PROGRAMMING CONCEPTS

5.1 PROGRAMMING CONCEPT

5.1.1 INTRODUCTION TO PROGRAMMING LANGUAGES

A programming language is a formal set of instructions and rules that a programmer uses to write a program, which a computer can then follow to perform a task.

A computer only understands electrical signals in the form of 0s and 1s. A programming language gives humans a way to write instructions in a form they can actually read and reason about, which is then converted into a form the machine can run.

Term	Meaning
Program	A set of instructions given to a computer to perform a specific task
Programming Language	A formal language with its own rules, used to write programs (e.g. C, C++, Java, Python)
Programmer	The person who writes computer programs
Source Code	The program as written by the programmer, in a high-level language, before translation
Object Code / Machine Code	The translated version of the source code, in binary, that the processor can actually run

Writing a program generally follows the same rough sequence, no matter which language is used: understand the problem, plan the logic (using an algorithm, flowchart, or pseudocode), write the source code, translate it into machine code, and then test and fix it. Skipping the planning step is one of the most common reasons beginner programs end up buggy or hard to follow.

5.1.2 Low Level, High Level and 4GL Programming Languages

Programming languages are grouped by how close they sit to the hardware versus how close they sit to human language.

- **Low Level Language:** written close to the hardware, using machine code (pure binary) or assembly language (short mnemonics like MOV, ADD). Fast to run but hard to write and read.
- **High Level Language:** written closer to human language, using English-like keywords and readable syntax. Easier to write, debug, and maintain, though it needs to be translated before the machine can run it. Examples: C, C++, Java, Python.

- **Fourth Generation Language (4GL):** designed to be even more accessible than a typical high-level language, often used for a specific purpose such as database queries or report generation, with less code required to get a result. SQL is a common example.

5.1.3 Compiler, Interpreter and Assembler

Source code written by a programmer has to be translated into a form the processor can execute. Different languages use different translators to do this.

Translator	How It Works	Example
Compiler	Translates the entire source code into machine code in one go, before the program runs. Errors are reported all together after compilation.	C, C++
Interpreter	Translates and executes the source code line by line, stopping as soon as it hits an error.	Python, early BASIC
Assembler	Translates assembly language (mnemonics) directly into machine code.	Assembly language programs

C is a compiled language: the whole program is translated by the compiler into an executable file first, and only then is it run.

5.1.4 Syntax, Semantic and Runtime Errors

- **Syntax Error:** a mistake in the grammar of the language, such as a missing semicolon or an unmatched bracket. The compiler catches this before the program can even run.
- **Semantic Error:** the code follows the rules of the language correctly but does not do what the programmer actually intended, such as using the wrong variable in a formula.
- **Runtime Error:** an error that only appears while the program is running, such as dividing a number by zero or trying to access an array index that doesn't exist.

5.1.5 Control Structures: Sequence, Selection and Iteration

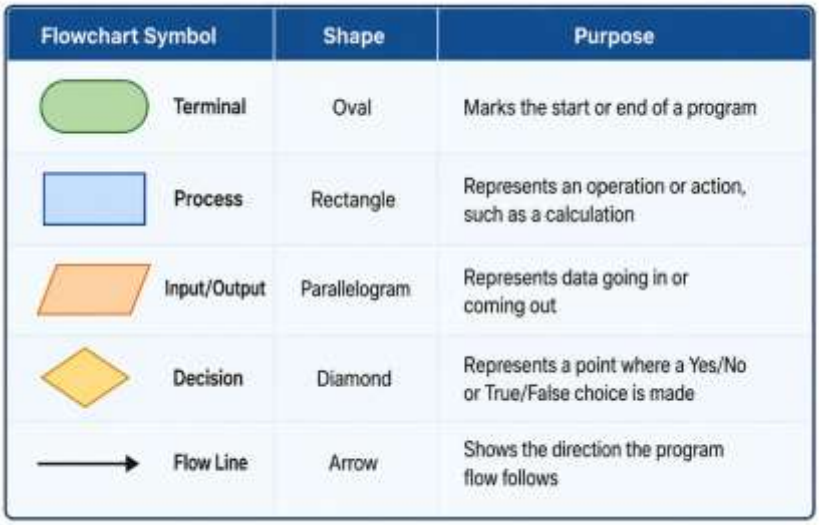
A control structure decides the order in which the statements of a program are executed.

