

5.1 Programming Concepts

5.1.1 Introduction to Programming Languages

1. A programming language is a medium to communicate with computers.
2. It is used to write programs that perform tasks.
3. Languages follow rules called syntax and semantics.
4. Programs are written in source code form.
5. They are converted into machine language.
6. Used to solve real-world problems.
7. Makes development easier and faster.
8. Different languages serve different purposes.
9. Examples: C, Java, Python.
10. Programming improves logical thinking.

5.1.2 Low Level, High Level and 4GL

1. Low-level languages are close to hardware.
2. Machine language uses binary (0,1).
3. Assembly language uses mnemonics.
4. High-level languages are human friendly.
5. Example: C, Java, Python.
6. They are machine independent.
7. Need translators (compiler/interpreter).
8. 4GL = Fourth Generation Language.
9. Used for database and reports.
10. Example: SQL.

5.1.3 Compiler, Interpreter, Assembler

1. Translator converts program to machine code.
2. Compiler translates entire program at once.
3. It shows errors after compilation.
4. Interpreter translates line by line.
5. Stops when error occurs.
6. Assembler converts assembly code.

7. Compiled programs run faster.
8. Interpreted programs are slower.
9. C uses compiler.
10. Python uses interpreter.

5.1.4 Syntax, Semantic, Runtime Errors

1. Errors are mistakes in program.
2. Syntax error = grammar mistake.
3. Example: missing semicolon.
4. Semantic error = wrong meaning/logic.
5. Example: wrong formula.
6. Runtime error occurs during execution.
7. Example: divide by zero.
8. Logical errors give wrong output.
9. Debugging removes errors.
10. Error handling improves reliability.

5.1.5 Control Structures

1. Control structures control program flow.
2. Sequence = default execution.
3. Selection = decision making.
4. Example: if, switch.
5. Iteration = repetition.
6. Example: for, while.
7. Used to solve complex problems.
8. Makes programs dynamic.
9. Reduces code repetition.
10. Essential in all languages.

5.1.6 Algorithm, Flowchart, Pseudocode

1. Program design tools help plan program.
2. Algorithm = step-by-step solution.
3. Must be clear and finite.

4. Flowchart = graphical representation.
5. Uses symbols (oval, rectangle).
6. Pseudocode = simple English code.
7. Easy to understand.
8. Used before coding.
9. Helps find logical errors.
10. Improves program structure.

5.1.7 Binary, BCD, ASCII, Unicode

1. Binary system uses 0 and 1.
2. Used internally by computers.
3. BCD = Binary Coded Decimal.
4. Each digit stored in 4 bits.
5. ASCII = American Standard Code.
6. Stores characters (A = 65).
7. Unicode stores all languages.
8. Uses more bits than ASCII.
9. Supports global communication.
10. Used in text processing.

1. Binary

Definition:

- Binary is a **number system with base 2**.
- Only uses digits **0 and 1**.
- Used **internally by computers** to store and process data.

Example:

Decimal 5 → Binary: 101

Decimal 10 → Binary: 1010

2. BCD (Binary Coded Decimal)

Definition:

- Each **decimal digit** is represented by a **4-bit binary number**.
- Combines decimal and binary systems.
- Useful in calculators and digital clocks.

Example:

Decimal 59 → BCD:

- 5 → 0101
 - 9 → 1001
- So 59 → 0101 1001

Table Example:

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101

3. ASCII (American Standard Code for Information Interchange)

Definition:

- Stores **characters as numeric codes**.
- Uses **7 or 8 bits** to represent each character.
- Supports letters, digits, punctuation, and control characters.

Example:

- 'A' → 65
- 'a' → 97
- '0' → 48

Table Example:

Character	ASCII Code
A	65
B	66
a	97
b	98
0	48
1	49

C Program Example:

```
#include <stdio.h>
int main() {
    char ch = 'A';
    printf("ASCII of %c = %d", ch, ch);
    return 0;
}
```

Output:

ASCII of A = 65

4. Unicode

Definition:

- Stores **all characters from all languages**.
- Uses **16 bits or more** per character.
- Supports **global languages** including special symbols, emojis, etc.
- Compatible with ASCII (first 128 codes same as ASCII).

