1. **Structure of a C program**
2. The structure of a C program follows a specific layout, which helps organize code efficiently and logically. Here's a basic breakdown of the typical structure of a C program:

### **Basic Structure of a C Program**

#include <stdio.h> // Preprocessor Directive

// 1. Global Declarations (optional)

int globalVar = 10; // Example of a global variable

// 2. Function Declarations (Prototypes)

void greet(); // Function prototype

// 3. Main Function - Entry point of the program

int main() {

// Local Variable Declarations

int num = 5;

// Statements

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

greet(); // Function call

return 0; // Exit status

}

// 4. Function Definitions

void greet() {

printf("Welcome to C programming!\n");

}

### **Explanation of Each Part**

| Part | Description |
| --- | --- |
| **#include <stdio.h>** | Preprocessor directive to include Standard Input/Output library. |
| **Global Declarations** | Variables or constants available throughout the program (optional). |
| **Function Prototypes** | Declarations of functions to inform the compiler before their use. |
| **main() Function** | **Required** function where program execution begins. |
| **Statements in main** | Code that does the actual task — variable declarations, logic, etc. |
| **Function Definitions** | Actual code of the user-defined functions. |

1. **Data types and variables**
2. data type defines **what kind of data** a variable can hold and **how much memory** will be allocated for it. Purpose of Data types:

* To define **type of data** (e.g., integer, float, character).
* To define **size of memory** needed (1 byte, 4 bytes, etc.).
* To define **valid operations** that can be performed on the data

### **Types of data type :**

**Basic/Primary Data Type:**

| Data Type | Description | Size (Typical 32-bit) | Format Specifier | Range |
| --- | --- | --- | --- | --- |
| int | Integer | 4 bytes | %d | -2,147,483,648 to 2,147,483,647 |
| float | Floating point | 4 bytes | %f | ~1.2E-38 to 3.4E+38 |
| double | Double precision float | 8 bytes | %lf | ~2.3E-308 to 1.7E+308 |
| char | Character (ASCII) | 1 byte | %c | -128 to 127 |

**Derived Data Type: (made from basic types)**

| Type | Description | Example |
| --- | --- | --- |
| array | Collection of similar types | int marks[5]; |
| pointer | Address of another variable | int \*ptr; |
| function | Code block that returns data | int add(int a, int b) |

**User-Defined Data Types: (Made by programmers to group different type)**

| Type | Description | Example |
| --- | --- | --- |
| struct | Group of variables | struct Student {...} |
| union | Like struct but shared memory | union Data {...} |
| enum | Enumeration of named constants | enum Color {Red, Blue} |

### **Void Data Type: (No Value – used mostly for functions that don’t return anything)**

## **💡 **Real-World Analogy****

Think of **data types** like different boxes:

* A **char** box holds **a letter**.
* An **int** box holds **a whole number**.
* A **float** box holds **a decimal**.
* A **struct** box is like a **toolbox** — it holds different kinds of items (int, float, char, etc.).

### **Types of data type Modifiers:**

**Modifiers alter the size or range of basic data types.**

### **List of Modifiers:**

* signed
* unsigned
* short
* long

### **Integer Types with Modifiers**

| Data Type | Size (in bytes) | Range |
| --- | --- | --- |
| short int | 2 | -32,768 to 32,767 |
| unsigned short int | 2 | 0 to 65,535 |
| unsigned int | 4 | 0 to 4,294,967,295 |
| signed int | 4 | -2,147,483,648 to 2,147,483,647 |
| long int | 4 or 8 | ~-2.1B to 2.1B (32-bit) / more (64-bit) |
| unsigned long int | 4 or 8 | 0 to 4.29B (32-bit) / more |
| long long int | 8 | -9 quintillion to +9 quintillion |
| unsigned long long | 8 | 0 to 18 quintillion |

### **Floating-Point Types;**

| Data Type | Size | Precision | Range |
| --- | --- | --- | --- |
| float | 4 | ~6 digits | ±1.2E−38 to ±3.4E+38 |
| double | 8 | ~15 digits | ±2.3E−308 to ±1.7E+308 |
| long double | 10 or 16 | ~19 digits | System dependent |

### ****Variables in C** - A variable is a named memory location used to store data. You must declare a variable before using it.**

**Syntax:**

data\_type variable\_name;

**Example:**

int age;

float salary;

char grade;

1. **Constants and literals**
2. **Constant - A constant is a named variable whose value cannot be changed once assigned.**

You define it like a variable, but tell the compiler: “Lock this value!”

### ✅ ****Ways to Declare Constants:****

#### 1. Using const keyword:

const int age = 18; // Can't change age now

#### 2. Using #define preprocessor:

#define PI 3.14159 // PI is a constant

Literal - A **literal** is a fixed value used directly in the code. It’s **not stored in a variable** — it's just written as it is.

| Type | Example |
| --- | --- |
| Integer Literal | 10, -20 |
| Floating Literal | 3.14, -0.001 |
| Character Literal | 'A', 'z' |
| String Literal | "Hello" |
| Boolean Literal | 0 (false), 1 (true) — C has no built-in true/false, but these are used |

1. **Operators and expressions**
2. **Operators** are symbols that tell the compiler to **perform specific operations** on variables and values.

Think of operators as the action-doers (like +, -, =, ==, etc.)

1. What Are Expressions in C - An expression is a combination of operands (variables, literals) and operators that produces a value.

## ****Types of Operators in C****

### 🔹 1. ****Arithmetic Operators -****Used for basic math.

| Operator | Meaning | Example |
| --- | --- | --- |
| + | Addition | a + b |
| - | Subtraction | a - b |
| \* | Multiplication | a \* b |
| / | Division | a / b |
| % | Modulus (remainder) | a % b |

### 🔸 2. ****Relational (Comparison) Operators -**** Used to compare values.

| Operator | Meaning | Example |
| --- | --- | --- |
| == | Equal to | a == b |
| != | Not equal to | a != b |
| > | Greater than | a > b |
| < | Less than | a < b |
| >= | Greater than or equal | a >= b |
| <= | Less than or equal | a <= b |

### 🔹 3. ****Logical Operators -**** Used in conditions (e.g., if statements).

| Operator | Meaning | Example |
| --- | --- | --- |
| && | Logical AND | a > 0 && b > 0 |
| ! | Logical NOT | !a (opposite of a) |

### 🔸 4. ****Assignment Operators -**** Used to assign values.

| Operator | Meaning | Example |
| --- | --- | --- |
| = | Assign | a = 10 |
| += | Add and assign | a += 5 (same as a = a + 5) |
| -= | Subtract and assign | a -= 2 |
| \*= | Multiply and assign | a \*= 3 |
| /= | Divide and assign | a /= 4 |
| %= | Modulus and assign | a %= 2 |

### 🔹 5. ****Increment & Decrement Operators -**** Used to increase or decrease a value by 1.

| Operator | Meaning | Example |
| --- | --- | --- |
| ++ | Increment by 1 | i++ or ++i |
| -- | Decrement by 1 | i-- or --i |

### ****🔸6. Bitwise Operators (used in low-level programming)****

| Operator | Description |
| --- | --- |
| & | Bitwise AND |
| ^ | Bitwise XOR |
| ~ | Bitwise NOT |
| << | Left Shift |
| >> | Right Shift |