Control Structure	What It Does
Sequence	Statements run one after another, in the exact order they are written
Selection	The program picks a path to follow based on whether a condition is true or false (if, if-else, switch)
Iteration	A block of statements repeats until a condition is met (while, do-while, for)

5.1.6 Program Design Tools: Algorithm, Flowchart and Pseudocode

Before writing actual code, a programmer usually plans out the logic of a program using one of these tools.

- **Algorithm:** a step-by-step written procedure for solving a problem, described in plain language rather than any particular programming language. A good algorithm is finite (it must end), definite (every step is precise and unambiguous), and effective (each step is simple enough to actually carry out).
- **Flowchart:** a diagram that represents the steps of a program visually, using standard symbols.
- **Pseudocode:** a structured, code-like description of a program's logic, written in plain English but following a programming-style layout, without worrying about the exact syntax of any real language.



Flowchart Symbol	Shape	Purpose
Terminal	Oval	Marks the start or end of a program
Process	Rectangle	Represents an operation or action, such as a calculation
Input/Output	Parallelogram	Represents data going in or coming out
Decision	Diamond	Represents a point where a Yes/No or True/False choice is made
Flow Line	Arrow	Shows the direction the program flow follows

Solved Example: Design Tools for "Find the Largest of Two Numbers"

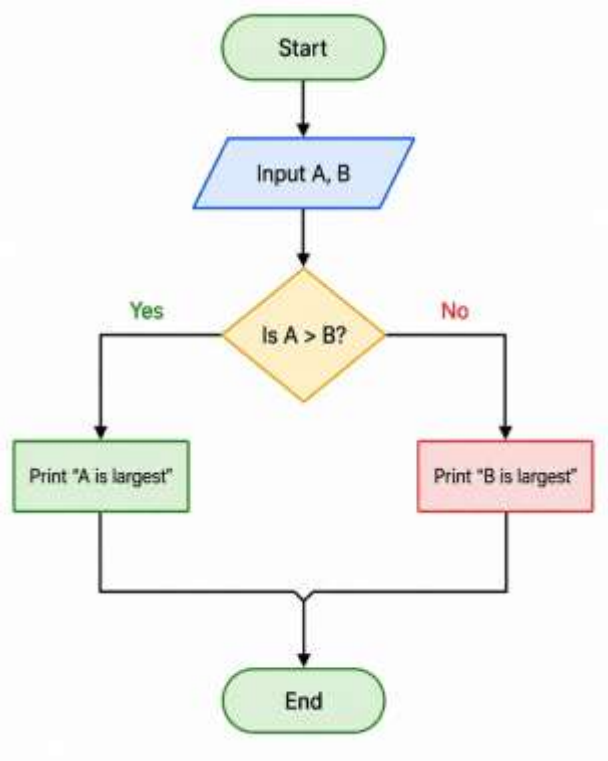
Algorithm:

1. Start
2. Read two numbers, A and B
3. If A is greater than B, display A as the largest
4. Otherwise, display B as the largest
5. Stop

Pseudocode:

```
BEGIN
 READ A, B
 IF A > B THEN
  PRINT "A is largest"
 ELSE
  PRINT "B is largest"
 END IF
END
```

Flowchart:



5.1.7 Absolute Binary, BCD, ASCII and Unicode

A computer stores every type of data, whether it is a number, a letter, or a symbol, in binary. These four terms describe different ways that binary is used to represent information.