Example:

- 'A' → 0041
- 'ह' → 0939 (Hindi character)
- '😊' → 1F60A (Emoji)

5.2 C Programming

5.2.1 Introduction & Features of C

1. Developed by Dennis Ritchie.
2. Structured programming language.
3. Middle-level language.
4. Fast execution.
5. Portable.

6. Rich library functions.
7. Supports pointers.
8. Used in system programming.
9. Case sensitive.
10. Widely used.

5.2.2 Structure of C Program

1. Documentation section.
2. Preprocessor section.
3. Global declarations.
4. main() function.
5. Local declarations.
6. Program statements.
7. User-defined functions.
8. Return statement.
9. Header files included.
10. Program starts from main().

```
/* Program to add two numbers */ #include <stdio.h>

// Global declaration

int globalVar = 50;

int main() {

    // Local declarations

    int a = 10, b = 20; int sum;

    // Program statements
    sum = a + b + globalVar; // using local + global variable
    printf("Sum = %d\n", sum);

    return 0; // Return statement

}
```

Output:

Sum = 80

5.2.3 Preprocessor & Header Files

1. Preprocessor runs before compilation.
2. Begins with #.
3. #include includes file.
4. #define defines macro.
5. Removes comments.
6. Expands macros.
7. Example: stdio.h.
8. Improves reusability.
9. Saves coding time.
10. Essential for libraries.

5.2.4 Character Set

1. Letters A–Z, a–z.
2. Digits 0–9.
3. Special symbols.
4. White spaces.
5. Escape sequences.
6. Used in identifiers.
7. Used in constants.
8. Case sensitive.
9. ASCII based.
10. Standardized.

5.2.5 Comments

1. Explain code.
2. Not executed.
3. Single line: //.
4. Multi-line: /* */.

5. Improves readability.
6. Helps debugging.
7. Used for documentation.
8. Ignored by compiler.
9. Can be placed anywhere.
10. Essential for teamwork.

Comments are **notes written in a program to explain the code.**

- They **are not executed** by the compiler.
- Used for **documentation** and **readability**.

Types of Comments

1. Single-line Comment

- Begins with `//`
- Used for short explanations.

Example:

```
#include <stdio.h>
int main() {
    int a = 10;    // variable to store a number
    printf("Value = %d", a); // print the value
    return 0;
}
```

Output:

Value = 10

The comments are ignored during execution.

2. Multi-line Comment

- Begins with `/*` and ends with `*/`
- Used for longer explanations.

Example:

```
#include <stdio.h>
int main() {
    /* This program demonstrates
       the use of comments in C
       It prints the value of a variable */
    int a = 20;
    printf("Value = %d", a);
    return 0;
}
```

Output:

Value = 20

All text inside `/* */` is ignored by the compiler.

5.2.6 Identifiers, Keywords, Tokens

1. Identifier = name.
2. Keywords = reserved.
3. Tokens = smallest unit.
4. Cannot use keyword as identifier.
5. Must start with letter.
6. No spaces allowed.
7. Case sensitive.
8. Example keyword: int.
9. Example identifier: total.
10. Tokens include operators.

An **identifier** is a name given by the programmer to a variable, function, or any user-defined item in a program.

Rules for Identifiers:

1. Must start with a letter (A–Z, a–z) or underscore `_`.
2. Can contain letters, digits (0–9), and underscores.
3. Cannot be a **keyword**.
4. Case-sensitive (Var \neq var).
5. Should be meaningful (like age, totalMarks).

Example Program:

```
#include <stdio.h>
int main() {
    int age = 20;          // 'age' is an identifier
    int totalMarks = 95;
    printf("Age = %d, Marks = %d", age, totalMarks);
    return 0;
}
```

Output:

Age = 20, Marks = 95

2. Keywords

Definition:

A **keyword** is a **reserved word** in C with a **special meaning**.

