Fundamentals of Programming Language 'C'

UNIT 1

C Programming

- It is a structured programming language and you can use the skills learned in C to master other programming languages.
- You can use C program to write efficient codes and develop robust projects.
- C is a low-level language and you can use it to interact more directly with the computer's hardware and memory.

Introduction to Programming

Programming is the process of designing and building an executable computer program to accomplish a specific computing task. It involves writing code in a programming language that a computer can understand and execute. The purpose of programming is to create software that performs tasks, solves problems, or provides entertainment and utility.

Algorithm

- An **algorithm** is a finite set of well-defined instructions or steps used to solve a particular problem or perform a task.
- It answers the **"how to solve"** part of a problem.

Example of an Algorithm:

Problem: Find the sum of two numbers.

Algorithm:

1. Start
2. Read number A
3. Read number B
4. Calculate SUM = A + B
5. Display SUM
6. Stop

C Program

```
#include <stdio.h>

int main() {
```

```
printf("Hello, World! \n");90

 return 0; }
```

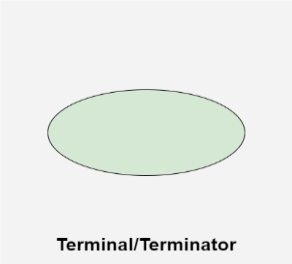
- The library functions must be loaded in any C program. The **"#include"** statement is used to include a header file. It is a **"preprocessor directive"**.
- For example, printf() and scanf() functions are needed to perform console I/O operations. They are defined in the stdio.h file. Hence, you invariably find **#include <stdio.h>** statement at the top of any C program.

Introduction to Flowcharts

The flowcharts are simple visual tools that help us understand and represent processes very easily. They use shapes like arrows, rectangles, and diamonds to show steps and decisions clearly. If someone is making a project or explaining a complex task, flowcharts can make complex ideas easier to understand.

Symbols used in Flowchart Designs

1. **Terminal/Terminator**
   The oval symbol indicates Start, Stop and Halt in a program's logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



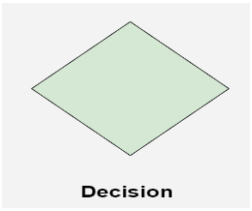
2. Input/Output
A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.

3. Action/Process

A box represents arithmetic instructions, specific action or operation that occurs as a part of the process. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action/process symbol.

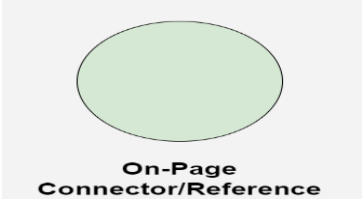
4. Decision

Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



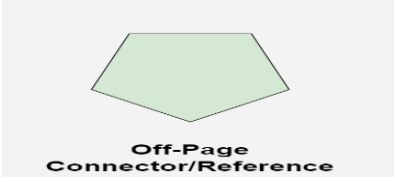
5. On-Page Connector/Reference

Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. connectors are used to indicate a jump from one part of the flowchart to another without drawing long or complicated lines. On-Page Connector is represented by a small circle



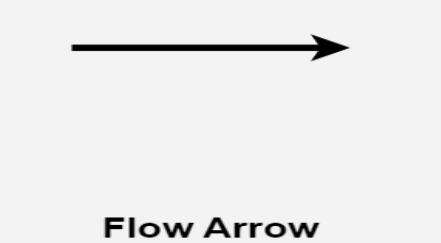
6. Off-Page Connector/Reference

Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. connectors are used to indicate a jump from one part of the flowchart to another without drawing long or complicated lines. Off-Page Connector is represented by a pentagon.



7. Flow lines

Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.



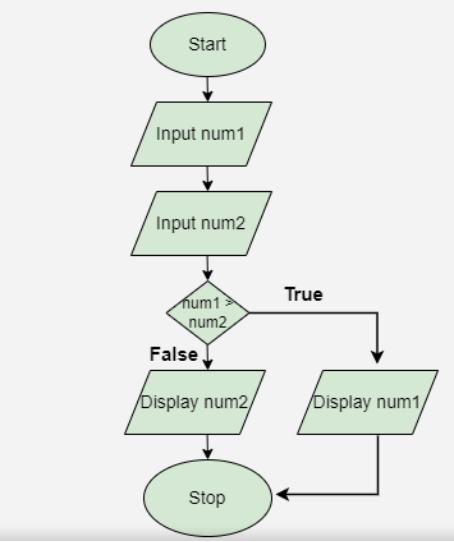
Rules For Creating a Flowchart

A flowchart is a graphical representation of an algorithm. It should follow some rules while creating a flowchart

- **Rule 1:** Flowchart opening statement must be ‘start’ keyword.
- **Rule 2:** Flowchart ending statement must be ‘end’ keyword.
- **Rule 3:** All symbols in the flowchart must be connected with an arrow line.
- **Rule 4:** Each decision point should have two or more distinct outcomes.
- **Rule 5:** Flow should generally move from top to bottom or left to right.