- **Absolute Binary:** data represented directly in raw 0s and 1s, the actual form in which a computer's hardware stores and processes everything.
- **BCD (Binary Coded Decimal):** a way of encoding numbers where each individual decimal digit is stored as its own separate 4-bit binary value, rather than converting the whole number to binary at once. For example, the decimal number 25 is stored as 0010 0101 (2 = 0010, 5 = 0101), not as the binary equivalent of 25 as a whole.
- **ASCII (American Standard Code for Information Interchange):** a character encoding standard that assigns a unique numeric code (originally 7-bit, later extended to 8-bit) to English letters, digits, punctuation, and control characters. For example, the capital letter 'A' is represented as 65 in ASCII.
- **Unicode:** a character encoding standard built to represent text from virtually every writing system in the world, not just English, using more bits per character than ASCII to fit the much larger number of symbols. This is what allows a computer to correctly display Devanagari script alongside English text.

5.2 C PROGRAMMING LANGUAGE

5.2.1 INTRODUCTION AND FEATURES OF C

C is a general-purpose, procedural, high-level programming language, developed by Dennis Ritchie at Bell Laboratories in the early 1970s.

C shows up everywhere from operating systems to compilers to everyday applications, mostly because it's efficient and gives a programmer close control over how a program runs.

Feature	What It Means
Structured Language	A program is organized into functions, and logic is broken into clear, ordered steps
Middle-level Language	Reads like a high-level language, but still allows fairly direct control over hardware and memory, like a low-level language
Portability	C code can run on different types of computers with little or no change
Rich Library Functions	Comes with a large set of built-in functions (in header files) for common tasks
Memory Management	Allows direct memory access using pointers, giving fine-grained control

Fast Execution Speed	Compiled C programs are generally very fast, since they run as native machine code

5.2.2 Structure of a C Program

Every C program follows roughly the same basic layout:

```
#include <stdio.h>          // Preprocessor Directive
int globalVar = 10;         // Global Variable Declaration (Optional)
void myFunction();          // Function Declaration (Optional)
int main() {                // Execution always starts here
   int localVar = 20;       // Local Variable Declaration
   printf("Hello, World!\n");
   printf("Global variable: %d\n", globalVar);
   printf("Local variable: %d\n", localVar);
   myFunction();            // Function call
   return 0;                // Successful termination
}
void myFunction() {         // Function Definition (Optional)
   printf("This is a user-defined function.\n");
}
```

- **Preprocessor directives** start with # and are processed before compilation. #include <stdio.h> tells the compiler to bring in the standard input/output header file.
- **Global variables** are declared outside every function and can be accessed from anywhere in the program.
- **main()** is the most important part of any C program. Execution always begins and ends there, and every program must have exactly one.
- **Local variables** are declared inside a function and can only be accessed within it.
- **return 0;** inside main() signals that the program ran successfully; a non-zero value usually signals an error.

5.2.3 C Preprocessor and Header Files

A preprocessor directive is an instruction that starts with a # symbol and is handled before the actual compilation begins. The most common ones are #include, which brings in a header file, and #define, which creates a macro.

A header file (with a .h extension) contains the declarations of functions and macros that belong to the C standard library. For example, stdio.h must be included before a program can use functions like printf() and scanf(), and string.h must be included before using functions like strlen() or strcpy().

5.2.4 Character Set Used in C

The character set is the collection of characters that C recognizes as valid when writing a program. It includes:

- Alphabets: uppercase A-Z and lowercase a-z
- Digits: 0-9
- Special characters: such as + - * / % ; , . ( ) { } [ ] # &
- White space characters: blank space, tab, and newline

5.2.5 Use of Comments

A comment is text in a program that the compiler ignores completely. It exists purely to help a human reader understand the code.

- **Single-line comment:** starts with // and runs to the end of the line
- **Multi-line comment:** starts with /* and ends with */, and can span several lines

5.2.6 Identifiers, Keywords and Tokens

- **Identifiers:** the names a programmer gives to variables, functions, and other elements in a program, such as total or calculateArea.
- **Keywords:** reserved words that already have a fixed meaning in C, such as int, if, while, and return. A keyword can never be used as an identifier.
- **Tokens:** the smallest individual units a C compiler recognizes while reading a program, which includes keywords, identifiers, constants, operators, and punctuation.

