword:** Used to declare a variable as a constant. Its value cannot be changed after initialization.

Example:

```
const float PI = 3.14159; // PI is a constant float
```

```
const int MAX_USERS = 100; // MAX_USERS is a constant integer
```

6. **#define Preprocessor Directive:** As discussed earlier, used to define symbolic constants.

Example:

```
#define PI_DEF 3.14159 // PI_DEF is a symbolic constant
```

### Variables

Variables are named memory locations used to store data. Their values can change during program execution.

- **Declaration:** Before using a variable, you must declare it, specifying its data type.
  - `dataType variableName;`
  - `int age;`
  - `float salary;`
- **Initialization:** Assigning an initial value to a variable during declaration.
  - `int count = 0;`
  - `char grade = 'A';`

## Data Types

Data types classify the kind of values a variable can hold and the operations that can be performed on them.

### Basic Data Types in C:

#### 1. **int (Integer):**

- Used to store whole numbers (positive, negative, or zero) without a decimal point.
- Typically occupies 2 or 4 bytes of memory, depending on the system/compiler.
- Range: Dependent on size (e.g., for 4 bytes, approximately  $\pm 2 \times 10^9$ ).
- **Modifiers for int:**
  - short int: Smaller range, typically 2 bytes.
  - long int: Larger range, typically 4 or 8 bytes.
  - long long int: Even larger range, typically 8 bytes (C99 standard).
  - unsigned int: Stores only non-negative numbers, effectively doubling the positive range.
  - signed int: Stores both positive and negative numbers (default for int).

Example:

```
int score = 100;
```

```
unsigned int population = 1300000000;
```

```
long long int distance_to_moon = 384400000000LL;
```

#### 2. **float (Floating-Point):**

- Used to store single-precision floating-point numbers (numbers with a decimal point).
- Typically occupies 4 bytes.
- Provides about 6-7 decimal digits of precision.

Example:

```
float temperature = 25.5f; // 'f' suffix is good practice for float literals
```

#### 3. **double (Double Precision Floating-Point):**

- Used to store double-precision floating-point numbers.
- Typically occupies 8 bytes.

- Provides about 15-17 decimal digits of precision. Preferred for calculations requiring higher accuracy.

Example:

```
double pi_value = 3.1415926535;
```

#### 4. char (Character):

- Used to store a single character.
- Internally, characters are stored as their ASCII (or other character set) integer values.
- Typically occupies 1 byte.

Example:

```
char grade = 'A';
```

```
char newline_char = '\n';
```

**Size and Range (Illustrative - actual values depend on compiler/system):**

Data Type	Bytes (typical)	Approximate Range	Format Specifier
char	1	-128 to 127 or 0 to 255 (signed/unsigned)	%c
short int	2	-32,768 to 32,767	%hd
int	4	-2,147,483,648 to 2,147,483,647	%d, %i
long int	4 or 8	Same as int or larger	%ld
long long int	8	$\pm 9 \times 10^{18}$	%lld
float	4	$\pm 3.4 \times 10^{38}$ (6-7 decimal digits)	%f
double	8	$\pm 1.7 \times 10^{308}$ (15-17 decimal digits)	%lf

You can use sizeof() operator to find the actual size of a data type on your system:

```
printf("Size of int: %zu bytes\n", sizeof(int));
```

## 1.3 Operators & Expressions, Unformatted and Formatted Input Output Statements, Data Type Casting

### Operators & Expressions

An **operator** is a symbol that tells the compiler to perform specific mathematical or logical manipulations. An **expression** is a combination of operators and operands (variables, constants, function calls) that evaluates to a single value.

#### Types of Operators in C:

1. **Arithmetic Operators:** Perform mathematical calculations.

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	10 / 3	3 (integer division)
%	Modulo (Remainder)	10 % 3	1

- **Integer Division:** When both operands of / are integers, the result is an integer (truncates the decimal part).
- **Modulo Operator:** Works only with integer operands.

2. **Relational Operators:** Used to compare two operands. They return 1 (true) or 0 (false).