- Cannot be used as an identifier.

Examples:

int, float, if, else, return, while, for, char, void

Example Program:

```
#include <stdio.h>
int main() {
    int num = 10;        // 'int' is a keyword
    if(num > 5) {       // 'if' is a keyword
        printf("Number is greater than 5");
    }
    return 0;
}
```

Output:

Number is greater than 5

3. Tokens

Definition:

Tokens are the **smallest units of a C program**.

Everything in C is made up of tokens.

Types of Tokens:

1. **Keywords** – int, if, return
2. **Identifiers** – age, totalMarks
3. **Constants** – 10, 'A', 3.14
4. **Operators** – +, -, *, /
5. **Special symbols** – {, }, ;

Example Program:

```
#include <stdio.h>
int main() {
    int a = 10;          // 'int', 'a', '=', '10', ';' are tokens
    printf("%d", a);    // 'printf', '(', '"%d"', ',', 'a', ')', ';' are tokens
    return 0;          // 'return', '0', ';' are tokens
}
```

Output:

10

5.2.7 Basic Data Types

1. int → integers.
2. float → decimal.
3. char → character.
4. double → large decimal.
5. void → no value.
6. Each has size.
7. Stored in memory.
8. Used in declarations.

9. Determines storage.
10. Important for variables.

Data types specify the **type of data** a variable can store and the **amount of memory** required.

1. int

Definition:

int is used to store **integer numbers** (whole numbers) without decimals.

Program:

```
#include <stdio.h>
int main() {
    int a = 25;
    printf("Integer value = %d", a);
    return 0;
}
```

Output:

Integer value = 25

2. float

Definition:

float is used to store **decimal (fractional) values**.

Program:

```
#include <stdio.h>
int main() {
    float b = 12.5;
    printf("Float value = %f", b);
    return 0;
}
```

Output:

Float value = 12.500000

3. char

Definition:

char is used to store a **single character**.

Program:

```
#include <stdio.h>
int main() {
    char ch = 'A';
    printf("Character = %c", ch);
    return 0;
}
```

Output:

Character = A

4. double

Definition:

double is used to store **large decimal values** with more precision than float.

Program:

```
#include <stdio.h>
int main() {
    double d = 123.456789;
    printf("Double value = %lf", d);
    return 0;
}
```

Output:

Double value = 123.456789

5.2.8 Constants & Variables

1. Variable value changes.
2. Constant value fixed.
3. Use const keyword.
4. Must be declared.
5. Stored in memory.
6. Has data type.
7. Named using identifiers.
8. Used in expressions.
9. Improves flexibility.
10. Essential in programs.

A variable is a name given to a memory location whose **value can change** during program execution.

Key Points:

1. Variable values can be changed.
2. Must be declared before use.
3. Each variable has a data type.
4. Stored in memory.
5. Used to store user input and results.

Example Program:

```
#include <stdio.h>
int main() {
    int num = 10;    // variable
    printf("Value = %d\n", num);
    num = 20;       // value changed
    printf("New Value = %d", num);
    return 0;
}
```

Output:

```
Value = 10
New Value = 20
```

Constant

Definition:

A constant is a value that **cannot be changed** during program execution.

Key Points:

1. Constant value is fixed.
2. Declared using const keyword.
3. Cannot modify after declaration.
4. Makes program safe.
5. Used for fixed values like PI.

Example Program:

```
#include <stdio.h>
int main() {
    const float PI = 3.14;    // constant
    printf("PI = %f", PI);
    return 0;
}
```

Output:

PI = 3.140000

5.2.9 Type Specifiers

1. Modify data type.
2. short → less memory.
3. long → more memory.
4. signed → + and -.
5. unsigned → only +.
6. Used with int/char.
7. Improves range.
8. Affects storage.
9. Improves precision.
10. Example: long int.