Example of a Flowchart

Draw a flowchart to input two numbers from the user and display the largest of two numbers.



Below is the explanation of the above flowchart:

- **Start**: The process begins with the **Start** symbol, indicating the start of the program.
- **Input num1**: The first number, represented as **num1**, is entered.
- **Input num2**: The second number, represented as **num2**, is entered.
- **Decision (num1 > num2)**: A decision point checks if **num1** is greater than **num2**.
    - If **True,** the process moves to the next step where **num1** will be displayed.
    - If **False**, the process moves to display **num2**.
- **Stop**: The process ends with the **Stop** symbol, signaling the conclusion of the program.

Compiler

A **compiler** is a special program that translates the entire high-level source code of a program into **machine code (binary code)** in one go, before the program is executed. It reads the complete code, checks it for errors, and then converts it into an **executable file** (like .exe in Windows). This machine code is directly understood by the computer's hardware, which means that once a program is compiled successfully, it can be run **multiple times without recompiling**. Since the translation is done in advance, compiled programs usually run **much faster**. However, one major limitation is that if there are any errors in the source code, the compiler will only show them **after scanning the entire program**, which can make it slightly harder to debug. Common examples of compiled languages include **C**, **C++**, and **Java** (the Java compiler converts source code to bytecode).

Interpreter

An **interpreter** is a type of language processor that translates and executes a program **line by line**, rather than all at once. It reads one statement of the high-level source code, converts it into machine code, **executes it immediately**, and then moves to the next line. If an error is found in a particular line, the interpreter **stops execution** right there and reports the error, allowing the programmer to fix issues quickly and easily. This makes interpreters especially useful for beginners and for debugging during development. However, since the interpreter must translate code **every time the program is run**, it is usually **slower** than compiled execution. Also, no standalone executable file is created. Examples of interpreted languages include **Python**, **JavaScript**, **Ruby**, and **PHP**.

Overview of C – Introduction

What is C?

C is a **high-level**, **general-purpose** programming language developed in the early 1970s by **Dennis Ritchie** at **Bell Labs**. It was originally created to develop the **UNIX operating system**, and since then, it has become one of the most widely used programming languages in the world.

Key Features:

- **Simple and powerful**
- **Structured and procedural**
- Supports **low-level memory access** (using pointers)
- Portable across platforms
- Foundation for many modern languages like **C++, Java, and Python**

Basic Structure of a C Program

```
#include <headerfile>   // 1. Preprocessor Directive

int main() {            // 2. Main Function

   // 3. Variable Declarations

   // 4. Program Statements

   return 0;          // 5. Return Statement

}
```

Sample C Program

Here is a **simple C program** to print "Hello, World!":

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

int main() {          // Main function - starting point of the program
   printf("Hello, World!\n");  // Output statement
   return 0;         // Return 0 indicates successful execution
}
```

Programming Style in C

Writing readable and maintainable code is important. Follow these style guidelines:

- Use **meaningful variable names** (e.g., totalMarks, age)
- Use proper **indentation and spacing**
- Add **comments** to explain code logic (// single-line or /* multi-line */)
- Use **consistent naming conventions** (e.g., camelCase or snake_case)
- Break long code into **functions** for clarity

Constants, Variables and data Types

C Tokens

C tokens are the basic building blocks of any C program. They include keywords, identifiers, constants, string literals, and operators. In essence, tokens are the smallest units of a C program.

Types of C Tokens:

1. **Keywords**
2. **Identifiers**
3. **Constants**
4. **Operators**
5. **Special symbols**
6. **Strings**

1. Keywords

Keywords are reserved words that have special meanings in C. They cannot be used as identifiers (names for variables, functions, etc.).

Examples:

```
int main() {
    int number; // 'int' and 'return' are keywords
    return 0;
}
```

Common Keywords in C:

- int , return, if, else, while, for, break, continue, void

2. Identifiers

Identifiers are names given to various program elements such as variables, functions, arrays, etc.

Rules for Identifiers:

- Must start with a **letter (A-Z/a-z)** or an **underscore (_)**.
- Can contain **letters, digits, and underscores**.
- Cannot use **spaces or special characters**.
- Cannot be the **same as a keyword**.

Examples:

int age;

float salary;

void displayMessage();

3. Constants

Constants are fixed values that do not change during the execution of a program. They can be of various types like integer constants, floating-point constants, character constants, and enumeration constants.

Examples:

int age = 30; // Integer constant

float pi = 3.14; // Floating-point constant

char grade = 'A'; // Character constant

const int DAYS_IN_WEEK = 7; // Constant variable

4. String Literals

String literals are sequences of characters enclosed in double quotes.

Examples:

char name[] = "Alice";

printf("Welcome, %s", name);

5. Operators

Operators are symbols that specify operations to be performed on operands.

⚙️ Types of Operators:

Type	Examples	Description
Arithmetic	`+`, `-`, `*`, `/`, `%`	Addition, Subtraction, etc.
Relational	`==`, `!=`, `<`, `>`	Comparisons
Logical	`&&`, ` ` `	
Assignment	`=`, `+=`, `-=`	Assigning values
Increment/Decrement	`++`, `--`	Increase/Decrease by 1

Examples:

int a = 5, b = 10, c;

c = a + b; // '+' is an arithmetic operator

if (a < b) // '<' is a relational operator

printf("a is less than b");

6. Special Symbols

Special symbols are used to perform various operations and include characters like ;, {}, (), [], #, etc.

Common Special Symbols:

Symbol	Meaning
`{}`	Block of code
`[]`	Array brackets
`()`	Function brackets
`;`	Statement terminator
`#`	Preprocessor directive
`,`	Separator
`\`	Escape character

Examples:

```
int main() { // '{' and '}' are special symbols
    int arr[5]; // '[]' is a special symbol
    return 0; // ';' is a special symbol
}
```

7. Separators

Separators are used to separate statements and tokens in a program. The most common separator in C is the semicolon (;).

Examples:

```
int a = 10; // ';' is a separator
int b = 20;
```

Variables in C

Definition:

A **variable** is a **named memory location** used to store data. The value stored in a variable can **change during the execution** of the program.

Key Characteristics of Variables:

- A variable has:
  - A **name** (identifier)
  - A **data type**
  - A **value**
- Must be **declared** before use.
- Memory is allocated based on the data type.

Syntax of Variable Declaration:

```
data_type variable_name;
```

Or, with initialization:

```
data_type variable_name = value;
```

Example:

```
int age;          // Declaration
age = 18;         // Assignment

float weight = 55.5;  // Declaration + Initialization
char grade = 'A';
```

Rules for Naming Variables:

1. Must start with a **letter (A–Z, a–z)** or an **underscore (_)**
2. Can contain **letters, digits (0–9), and underscores**
3. **No spaces or special characters** allowed
4. Cannot use **keywords** (e.g., int, float, return)

Invalid Variable Names:

- 1value (starts with digit)
- total-marks (contains -)
- float (keyword)

Valid Variable Names:

- value1
- total_marks
- _counter

Data Types in C

Definition:

A **data type** specifies the **type of data** a variable can hold, and determines how much **memory** will be allocated for it.

A. **Primary (Basic) Data Types**

🔹 Primary Data Types Table

Data Type	Description	Size (Bytes*)	Format Specifier	Example Values
`int`	Stores whole numbers (integers)	2 or 4	`%d`	`5`, `-20`
`float`	Stores decimal (floating-point) numbers	4	`%f`	`3.14`, `-0.5`
`double`	Double-precision floating-point number	8	`%lf`	`12.123456`, `-55.01`
`char`	Stores a single character	1	`%c`	`'A'`, `'9'`

Example

```
#include <stdio.h>
int main() {
    int age = 25;
    float height = 5.9;
    double weight = 72.456;
    char grade = 'A';
    printf("Age: %d\n", age);
    printf("Height: %f\n", height);
    printf("Weight: %lf\n", weight);
    printf("Grade: %c\n", grade);
    return 0;
}
```

Output

```
Age: 25
Height: 5.900000
Weight: 72.456000
Grade: A
```

B. Derived Data Types

Derived data types are **built from basic (primitive) data types** and allow more complex data handling. These include arrays, pointers, functions, and structures.

- **Array**: Collection of elements of same type
  Example: `int arr[5];`
- **Pointer**: Stores memory address
  Example: `int *ptr;`
- **Function**: Group of statements performing tasks
  Example: `void greet();`
- **Structure (`struct`)**: Group of variables of different types
  Example:

1. Array

An **array** is a collection of **multiple elements of the same data type** stored in **contiguous memory locations**.

```
Syntax:
data_type array_name[size];
```

Example:

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

2. Pointer

A **pointer** is a variable that **stores the memory address** of another variable.

Syntax:

```
data_type *pointer_name;
```

Example:

```
int num = 10;
int *ptr = &num;
```

3. Function

A **function** is a **block of code** that performs a specific task. Functions help in **modular programming** and code reuse.

Syntax:

```
return_type function_name(parameters) {
    // code
```

```
}
```

Example:

```
void greet() {
    printf("Hello, Students!\n");
}
```

4. Structure (`struct`)

A **structure** is a **user-defined data type** that allows grouping of **variables of different types** under one name.

Syntax:

```
struct structure_name {
    data_type member1;
    data_type member2;
    ...
};
```

Example:

```
#include <stdio.h>

#include <string.h>

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

Declaration of Variables

Declaring a variable means **telling the compiler** the **name** of the variable and the **type of data** it will hold.

Syntax:

data_type variable_name;

You can also assign a value while declaring:

data_type variable_name = value;

Examples:

int age; char grade = 'A';