Operator	Description	Example (a=5, b=3)	Result
==	Equal to	a == b	0 (false)
!=	Not equal to	a != b	1 (true)
>	Greater than	a > b	1 (true)

Operator	Description	Example (a=5, b=3)	Result
<	Less than	a < b	0 (false)
>=	Greater than or equal to	a >= b	1 (true)
<=	Less than or equal to	a <= b	0 (false)

3. **Logical Operators:** Used to combine or negate boolean expressions.

Operator	Description	Example (a=1, b=0)	Result
&&	Logical AND	a && b	0 (false)
			Logical OR
!	Logical NOT	!a	0 (false)

- && returns true if both operands are true.
- || returns true if at least one operand is true.
- ! negates the truth value of an operand.
- In C, any non-zero value is considered true, and 0 is considered false.

4. **Assignment Operators:** Used to assign values to variables.

Operator	Description	Example	Equivalent to
=	Simple Assignment	x = 10	
+=	Add and Assign	x += 5	x = x + 5
-=	Subtract and Assign	x -= 2	x = x - 2
*=	Multiply and Assign	x *= 3	x = x * 3
/=	Divide and Assign	x /= 2	x = x / 2

Operator	Description	Example	Equivalent to
%=	Modulo and Assign	x %= 3	x = x % 3

5. **Increment and Decrement Operators:** Used to increase or decrease a variable's value by 1.

Operator	Description	Example	Explanation
++	Increment	x++ (postfix) ++x (prefix)	Adds 1 to x
--	Decrement	x-- (postfix) --x (prefix)	Subtracts 1 from x

- **Prefix (++x, --x):** Increments/decrements the value *before* the expression is evaluated.
- **Postfix (x++, x--):** Increments/decrements the value *after* the expression is evaluated.

Example:

```
int i = 5, j = 5;
```

```
int a, b;
```

```
a = ++i; // i becomes 6, a becomes 6
```

```
b = j++; // j becomes 6, b becomes 5
```

```
printf("a=%d, i=%d\n", a, i); // Output: a=6, i=6
```

```
printf("b=%d, j=%d\n", b, j); // Output: b=5, j=6
```

6. **Bitwise Operators:** Perform operations on individual bits of integer operands. (More advanced topic, but good to be aware of).

- & (Bitwise AND)
- | (Bitwise OR)
- ^ (Bitwise XOR)
- ~ (Bitwise NOT)
- << (Left Shift)
- >> (Right Shift)

## 7. Special Operators:

- `sizeof()`: Returns the size in bytes of a variable or data type.
- `&` (Address-of operator): Returns the memory address of a variable.
- `*` (Pointer dereference operator): Accesses the value at a memory address (used with pointers).
- `? :` (Ternary/Conditional operator): A shorthand for if-else.

Example:

```
// Ternary operator example
```

```
int num = 10;
```

```
char *result = (num > 5) ? "Greater" : "Smaller";
```

```
printf("%s\n", result); // Output: Greater
```

Operator Precedence and Associativity:

Operators have a specific order of evaluation (precedence) and direction of evaluation (associativity) when multiple operators are present in an expression. For example, multiplication and division have higher precedence than addition and subtraction. Parentheses () can be used to override precedence.

### Unformatted and Formatted Input/Output Statements

We've already covered `printf()` and `scanf()` which are **formatted** I/O functions. They use format specifiers to interpret and display data.

### Unformatted Input/Output Functions:

These functions are simpler and typically deal with single characters or strings without explicit format specifiers.

#### 1. Character I/O:

- **`getchar()`**: Reads a single character from the standard input (keyboard). Returns an int (the ASCII value of the character) or EOF (End of File).

Example:

```
char ch;
```

```
printf("Enter a character: ");
```

```
ch = getchar();
```

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

- **`putchar()`**: Writes a single character to the standard output (screen). Takes an int argument (the ASCII value of the character).

Example:

```
char myChar = 'X';  
  
printf("Outputting a character: ");  
  
putchar(myChar);  
  
putchar('\n'); // Newline character
```

## 2. String I/O:

- **gets() (Avoid using in new code!):** Reads a string from the standard input until a newline character is encountered. It doesn't perform bounds checking, which can lead to buffer overflows (a security vulnerability).

Example

```
// DON'T USE THIS IN PRODUCTION CODE  
  
char name[50];  
  
printf("Enter your name: ");  
  
gets(name); // UNSAFE!  
  
printf("Hello, %s\n", name);
```

- **puts():** Writes a string to the standard output, automatically adding a newline character at the end.