Naming rules for an identifier:

- Must begin with an alphabet (A-Z, a-z) or an underscore (_)
- Can contain alphabets, digits (0-9), and underscores after the first character
- Cannot contain spaces or special symbols other than an underscore

- Cannot be a reserved keyword
- Case-sensitive — age and Age are treated as two different identifiers

5.2.7 Basic Data Types in C

Data Type	Description	Typical Size	Format Specifier
int	Whole numbers, positive or negative	2 or 4 bytes	%d or %i
float	Single-precision decimal numbers	4 bytes	%f
double	Double-precision decimal numbers, more accurate than float	8 bytes	%lf
char	A single character	1 byte	%c
void	Represents the absence of a value or type	-	-

5.2.8 Constants and Variables

- **Variable:** a named location in memory that holds a value which can change during program execution, such as int score = 0;. It must be declared with a data type before use.
- **Constant:** a fixed value that does not change while the program is running.

Types of constants: integer (10, -5), floating-point (3.14, 2.5e-3), character ('A', enclosed in single quotes), and string literals ("Nepal", enclosed in double quotes).

A constant can be defined either with the const keyword (const float PI = 3.14159;) or with the #define preprocessor directive (#define PI 3.14159).

5.2.9 Type Specifiers

A type specifier modifies a basic data type to adjust its size or the range of values it can store. C provides four: short, long, signed, and unsigned. For example, unsigned int can only store non-negative numbers, but in exchange it can hold a larger positive range than a plain signed int.

5.2.10 Simple and Compound Statements

- **Simple statement:** a single instruction ending with a semicolon, such as a = a + 1;

- **Compound statement (block):** a group of statements enclosed within curly braces { }, treated by the compiler as a single unit, commonly used inside loops and conditions.

5.2.11 Operators and Expressions

An operator is a symbol that performs an operation on operands, and an expression is a combination of operators, operands, and function calls that evaluates to a single value.

Type	Operators	Purpose
Arithmetic	+ - * / %	Performs mathematical calculations
Relational	== != < > <= >=	Compares two values, returns true or false
Logical	&& || !	Combines or negates conditions
Assignment	= += -= *= /= %=	Assigns or updates a value in a variable
Increment/Decrement	++ --	Increases or decreases a variable's value by 1 (pre or post)
Conditional (ternary)	? :	A shorthand for a simple if-else, e.g. (a > b) ? a : b

Note: pre-increment (++a) changes the value before it is used in an expression, while post-increment (a++) changes it after. The same applies to decrement.

5.2.12 Input/Output Functions

C uses two standard library functions for basic input and output, both defined in stdio.h.

```c
#include <stdio.h>
int main() {
    int rollNo;
    char grade;
    float percentage;

    printf("Enter your Roll Number: ");
    scanf("%d", &rollNo);          // & is the address-of operator

    printf("Enter your Grade (A/B/C): ");
    scanf(" %c", &grade);          // space before %c clears leftove

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

    printf("\nRoll Number: %d\n", rollNo);
    printf("Grade: %c\n", grade);
    printf("Percentage: %.2f%%\n", percentage);
    return 0;
}
```

```
Output
Enter your Roll Number: 101
Enter your Grade (A/B/C): A
Enter your Percentage: 78.5

Roll Number: 101
Grade: A
Percentage: 78.50%
```

Common format specifiers: *%d/%i* for integers, *%f* for floats (%.2f for 2 decimal places), *%c* for characters, *%s* for strings, *\n* for a newline, *\t* for a tab space.

The & (address-of) operator is essential with *scanf()*, since it tells the function the memory address to store the input at. For reading a string with *scanf("%s", name)*, & is not needed because an array name already acts as a pointer to its own base address. Note that *scanf()* with *%s* stops reading at the first space, so *fgets()* is used instead when a string may contain spaces.