5.2.10 Statements

1. Simple statement → single line.
2. Ends with semicolon.
3. Compound statement → block {}.
4. Used in loops.
5. Used in conditions.
6. Groups statements.
7. Improves readability.
8. Allows multiple operations.
9. Required in functions.
10. Structured code.

5.2.11 Operators

1. Arithmetic operators.
2. Relational operators.
3. Logical operators.
4. Assignment operators.
5. Unary operators.
6. Conditional operator.
7. Used in expressions.
8. Perform calculations.
9. Help decision making.
10. Essential in programming.

Operators are symbols used to perform operations on variables and values.

1. Arithmetic Operators

Definition: Used to perform mathematical calculations.

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication

/ Division
% Modulus

Program:

```
#include <stdio.h>
int main() {
    int a = 10, b = 3;
    printf("Add = %d\n", a + b);
    printf("Sub = %d\n", a - b);
    printf("Mul = %d\n", a * b);
    printf("Div = %d\n", a / b);
    printf("Mod = %d\n", a % b);
    return 0;
}
```

Output:

Add = 13
Sub = 7
Mul = 30
Div = 3
Mod = 1

2. Relational Operators

Definition: Used to compare two values. Result is true (1) or false (0).

Operator	Meaning
>	Greater than
<	Less than
==	Equal to
!=	Not equal
>=	Greater or equal
<=	Less or equal

Program:

```
#include <stdio.h>
int main() {
    int a = 5, b = 10;
```

```
    printf("%d\n", a > b);
    printf("%d\n", a < b);
    return 0;
}
```

Output:

```
0
1
```

3. Logical Operators

Definition: Used to combine conditions.

Operator	Meaning
&&	AND

!	NOT
---	-----

Program:

```
#include <stdio.h>
int main() {
    int a = 5, b = 10;
    printf("%d\n", (a<10 && b>5));
    printf("%d\n", (a>10 || b>5));
    printf("%d\n", !(a>3));
    return 0;
}
```

Output:

```
1
1
0
```

4. Assignment Operators

Definition: Used to assign values.

Operator	Example
=	a = 5
+=	a += 2
-=	a -= 2
*=	a *= 2
/=	a /= 2
%=	a %= 2

Program:

```
#include <stdio.h>
int main() {
    int a = 5;
    a += 3;
    printf("%d", a);
    return 0;
}
```

Output:

8

5. Unary Operators

Definition: Operate on one operand.

Operator	Meaning
++	Increment
--	Decrement
!	Logical NOT

Program:

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

```
    printf("%d\n", --a);
    return 0;
}
```

Output:

```
6
5
```

6. Conditional Operator (?:)

Definition: Short form of if-else.

Syntax:

```
condition ? value_if_true : value_if_false;
```

Program:

```
#include <stdio.h>
int main() {
    int a = 10, b = 20;
    int max = (a > b) ? a : b;
    printf("Max = %d", max);
    return 0;
}
```

Output:

```
Max = 20
```

5.2.12 I/O Functions

1. printf() output.
2. scanf() input.

3. Uses format specifiers.
4. %d, %f, %c.
5. Defined in stdio.h.
6. Used for communication.
7. Easy to use.
8. Supports multiple values.
9. Important for user input.
10. Used in almost all programs.

5.2.13 Selection Statements

1. Used for decisions.
2. if statement.
3. if-else statement.
4. else-if ladder.
5. Nested if.
6. switch statement.
7. Used based on condition.
8. Improves logic.
9. Controls program flow.
10. Essential in real programs.

5.2.14 Iteration Statements

1. Used for repetition.
2. while loop.
3. do-while loop.
4. for loop.
5. Nested loops.
6. Saves time.
7. Reduces code size.
8. Used in counting.
9. Used in arrays.
10. Important control structure.

Iteration statements are used to **repeat a block of code** multiple times until a condition becomes false.