Example:

```
char message[] = "Hello from puts!";  
  
puts(message); // Prints "Hello from puts!\n"
```

- **Safer Alternatives for String Input:**
  - **fgets():** Reads a string from a specified stream (e.g., stdin), with a size limit to prevent buffer overflows.

Example:

```
char safeName[50];  
  
printf("Enter your name (safe): ");  
  
// Read up to 49 characters + null terminator  
  
fgets(safeName, sizeof(safeName), stdin);  
  
// Remove trailing newline if present  
  
safeName[strcspn(safeName, "\n")] = 0;  
  
printf("Hello (safe), %s\n", safeName);
```

## Data Type Casting (Type Conversion)

Data type casting is the process of converting a value from one data type to another.

### 1. Implicit Type Conversion (Automatic Conversion/Coercion):

- Performed automatically by the compiler when different data types are involved in an expression.
- Conversions happen from a "smaller" type to a "larger" type (promotion) to avoid loss of data.
- Example: int to float, float to double.

Example:

```
int i = 10;
```

```
float f = 3.5;
```

```
double d = i + f; // i (int) is implicitly converted to float, then added to f. Result is float, then promoted to double for d.
```

```
printf("d = %f\n", d); // Output: d = 13.500000
```

### 2. Explicit Type Conversion (Type Casting):

- Performed by the programmer using the cast operator (dataType).
- Allows you to force a conversion, even if it might involve data loss (e.g., float to int).

Example:

```
float numFloat = 10.75;
```

```
int numInt;
```

```
numInt = (int)numFloat; // Explicitly cast float to int. Decimal part is truncated.
```

```
printf("numInt = %d\n", numInt); // Output: numInt = 10
```

```
int numerator = 10;
```

```
int denominator = 3;
```

```
float result;
```

```
result = (float)numerator / denominator; // Cast numerator to float before division to get float result
```

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

// Without casting:

// result = numerator / denominator; // Integer division: result would be 3.000000

### When to use type casting:

- To avoid integer division when one operand is an integer and you need a floating-point result.
- To convert between compatible but different data types when explicit control is needed.
- When passing arguments to functions that expect a specific type.

## 1.4 Decision Making with IF – Statement, IF – Else and Nested IF

Decision-making statements allow a program to execute different blocks of code based on certain conditions.

### 1. if Statement

The if statement is the simplest decision-making statement. It executes a block of code if a specified condition is true.

#### Syntax:

Example:

```
if (condition) {
```

```
    // Code to be executed if the condition is true
```

```
}
```

```
// Code outside the if block (always executes regardless of condition)
```

- The condition is an expression that evaluates to either true (non-zero) or false (zero).
- The curly braces {} define the block of code. If there's only one statement, the braces are optional but recommended for clarity.

#### Example:

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

```
int main() {
```

```
    int age = 18;
```

```
    if (age >= 18) {
```

```
        printf("You are eligible to vote.\n");
```

```
    }
```

```
    printf("Program continues here.\n");
```

```
    return 0;
}
```

## 2. if-else Statement

The if-else statement provides two paths of execution: one if the condition is true, and another if the condition is false.

### Syntax:

```
if (condition) {
    // Code to be executed if the condition is true
} else {
    // Code to be executed if the condition is false
}
```

### Example:

```
#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);
    }

    return 0;
}
```

## 3. if-else if-else Ladder (Chained if-else)

This structure is used when you have multiple conditions to check, and each condition leads to a different action.

### Syntax:

```
if (condition1) {  
    // Code if condition1 is true  
} else if (condition2) {  
    // Code if condition1 is false AND condition2 is true  
} else if (condition3) {  
    // Code if condition1 and condition2 are false AND condition3 is true  
} else {  
    // Code if all preceding conditions are false  
}
```

### **Example: Grading System**

```
#include <stdio.h>  
  
int main() {  
    int score;  
    printf("Enter your score: ");  
    scanf("%d", &score);  
    if (score >= 90) {  
        printf("Grade: A\n");  
    } else if (score >= 80) {  
        printf("Grade: B\n");  
    } else if (score >= 70) {  
        printf("Grade: C\n");  
    } else if (score >= 60) {  
        printf("Grade: D\n");  
    } else {  
        printf("Grade: F\n");  
    }  
    return 0;  
}
```

### **4. Nested if Statements**

When an if or else if statement contains another if or if-else statement, it's called nesting. This allows for more complex decision logic.