5.2.13 Selection Control Statement: Decisions

Selection statements let a program choose which block of code to run, based on a condition.

if statement

```c
int age = 19;
if (age >= 18) {
    printf("You are eligible to vote.\n");
}
```

```
Output
You are eligible to vote.
```

if-else statement

```
int marks = 65;
if (marks >= 40) {
    printf("You have passed.\n");
} else {
    printf("You have failed.\n");
}
```

```
Output
You have passed.
```

Nested if-else / if-else-if ladder

```
int num = 10;
if (num > 0) {
    printf("The number is positive.\n");
    if (num % 2 == 0) {
        printf("It is even.\n");
    } else {
        printf("It is odd.\n");
    }
} else if (num < 0) {
    printf("The number is negative.\n");
} else {
    printf("The number is zero.\n");
}
```

```
Output (for input 10)
The number is positive.
It is even.
```

switch statement

```
char grade = 'B';
switch (grade) {
    case 'A': printf("Excellent!\n"); break;
    case 'B': printf("Good job!\n"); break;
    case 'C': printf("Satisfactory.\n"); break;
    default:  printf("Invalid grade.\n");
}
```

```
Output
Good job!
```

The break statement exits the switch block once a match is found. If it's left out, execution "falls through" to the next case. The default case runs only if nothing else matched.

Feature	if-else	switch

Condition type	Any boolean expression, using relational/logical operators	A single variable compared against fixed integer/character values
Flexibility	Can handle ranges and complex, multi-part conditions	Best suited to a fixed, known set of values
Execution	Runs the first block whose condition is true	Runs the block matching the case value; needs break to avoid fall-through

5.2.14 Iteration Control Statement: Looping

Iteration statements repeat a block of code as long as a condition holds true.

for loop

```
for (int i = 1; i <= 5; i++) {
   printf("%d\n", i);
}
```

Output1
2
3
4
5

while loop

```
int i = 1;
while (i <= 5) {
   printf("%d\n", i);
   i++;
}
```

Output1
2
3
4
5

do-while loop

```
int i = 1;
do {
   printf("%d\n", i);
   i++;
} while (i <= 5);
```

Output1

2
3
4
5

Loop	Condition Checked	Minimum Executions
while	Before the body runs	0 (may never run)
do-while	After the body runs	1 (always runs once)
for	Before the body runs, with init/update built in	0 (may never run)

A **nested loop** is a loop placed inside another loop, most often used to work through a 2D array or matrix row by row.

5.2.15 Arrays: 1D and 2D

An array is a collection of elements of the same data type, stored in adjoining memory locations and accessed using a single variable name with an index that starts from 0.

1D (One-Dimensional) Array

A 1D array is a simple, linear list of values, declared as dataType arrayName[size];.

```
int studentMarks[5] = {75, 80, 65, 90, 88};
int i;
for (i = 0; i < 5; i++) {
  printf("Student %d: %d\n", i + 1, studentMarks[i]);
}
```

OutputStudent 1: 75

Student 2: 80

Student 3: 65

Student 4: 90

Student 5: 88

2D (Two-Dimensional) Array

A 2D array is an array of arrays, visualized as a table of rows and columns, declared as dataType arrayName[rows][columns];. Each element needs two indices, one for the row and one for the column.

```
// Matrix addition using two 2D arrays
        for (i = 0; i < 3; i++) {
          for (j = 0; j < 3; j++) {
             c[i][j] = a[i][j] + b[i][j];
          }
        }
```

5.2.16 Strings and String Functions

A string in C is an array of characters that ends with a special null character (\0), which marks where the string stops. The string.h header file provides functions to work with strings.