Types:

- while loop
- do-while loop
- for loop
- Nested loops

1. while Loop

Definition:

The while loop checks the condition **before** executing the loop body. If the condition is true, the loop runs.

Program:

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

Output:

1 2 3 4 5

2. do-while Loop

Definition:

The do-while loop executes the loop body **at least once**, then checks the condition.

Program:

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

```
        i++;
    } while(i <= 5);
    return 0;
}
```

Output:

1 2 3 4 5

3. for Loop

Definition:

The for loop is used when the number of repetitions is known. It has initialization, condition, and increment/decrement in one line.

Program:

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

Output:

1 2 3 4 5

4. Nested Loops

Definition:

A loop inside another loop is called a nested loop. Used for patterns, tables, and matrices.

Program (Print 3×3 pattern):

```
#include <stdio.h>
int main() {
    for(int i = 1; i <= 3; i++) {
        for(int j = 1; j <= 3; j++) {
            printf("* ");
        }
    }
}
```

```
    }  
    printf("\n");  
}  
return 0;  
}
```

Output:

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

5.2.15 Arrays

1. Collection of same type.
2. 1D and 2D arrays.
3. Indexed from 0.
4. Stores multiple values.
5. Used for matrices.
6. Fixed size.
7. Improves memory use.
8. Easy access using index.
9. Used in loops.
10. Important data structure.

5.2.16 Strings & Functions

1. String = character array.
2. Ends with '\0'.
3. Stored in char array.
4. Used for text.
5. strlen() length.
6. strcat() join.
7. strcmp() compare.
8. strcpy() copy.
9. strlwr()/strupr() case change.

STRING IN C

Definition of String

1. A string is a group of characters.
2. It is stored in a **character array**.
3. Every string ends with a special character **\0 (null character)**.
4. Strings are used to store text like names and messages.
5. String functions are available in the header file **string.h**.

Example:

```
char name[10] = "Hello";
```

String Functions in C

1. strlen()

Definition:

strlen() returns the **length of a string** (number of characters), excluding \0.

Program:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[20] = "Computer";
    printf("Length = %lu", strlen(str));
    return 0;
}
```

Output:

Length = 8

2. strcat()

Definition:

strcat() joins two strings. The second string is added to the end of the first.

Program:

```
#include <stdio.h>
#include <string.h>
int main() {
    char a[20] = "Good ";
    char b[20] = "Morning";
    strcat(a, b);
    printf("Result = %s", a);
    return 0;
}
```

Output:

Result = Good Morning

3. strcmp()

Definition:

strcmp() compares two strings.

- Returns **0** if equal
- Returns **non-zero** if not equal

Program:

```
#include <stdio.h>
#include <string.h>
int main() {
    char a[20] = "apple";
    char b[20] = "apple";
    if(strcmp(a, b) == 0)
        printf("Strings are equal");
    else
        printf("Strings are not equal");
    return 0;
}
```

Output:

Strings are equal

4. strrev()

Definition:

strrev() reverses the string.

Program:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[20] = "hello";
    strrev(str);
    printf("Reverse = %s", str);
    return 0;
}
```

Output:

Reverse = olleh

5. strcpy()

Definition:

strcpy() copies one string into another.

Program:

```
#include <stdio.h>
#include <string.h>
int main() {
    char a[20];
    char b[20] = "C Language";
    strcpy(a, b);
    printf("Copied String = %s", a);
    return 0;
}
```

```
}
```

Output:

Copied String = C Language

6. strlwr()

Definition:

strlwr() converts all characters of a string to **lowercase**.

Program:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[20] = "HELLO";
    strlwr(str);
    printf("Lowercase = %s", str);
    return 0;
}
```

Output:

Lowercase = hello

7. strupr()

Definition:

strupr() converts all characters of a string to **uppercase**.

Program:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[20] = "hello";
    strupr(str);
    printf("Uppercase = %s", str);
}
```

```
    return 0;  
}
```

Output:

Uppercase = HELLO