**Syntax:**

```
if (condition1) {  
    // Outer if block  
    if (nested_condition1) {  
        // Code if condition1 AND nested_condition1 are true  
    } else {  
        // Code if condition1 is true AND nested_condition1 is false  
    }  
} else {  
    // Outer else block  
    if (nested_condition2) {  
        // Code if condition1 is false AND nested_condition2 is true  
    } else {  
        // Code if condition1 is false AND nested_condition2 is false  
    }  
}
```

**Example: Checking Eligibility for a Ride (Age and Height)**

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

```
int main() {  
    int age;  
    float height;  
    printf("Enter your age: ");  
    scanf("%d", &age);  
    printf("Enter your height in meters: ");  
    scanf("%f", &height);  
  
    if (age >= 12) {  
        if (height >= 1.5) {
```

```

    printf("You are eligible for the ride!\n");
} else {
    printf("You are old enough, but not tall enough for the ride.\n");
}
} else {
    printf("You are not old enough for the ride.\n");
}
return 0;
}

```

### Important Considerations for if statements:

- **Semicolon after if:** Never put a semicolon immediately after the if (condition) or else statement, as it would terminate the statement, making the block of code always execute or be associated incorrectly.
  - if (condition); { /\* code \*/ } is usually a bug. The semicolon makes the if statement an empty statement, and the block {} will always execute.
- **Braces {}:** Always use curly braces for code blocks, even if there's only one statement. This prevents logical errors when adding more statements later.
- **Boolean Values:** In C, 0 is considered false, and any non-zero value (typically 1) is true.

## 1.5 While and do-while, for loop, Break, Continue, goto and switch statements

Looping statements (or iteration statements) allow a block of code to be executed repeatedly as long as a certain condition is met.

### 1. while Loop

The while loop repeatedly executes a block of code as long as a given condition is true. The condition is checked *before* each iteration.

#### Syntax:

```

while (condition) {
    // Code to be executed repeatedly
    // (Must contain something that eventually makes the condition false)
}

```

#### Example: Print numbers from 1 to 5

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

```
int main() {
```

```

int i = 1; // Initialization
while (i <= 5) { // Condition
    printf("%d ", i);
    i++; // Update/Increment
}

printf("\n");

return 0;
}

```

## 2. do-while Loop

The do-while loop is similar to the while loop, but the code block is executed at least once *before* the condition is checked.

### Syntax:

```

do {
    // Code to be executed repeatedly
    // (Must contain something that eventually makes the condition false)
} while (condition); // Don't forget the semicolon here!

```

### Example: User input validation (gets at least one input)

```

#include <stdio.h>

int main() {
    int num;

    do {
        printf("Enter a positive number (or 0 to exit): ");
        scanf("%d", &num);
        if (num < 0) {
            printf("Invalid input! Please enter a positive number.\n");
        }
    } while (num < 0); // Loop continues if number is negative

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

    return 0;
}

```

### 3. for Loop

The for loop is typically used when you know in advance how many times you want to iterate, or when the loop's control variable has a clear starting point, ending point, and increment/decrement.

#### Syntax:

```
for (initialization; condition; update) {  
    // Code to be executed repeatedly  
}
```

- **Initialization:** Executed once at the beginning of the loop.
- **Condition:** Checked before each iteration. If true, the loop body executes. If false, the loop terminates.
- **Update:** Executed after each iteration of the loop body.

#### Example: Print numbers from 1 to 5

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

**Nested for loops:** One loop inside another, often used for working with 2D structures (like matrices) or patterns.

// Example: Print a 3x3 square of asterisks

```
for (int i = 0; i < 3; i++) { // Outer loop for rows  
    for (int j = 0; j < 3; j++) { // Inner loop for columns  
        printf("* ");  
    }  
    printf("\n"); // Move to the next line after each row  
}
```

#### 4. break Statement

The break statement is used to immediately terminate the innermost loop (or switch statement) in which it appears. Program control resumes at the statement immediately following the loop/switch.