Function	What It Does	Example
strlen()	Returns the length of a string, not counting the null character	strlen("Nepal") returns 5
strcpy()	Copies one string into another	strcpy(dest, "Hello")
strcat()	Joins (concatenates) one string onto the end of another	strcat(s1, " World")
strcmp()	Compares two strings; returns 0 if equal	strcmp("apple","banana") returns negative
strrev()	Reverses a string	strrev("C Prog") gives "gorP C"
strupr()	Converts a string to uppercase	strupr("hello") gives "HELLO"
strlwr()	Converts a string to lowercase	strlwr("HELLO") gives "hello"

```c
#include <stdio.h>
#include <string.h>
int main() {
    char name[20] = "Kathmandu";
    char city[20];
    char greeting[30] = "Namaste";

    printf("Length of name: %d\n", strlen(name));
    strcpy(city, name);
    printf("Copied city: %s\n", city);
    strcat(greeting, " Nepal!");
    printf("Concatenated: %s\n", greeting);

    return 0;
}
```

```
Output
Length of name: 9
Copied city: Kathmandu
Concatenated: Namaste Nepal!
```

Note: strrev(), strupr(), and strlwr() are not part of the official ANSI C standard, but they are available as common extensions in compilers such as Turbo C and GCC. For strict portability, the same result can be achieved manually using toupper()/tolower() from ctype.h inside a loop.

Solved Program Examples

A definition is easier to remember once it has been used in an actual program. The programs below cover the most commonly asked coding questions from this unit, each with its expected output.

1. Sum of Two Numbers

1. Sum Of Two Numbers

```c
#include <stdio.h>
int main() {
    int a, b, sum;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    sum = a + b;
    printf("Sum = %d", sum);
    return 0;
}
```

```
Output
Enter two numbers: 4 7
Sum = 11
```

2. Check Whether a Number is Even or Odd

int main() {

2. Check Whether A Number Is Even Or Odd

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

3. Factorial of a Number Using a Loop

3. Factorial Of A Number Using A Loop

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

4. Sum and Average of 5 Numbers Using a 1D Array

4. Sum And Average Of 5 Numbers Using A 1D Array

```c
#include <stdio.h>
int main() {
    int numbers[5], sum = 0, i;
    float average;

    printf("Enter 5 numbers:\n");
    for (i = 0; i < 5; i++) {
        printf("Enter number %d: ", i + 1);
        scanf("%d", &numbers[i]);
        sum += numbers[i];
    }
    average = (float)sum / 5;

    printf("\nSum = %d\n", sum);
    printf("Average = %.2f\n", average);
    return 0;
}
```

5. Input and Display a 2x3 Matrix

5. Input And Display A 2x3 Matrix

```c
#include <stdio.h>
int main() {
    int matrix[2][3], i, j;

    printf("Enter elements for a 2x3 matrix:\n");
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++) {
            printf("Element [%d][%d]: ", i, j);
            scanf("%d", &matrix[i][j]);
        }
    }

    printf("\n--- Matrix ---\n");
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++) {
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

6. Reverse a String

6. Reverse A String

```c
#include <stdio.h>
#include <string.h>
int main() {
    char name[50];
    printf("Enter a string: ");
    scanf("%s", name);
    strrev(name);
    printf("Reversed = %s", name);
    return 0;
}
```

ACTIVITIES AND PRACTICE

Write an algorithm to check if a given number is even or odd.

An algorithm is a step-by-step procedure to solve a problem:

START

Declare an integer variable num.

Read num from the user.

If num % 2 == 0, display "Number is Even".

Else, display "Number is Odd".

STOP

What are the key differences between a while loop and a do-while loop?

A while loop checks its condition before executing the body, so it may not run at all if the condition starts out false. A do-while loop executes the body once first, then checks the condition, so it always runs at least once. For example, while (i < 0) never runs if i starts at 5, but the equivalent do { ... } while (i < 0) still runs once before stopping.

Why isn't the & operator always needed with scanf()?

The & (address-of) operator gives scanf() the memory address of a variable, so it knows exactly where to store the value typed in. Arrays are the exception: when reading a string with %s, the array's name already refers to its base memory address on its own, so adding & in front of it is unnecessary.