##### Example: Searching for a number

```
#include <stdio.h>

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int search_num = 30;
    int found = 0;
    for (int i = 0; i < 5; i++) {
        if (numbers[i] == search_num) {
            printf("%d found at index %d\n", search_num, i);
            found = 1;
            break; // Exit the loop once found
        }
    }
    if (!found) {
        printf("%d not found.\n", search_num);
    }
    return 0;
}
```

#### 5. continue Statement

The continue statement is used to skip the rest of the current iteration of the loop and proceed to the next iteration.

##### Example: Print odd numbers from 1 to 10

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) { // If number is even
            continue; // Skip the rest of this iteration
        }
        printf("%d ", i);
    }
    printf("\n");
    return 0;
}
```

```

    }

    printf("%d ", i); // Only odd numbers will be printed
}

printf("\n");

return 0;

}

```

## 6. goto Statement (Avoid using if possible)

The goto statement is an unconditional jump statement that transfers program control to a specified label within the same function. While it exists in C, its use is generally discouraged as it can lead to spaghetti code, making programs hard to read, debug, and maintain.

### Syntax:

```

goto label;

// ... some code ...

label:

    // Code to jump to

```

### Example (Illustrative, not recommended):

```

#include <stdio.h>

int main() {
    int i = 1;

start_loop:
    if (i <= 5) {
        printf("%d ", i);

        i++;

        goto start_loop; // Jump back to start_loop
    }

    printf("\nLoop finished.\n");

    return 0;

}

```

### Why goto is generally avoided:

- **Readability:** Makes code difficult to follow the flow of execution.
- **Maintainability:** Changes in one part of the code can have unexpected consequences due to distant jumps.
- **Debugging:** Tracing logic becomes much harder.
- Structured programming constructs (if, for, while, functions) almost always provide better and clearer alternatives.

## 7. switch Statement

The switch statement is a multi-way decision-making statement that allows a variable (or expression) to be tested for equality against a list of values. It provides an alternative to a long if-else if-else ladder when you are comparing a single value against multiple constant possibilities.

### Syntax:

```
switch (expression) {  
    case constant1:  
        // Code to execute if expression == constant1  
        break; // Important! Exits the switch  
    case constant2:  
        // Code to execute if expression == constant2  
        break;  
    // ... more cases ...  
    default:  
        // Code to execute if no case matches (optional)  
        break; // Good practice, though not strictly necessary here  
}
```

- The expression must evaluate to an integer type (including char).
- case labels must be unique constant integer expressions.
- The break statement is crucial. Without it, execution will "fall through" to the next case label, executing its code as well (this is known as "fall-through behavior" and is often a source of bugs if not intended).
- The default case is optional and executes if none of the case values match the expression.

### Example: Simple Calculator using switch

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

```
int main() {
```

```
    char operator;
```

```
    double n1, n2, result;
```

```
    printf("Enter an operator (+, -, *, /): ");
```

```
    scanf(" %c", &operator); // Note the space before %c to consume leftover newline
```

```
    printf("Enter two operands: ");
```

```
    scanf("%lf %lf", &n1, &n2);
```

```
    switch (operator) {
```

```
        case '+':
```

```
            result = n1 + n2;
```

```
            printf("%.2lf + %.2lf = %.2lf\n", n1, n2, result);
```

```
            break;
```

```
        case '-':
```

```
            result = n1 - n2;
```

```
            printf("%.2lf - %.2lf = %.2lf\n", n1, n2, result);
```

```
            break;
```

```
        case '*':
```

```
            result = n1 * n2;
```

```
            printf("%.2lf * %.2lf = %.2lf\n", n1, n2, result);
```

```
            break;
```

```
        case '/':
```

```
            if (n2 != 0) {
```

```
                result = n1 / n2;
```

```
                printf("%.2lf / %.2lf = %.2lf\n", n1, n2, result);
```

```
            } else {
```

```
                printf("Error: Division by zero is not allowed.\n");
```

```
            }
```

```
        break;
```

default:

```
printf("Error: Invalid operator.\n");
```

```
break;
```

```
}
```

```
return 0;
```

```
}
```

This concludes UNIT I of your C Programming class notes. Make sure to practice these concepts thoroughly by writing and running various programs.